A BEHAVIOR-BASED KERNEL LEVEL AUTHENTICATION MECHANISM FOR PROTECTING SYSTEM SERVICES AGAINST MALICIOUS CODE ATTACKS

A THESIS

Submitted to Pondicherry University in partial

fulfilment of the requirements for the award of the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

By

K. MUTHUMANICKAM

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
PONDICHERRY ENGINEERING COLLEGE
PUDUCHERRY – 605 014
INDIA

DECEMBER 2016

Dr. E. ILAVARASAN, M.Tech., Ph.D.,
Professor
Department of Computer Science and Engineering
Pondicherry Engineering College
Puducherry – 605 014.

CERTIFICATE

Certified that this thesis entitled **"A BEHAVIOR-BASED KERNEL LEVEL AUTHENTICATION MECHANISM FOR PROTECTING SYSTEM SERVICES AGAINST MALICIOUS CODE ATTACKS"** submitted for the award of the degree of **DOCTOR OF PHILOSOPHY** in **COMPUTER SCIENCE AND ENGINEERING** of the Pondicherry University, Puducherry is a record of original research work done by **Mr. K. MUTHUMANICKAM** during the period of study under my supervision and that the thesis has not previously formed the basis for the award to the candidate of any Degree, Diploma, Associateship, Fellowship or other similar titles. This thesis represents independent work on the part of the candidate.

(Dr. E. ILAVARASAN)
Supervisor

Date : 13.12.2016

Place: Puducherry

ACKNOWLEDGMENT

It is my great pleasure to express my deep sense of gratitude to my supervisor **Dr. E. Ilavarasan,** Professor, Department of Computer Science and Engineering, Pondicherry Engineering College, Puducherry, for his invaluable guidance as well as his timely advice during the period of the research work. His consistent encouragement and personal attention are accountable for the successful completion of this thesis.

I express my sincere thanks to **Dr. P. Dananjayan**, Principal, Pondicherry Engineering College and **Dr. T. Sundararajan** and **Dr. D. Govindarajulu**, Former Principals, Pondicherry Engineering College for their support and permission to carry out my research work. I am grateful to **Dr. M. Sugumaran**, Professor and Head, Department of Computer Science and Engineering and **Dr. D. Loganathan,** Professor and Former Head, Department of Computer Science and Engineering, for their valuable suggestions, advice and support during the entire course period.

I sincerely thank my Doctoral Committee members, **Dr. V. Vijayalakshmi**, Associate Professor, Department of Electornics and Communication Engineering, Pondicherry Engineering College and **Dr. V. Ramalingam**, Professor, Department of Computer Science and Engineering, Annamalai University for providing valuable comments and constructive suggestions towards improving the quality of this research. And I would also like thank to **Dr. S. Himavathi**, Professor and Dean (Research), Pondicherry Engineering College for her moral support during the entire course period.

I am very much thankful to my fellow researcher **Mrs. N. Danapaquiame** for her continuous encouragement and support throughout this research, has helped in the successful completion of this thesis.

I express sincere and heartfelt thanks to my family members, especially to my father **Shri. V. Krishnan**, my mother **Shrimathi. K. Chellamma**l, my brothers **Mr. K. Nagarajan**, **Mr. K. Ravichandran, Mr. K. Kumar**, **Mr. K. Chandrasekar**, my daughter **M. B. Shivaaniritu** and my beloved wife **Mrs. R. Bhuvaneswari** and friends for their unlimited love support during every stage of my research period.

K. MUTHUMANICKAM

Abstract

Computer security has long been subjected to protect either an important end-system or an entire network of hosts. Despite the latest advances in safeguarding the security of end-system and network, malicious attacks on a large network constantly become a serious threat to protecting data (data privacy) and service integrity. There are two main problems that may lead to security risks in a large networked environment. First, poor security policies and system configurations that acts as entry point to permit attackers to easily bypass the predefined security defences, and seconly, hidden vulnerabilities which permit an attacker to execute various attacks remotely against vulnerable servers and workstations.

Advanced stealthy attacks on a large network is only possible by the presence of increased vulnerable hosts in the network. The existence of vulnerabilities in the Internet connected workstations and servers, that compromise the network, are envitable. Malware writers aims to control the functionalities of the operating system of the victim computer by executing malicious programs through vulnerabilities found either in the application or system. As Windows is one of the most popular and widely used operating system in the modern online world for personal computers, it becomes most malware ridden platform. Therefore, the security compromise of even a single operation in a system will question the overall system security, including the security assurance of all currently running applications within the system. Furthermore, the proper control and management of ensuring the security of individual applications within a computer system makes the security analysis a challenging task. Therefore, the compromise of just a single computer on a network may permit an intruder to gain access to important system resources and distrupt the normal operations the entire network.

Though there are different security defense mechanisms such as Discretionary Access Control protection that protect system resources from unauthorized access and the Mandatory Access Control mechanism which ensures the safe execution of untrustworthy applications, they are not sufficient to offer complete protection against stealthly advanced malicious code attacks. This research work mainly focused on improving protection level and security incident reponse capabilities of an end-system or workstation.

First, a Graph-based static Malware Detection approach namely, GraMD has been proposed for detecting malware attacks. The proposed GraMD approach has been compared with the existing well-known mechanisms for malware detection. The experimental simulation results shows that the proposed GraMD approach outperforms than the existing approaches for malware detection. However, the static malware detection approches fails to detect unknown malware attacks and hence found not suitable for trusted computing environment. This has led to the development of a behaviour-based dynamic malware detection approach at user-mode namely, User-mode Malware Detection (UMDetect) for Windows has been developed to provide better malware detection and prevention than the existing techniques. The experimental results shows that UMDetect outperforms than existing mecahnisms with improved security against both known and unknown malicious code attacks. However, UMDetect fails to detect and prevent stealthy malware attacks which incorporate masquerading technique to evade its footprints and malware that directly target higher level Application Programming Interface functions to hook kernel level data structures of the Window operating system.

Currently, advanced stealthy malware writers make use of rootkit technique to mask malware footprints from Antivirus software. This makes malicious code attacks difficult to detect. Though there exist different algorithms for detecting the presence of hidden information of a malicious executable, significant research has not been carried out to effectively apply such algorithms to optimize malware detection technique. Therefore, to discover and list all hidden information of a malicious executable, a novel algorithm namely, Concealed Processes and services Discovery Algorithm (CoPDA) that relies on cross-check based technique has been proposed. The CoPDA algorithm discovers all hidden footprints of a malicious executable by comparing higher-level information about running processes and services against its lower-level information that is obtained from kernel of the operating system. The proposed CoPDA algorithm has been compared with the existing algorithms for detecting hidden malicious code by simulation. The experiments have been conducted using a standard real-time datasets and some real-time anti-malware detection tools. Experimental results shows that CoPDA outperforms against the existing techniques and anti-rootkit detection tools based on their false positives, true positives, precision rate, and detection accuracy.

As the existing conventional authorization techniques lack a reliable and strong process authentication policies against stealthy malicious code attacks, a behaviour-based kernel level Process Authentication Mechanism (PAM) has been proposed. The Proposed PAM mechanism improves the security strength of the kernel of the operating system by incorporating user-mode information through the implemented CoPDA algorithm for discovering all suspicious processes of a malicious executable and kernel-mode information for authenticating each identified suspicious process during run-time. The effectiveness of the proposed PAM mechanism has been evaluated by conducting various simulation experiments using real-time datasets and standard benchmarks. The experimental results show that PAM outperforms than the existing widely used anti-rootkit detection tools and techniques found in the literature in terms of false positives, execution time, accuracy, and performance overhead. As a runtime mechanism, PAM includes secret authentication information and an authentication module to ensure high security assurance.

The limitations of process authentication mechanism for preventing malicious code attacks have also been identified and presented for further exploration in this thesis.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

ACRONYMS	ABBREVIATIONS
ACL	Access Control List
ACM	Access Control Matrix
ANIDS	Anomaly based Network Intrusion Detection System
API	Application Programming Interface
AR	Accuracy Rate
ASLR	Address Layout Randomization
CIG	Cretential Information Generator
CIL	Credential Information List
CoPDA	Concealed Process and services Discovery Algorithm
CRSS	Client server Runtime SubSystem
DA	Detection Accuracy
DAC	Discreationary Access Control
DACL	Discreationary Access Control List
DEP	Data Execution Prevention
DLL	Dynamic Link Library
DR	Detection Rate
DRM	Digital Rights Management
EAT	Export Address Table
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate

ACRONYMS	ABBREVIATIONS
FTP	File Transfer Protocol
HIDS	Host Intrusion Detection System
HMAC	Hashed Message Authentication Code
HMM	Hidden Markov Model
IAT	Import Address Table
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IHD	IAT Hook Detector
ILT	Import Lookup Table
IOCTL	Input Output Control
IRC	Internet Relay Chat
KPP	Kernel Patch Protection
LCSA	Longest Common Subsequence Algorithm
LHC	Inline Hook Detector
MAC	Message Authentication Code
MDS	Malware Detection System
MTR	Security Monitor
MZ	MS-DOS Header
NIDS	Network Intrusion Detection System
OS	Operating System
PAIDS	Proximity Assisted Intrusion Detection System
PAM	Process Authentication Mechanism
PE	Portable Executable

ACRONYMS	ABBREVIATIONS
PF	Perservation Function
PID	Process Indentifier
PMP	Protected Media Path
PR	Precision Rate
RAM	Random Access Memory
ROC	Receiver Operating Characteristic
RVA	Relative Virtual Address
SCM	System Call Monitoring
SMTP	Simple Mail Transfer Protocol
SPC	Secret Process Credential
SPS	Security Policy Specification
SPT	Shadow Page Table
SSDT	System Service Descriptor Table
SVM	Support Vector Machine
TEB	Thread Environment Block
TP	True Positive
TPM	Trusted Patform Module
TPR	True Positive Rate
WHIPS	Windows Host based Intrusion Prvention System
WOW64	Windows 32-bit on Windows 64-bit
RDTSC	Read Time Stamp
WDK	Windows Driver Development Tool Kit

CHAPTER 1

INTRODUCTION

The continuing growth of Internet connected devices will drive malware authors to use either unpatched system or software vulnerabilities as a way to point a full-blown attack. Designing and maintaining a trusted and secure system environment is increasingly more important as data flows more freely through the interconnected devices. Though many vendors, import different security features into their product, they cannot design a complete, secure and trustworthy system to handle current computing environments. Malware writers usually look for either system vulnerabilities or application vulnerabilities to deposit their code into the victim computer. There are three important issues that the defenders may encounter to fix. First, the application process has more flexibility to carry out their illegal operations on the victim computer. Second, both malware and security system can run in the same execution environment. Therefore, malware tries to modify the code segment and execute data segment to achieve their ultimate goal. Third, most security systems have incorporated with limited methods to dynamically detect the behavior of malicious executable. Malware authors are taking the advantage of these three problems making malware more powerful.

Recent malware often designed with intention to compromise, possibly, many victim computers, stay long by hiding its footprints, and circumvent the secured system. Remote attackers can even control a large private network by compromising a secured server or workstation. This leads to tamper the kernel of the underlying operating system which would question the trustworthiness of the entire computing environment. Kernel integrity is more important to ensure a secure computing environment.

Therefore, a new security mechanism needs to be devised to strengthen the security of the kernel by avoiding the above said three issues which can permit malware to compromise the victim computer. The security of an operating system has long been subjected thirst research area. However, recent operating system design concentrates to offer better performance, more reliability, compatible and portable over security. This drift will never likely face the challenges of the predictable.

The gap between possible features and solutions offered by the current computing environments differs what is actually offered by them. However, it is more difficult to secure current computing environment without incorporating strong security measures. One possible solution is to introduce a mechanism to improve the security strength of the kernel to certain extent. In this thesis work, a new security enhancement mechanism namely, kernel level Process Authentication Mechanism (PAM) for detecting and preventing malicious code attacks has been proposed for Windows environment. By authenticating suspicious processes of the executable which are identified by the CoPDA algorithm at kernel level during run time, PAM can detect and prevent such attacks specifically before being serviced by the kernel.

The remaining of this chapter is continued by providing the background information regarding the problem under investigation. The motivation and challenges of the research work, objectives of this work and thesis problem statement are also devised and explained. Finally, the contribution out of thesis work and the organizations of the chapters in this work are presented in a nutshell.

1.1 Background

This chapter presents background information to make a clear understanding of what type of issues, PAM tries to resolve and what kind of different technologies PAM makes use of. This chapter also presents information about the layers of abstraction of a computer, the growth of current malware families, the most familiar targeted system resources and how they are compromised, the different techniques to combat malware attacks, and finally presents the background information about the execution environment for PAM.

All modern computer systems have a piece of most important system software, entitled Operating System (OS) or kernel which basically runs on top of the hardware that assigns the necessary system resources and supervises the execution of each application within the system. The OS as a whole consists of the kernel and may comprise other relevant programs for providing necessary services for each incoming request. More importantly, the kernel which acts as a part of the OS is responsible for many functions such as system calls, manage memory, and interrupts, exceptions, etc,.

One of the security mechanisms the system exercises is protection rings which are basically a construct of X86 processor architecture. These protection rings strictly provide definitions and boundaries for what type of operations they can able to execute. On most operating systems, there are four protection rings or privilege levels, which are numbered from most privileged (numbered zero) to 3 that represents least privileged mode. At any given time, for example, an X86 processor can run in a specific privilege mode, which decides what program code can and cannot permit. Basically the kernel of the OS runs in ring 0 and all user code can only run in ring 3. Figure 1.1 shows the various layers of abstraction a computer.

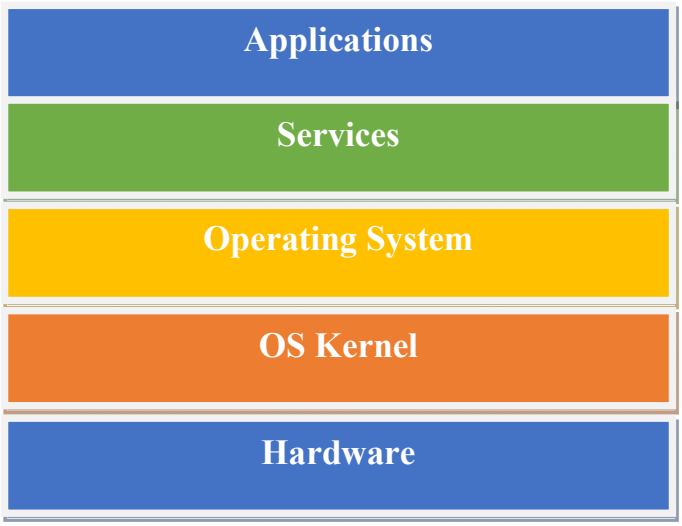

Figure 1.1 Important layers of abstraction

Since the Microsoft Windows is the most popular desktop OS, it becomes an attractive target for malware writers. The Windows Application Programming Interface (API) permits many different applications to utilize the powered features of the Windows family of OS. The Window API provides a uniform development environment, many communication and synchronization mechanisms, so that users can develop applications which are capable of running on all versions of the Windows family. The Windows64 API incorporated the features supported by the 64-bit versions of Windows to allow programming on the 64-bit Windows. The programming concept and API features are about the same as 32-bit Windows architecture. But, in WIN32 the size of the pointer is 32 bits whereas in WIN64, its size is 64 bits.

According to Microsoft, WOW64 (Windows 32-bit On Windows 64-bit) represents a subsystem which can run 32-bit applications on 64-bit versions of Windows. The limitation of a 64-bit copy of KERNELBASE.dll, kernel32, etc. a malware using its code in the 64-bit OS that facilitates the WOW64 environment. This indicates that the higher level APIs such as LoadLibrary(), VirtualProtect(), etc. are not available for direct access. For example, let us take a system call invocation under WOW64 on Windows 7 OS. The code depicted in Figure 1.2 shows that the NtProtectVirtualMemeory() cannot be accessed directly, instead a call to the function pointer within the thread environment block.

```
0:004:x86> uf ntdll32!ZwProtectVirtualMemory
ntdll32!ZwProtectVirtualMemory:
774e0038 b84d000000 mov eax,4Dh
774e003d 33c9 xor ecx,ecx
774e003f 8d542404 lea edx,[esp+4]
774e0043 64ff15c0000000 call dword ptr fs:[0C0h]
774e004a 83c404 add ESP, 4
774e004d c21400 ret 14h
```

Figure 1.2.Code to access NtProtectVirtualMemory function

A system call in computing is a technique in which a software application requests necessary services from the kernel of the operating system for completion. System calls are the only way to communicate with the kernel of the OS, and they can be accessed by programs through a high-level API rather than permitting direct access to it. A system call number is usually associated with each system call that can be used as unique identify. The computer OS preserves a system call handler table which is indexed according to the system call numbers and each entry in the table points the code to be executed. Figure 1.3 shows a system call invocation in x64 processor. The WOW64 consists of the following Dynamic Link Library (DLL) files to support Windows 32-bit programs: Wow64.dll which is responsible for marshalling all system calls while translating arguments using a system call table to ntdll.dll and wow64win.dll via wow64cpu.dll, and Wow64Win.dll which is acting as additional system call marshalling for console subsystems and windowing.

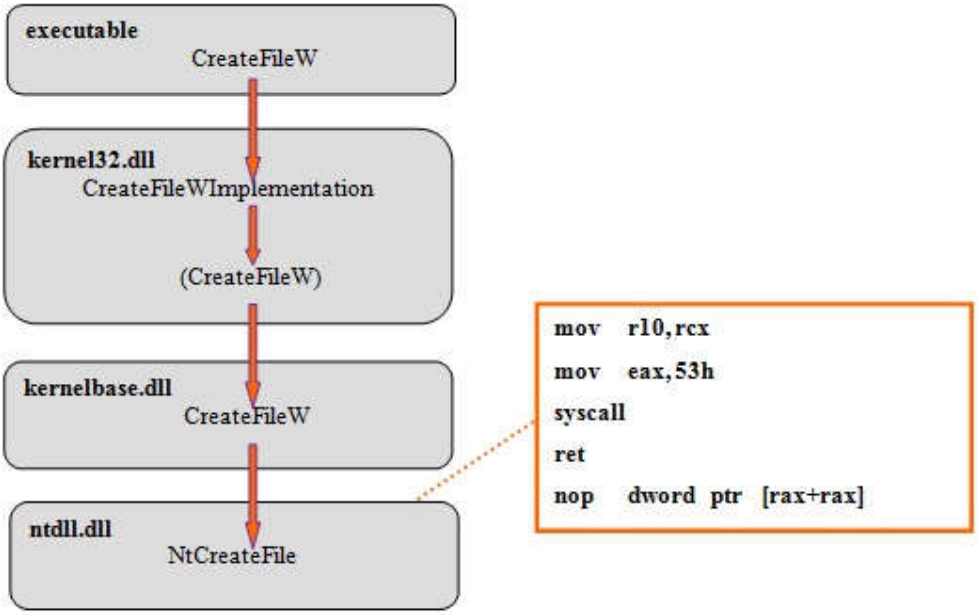

Figure 1.3 System call in x64

Few important functionalities of WOW63 are performing the mode-switches and dispatching system calls that can be either directly or through wow64!Wow64SystemServiceEx(). The system call invocation in WOW64 is slightly differs from accessing in X64 as shown in Figure 1.4.

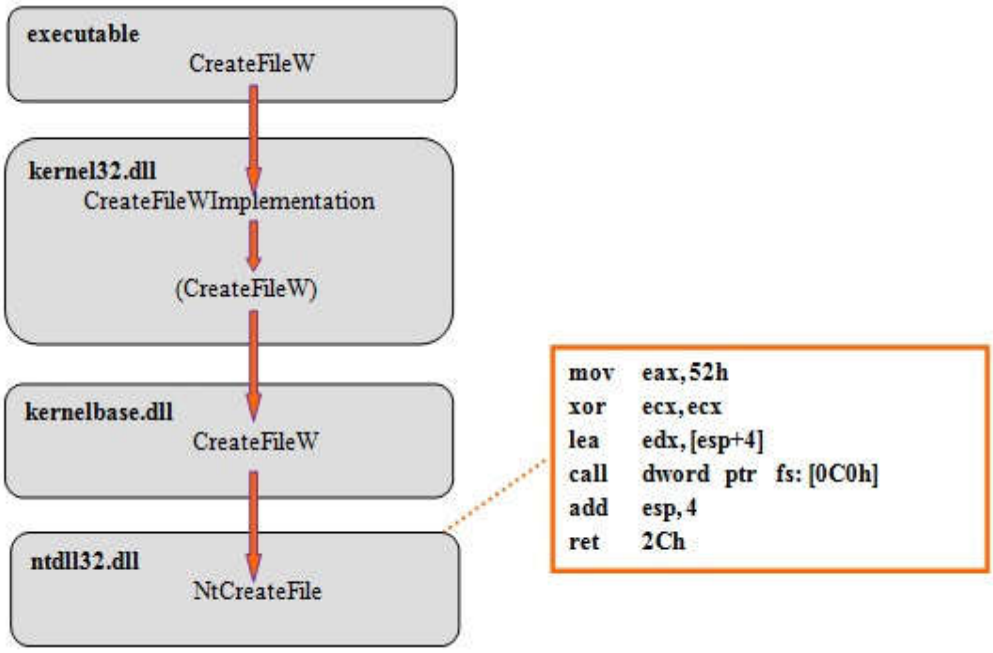

Figure 1.4 System call in WOW64

Today, stealthy malicious software is one of the most dangerous and challenging security threats. In reality, it is difficult to design a system which guarantees to be either secured or to remain secure over time. The kernel of the OS maintains many different data structures to provide services to either user programs or to run its own code. Protecting such important data structures and data against unauthorized access is a major issue for security designers or defenders. A new type of malware will be launched every day by modifying its predecessor, reaching 41 million new unique malware samples during second quarter of 2015 [1]. Malicious software can exploit hidden vulnerabilities found in the Internet computing environment without user's knowledge. Basically malware includes virus, worm, Trojan horse, and spyware that can be used to disrupt predefined computer security policies or operations, gain access to computer systems, gather sensitive data, or display unwanted advertising. It can appear in the form of scripts, active content, code, or other software. Today, more advanced malicious software is incorporated with rootkit techniques to make detection more difficult.

A rootkit is a technique designed with the intent of permitting a remote attacker to maintain highest privilege via backdoor over the resources of the victim computer. Different malware adopts the different masquerading technique to avoid its detection. As a result, rootkits can dynamically defy detection either by hiding from view or messing Antivirus (AV) software. Because of these characteristics, rootkits are potentially dangerous to the integrity of user data. In order to carry out illicit operations malicious rootkits make use of hooking mechanism which can able to modify the execution flow of a system call, but rootkits required to access kernel level APIs to accomplish their predefined tasks. The emergence of the first form of malware attack and hacking tool was followed by defensive techniques and automated tools. Today, many security software products integrate various defensive measures such as firewalls, AV software, spam blockers, etc. Figure 1.5 shows the different possible malware detection techniques, classifications of detection, and vital information available for malware analysis. The dynamic behavioral malware defense technique must intercept suspicious event and analyze them to detect their presence. However, deciding which event to intercept is very tedious task. In additions, information about an event alone is not adequate to discover whether a process of an application is malicious or legitimate. Though many researchers actively involved in this area, they fail to specify perfect definition to it [2-4]. To overcome this issue, researchers concentrate on two areas to reveal malicious activities.

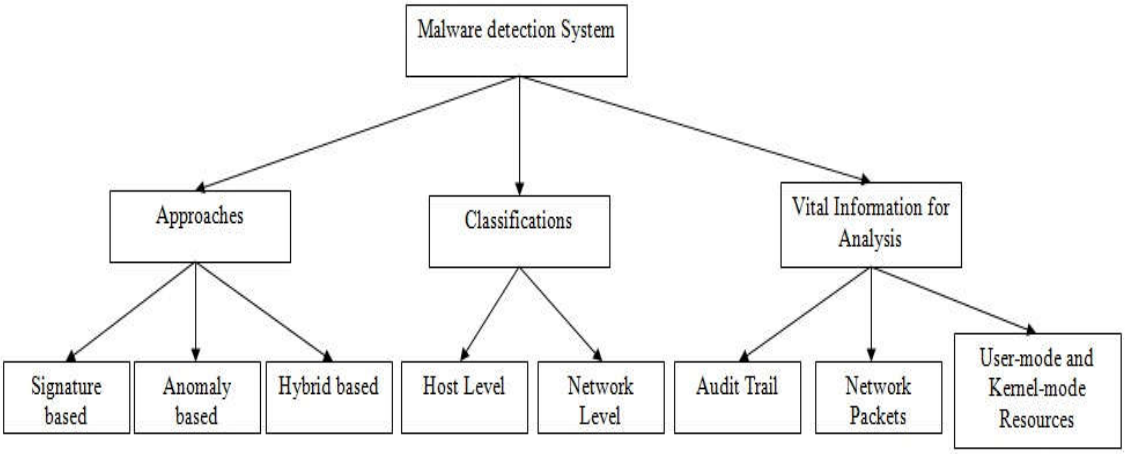

Figure 1.5 Overview of malware detection system

First, revealing malicious actions through determining the association between memory, files, processes, and other system resources [5-6]. In [6], the author proposed one such technique to identify a malware by discovering the concealed link between processes using data tainting method. The second approach is the movement towards security policy enforcement mechanism which elucidates malicious operations by controlling access privileges of various system resources. Any event which violates the predefined security policies is concluded as malicious.

Security Policy Enforcement Mechanism

There exist two useful common malware defense techniques to protect system resources against malicious code attacks. They are System Call Monitoring (SCM) technique and mandatory Access Control List (ACL) mechanism. The former technique is widely used to detect compromised applications and analyze them to minimize the harm that they can cause [7-13]. SCM technique relies on setting up policies that impersonate as a legitimate application system call and then suspending or terminating execution if the application pretended to be legitimate. Though SCM technique alone cannot completely protect an end-system, hence it can be used as an additional technique to strengthen the detection capabilities of Intrusion Detection System (IDS). The ACL technique relies on implementing a Message Authentication Code (MAC) mechanism that requires a malware's signature to define Security Policy Specification (SPS) to enforce access restriction to various system objects.

Existing MAC security solutions such as APPArmor [14] and grsecurity [15] that rely on an authorization mechanism allows a user to enforce strong security policies against malicious executables. These MAC solutions are implemented in Linux open OS to supervise access rights to different system resources by applying security policy specification. An online anomaly based detection technique [16] proposed to identify a suspected malicious execution path of an application which relies on measuring similarities between execution paths. However, few malicious executable might behave like legitimate applications which are very hard to distinguish. Therefore protecting individual computers or workstations in a network is a very important.

The security policy enforcement mechanism elucidates malicious operations by controlling access privileges of various system resources. Any event which violates the predefined security policies is concluded as malicious. Many related works have been found in association with implementing security policy enforcement for modern computing environments [6] [17-19]. A kernel-mode protection framework namely, WHIPS has been developed for Windows environment by Battistoni et al., [20]. WHIPS inspects every system call request in the kernel mode and validates the caller's service it requests such as process name and parameters of the request using a access control database and blocked a request that is invalid. The challenge in using WHIPS for Windows environment is to exactly defining the access control database specifically deciding the safeness of parameters.

KVMSec [21] focused on periodically checking the integrity of several system objects by maintaining secure communication channel with guest environment. However, the system runs with partial completion of integrity checking component. The system proposed by Payne et al. [18] monitors system-wide process manipulation activities to hook important system calls and maintains integrity over such hooks to detect malicious activites. In addition, malwares that do not adopt system call hook technique can easily bypass. To detect the presence of rootkits, XenKIMONO [22] used different techniques such as cross-view check, integrity check, etc. However, designing a system to detect all kinds of rootkit types is really a difficult task. Another approach [23] that is relying on policy enforcement is, system call monitoring. This approach is based on intercepting system call invocations and directs the response to the actual system call. The system mainly relying on process granularity level and can directly able to discover and prevent malicious system call operations. However, this approach suffered from three issues.

First, a malicious code which pretends to be legitimate can evade its detection. Second, determining which system calls to be controlled is really a difficult task. Third, this technique suffered from high false positives. If both malicious executable and anti-malware detection software run at the same privilege level, then the policy based technique to detect and prevent malware activities become useless defense technique. This is one of the main drawbacks of policy enforcement mechanism.

Microsoft's new security update

The elementary problem of designing security features to guarantees a secure computing environment is that any process running in the kernel mode obtains highest privilege can increase its privilege level to access and control over other system resources. This also permits the malicious code to rewrite any code segment in memory to run its own code and access data part of other processes illegally. Therefore, preventing illegal access to code part, kernel object manipulation, and data execution can raise the security level substantially. In order to improve the security strength of the operating system, Microsoft introduced new additional security features such as Kernel Patch Protection (KPP), Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) from Windows Vista onwards to ensure the trustworthiness of the underlying computing environment. ASLR is an important technique which runs to arbitrarily check the address range of each process to hide the target location from the attackers. KPP averts malicious executable from patching vital data structure and code part by periodically validating if any protected kernel memory part is modified. DEP permits marking a certain part of primary memory for data keeping sensitive information and prevents the preset data area being executed.

Another useful security enhancement solution to ensure the trustworthiness of the underlying computing environment is hardware level implementation of Trusted Platform Module (TPM) [24]. TPM is implemented to offer secure information protection by integrating cryptographic mechanism in hardware. TPM is one of the popular security solution techniques which can guarantee that the system resources are not altered and limit access to data. Although TPM is a strong hardware level security solution against tampering resistance attacks, the hardware level implementation might be flexible to modify the security of the system when needed. There were few hooking techniques have been proposed which can bypass these security measures [25-27].

Malware can still run with the same level as the security system and may thrive in challenging entire security system. A virtualization mechanism permits the execution of guest code directly on the physical machine while the executions of traditional virtual machine environment like QEMU [28] in user-mode. QEMU is another useful open source virtualization technique which offers benefits such as Input / Output emulation and initialization of virtual platform.

Kernel Virtual Machine (KVM) behaves similar to device driver and service user request using Input Output Control (IOCTL). KVM uses a virtual memory technique called the Shadow Page Table (SPT) [29] which provides a huge memory address space to each process than the host's real memory. The SPT is mainly used for mapping the guest code virtual memory onto the system primary memory. However, the introduction of SPT technique involves lavish context switching between host and guest. There are many barriers to policy enforcement and acceptance. First, if both malicious executable and anti-malware detection software runs at the same privilege mode then the policy enforcement technique to detect and prevent malware become useless defense technique. Second, recent malwares with advanced hook techniques can bypass the predefined security policies. Typical OS's kernel often fails to include either stronger restrictions on the program executable or protecting system services against malicious code attacks.

Some effective real-time anti-rootkit solutions also exist to dynamically analyze and discover hidden rootkit malware. However, such tools failed to discover kernel level API hooks dynamically. Therefore designing a mechanism which is capable of detecting and preventing malicious code attacks to protect both user-mode and kernel-mode is a challenging problem. Although there were many different approaches have been proposed in the past to prevent system service against malicious code attacks, the following issues were not addressed significantly in the existing approaches.

- Resist hidden processes attacks and existential forgery attacks and code injection attacks.
- Strong kernel level authentication to protect system services and important data structures.
- Minimizing the false Positives and runtime overhead for the entire system.

Some existing research works have addressed the problem of detecting the hidden footprints of malicious executables using crosscheck based approach. These algorithms only list the properties of the hidden processes. Microsoft Windows treats the kernel of the operating system as a black box; hence makes the complexity of designing a monitoring framework becomes a challenging task. Therefore, there is a growing interest for the kernel level authentication techniques to protect system services against malicious code attacks which enable designers to test and validate suspicious processes either even before causing damage to the system resources or compromise the entire system. Malicious code is one of the security threats today on the Internet. Attackers can inject malicious code into the software by exploiting a hidden bug that may exist in it and execute the injected code abnormally. During code injection phase, attackers need to execute privileged events, by calling a system service, more importantly, native APIs. Without distinction, the kernel provides services to both malicious processes of an application and also to legitimate process. Though many vendors, import different security features into their product like providing security functionality to completely protect a system, but it may not be adequate to trust the system.

1.2 Motivation and Challenges

Due to the decisive responsibility of the OS in managing large number operations in a computer system, the lack of security of an OS will greatly impact the overall security of the computing environment, as well as the security assurance of all programs running within the environment. If the underneath OS can be compromised, it will certainly lead to full information compromise of a secure computing system. Inadequate access control and management of execution of individual application processes in an OS can lead to break the entire system security policies.

Analysis of malicious code attack risk enable defenders to model attack reasoning and scenarios about the relationship between dependencies between attack paths. By generating comprehensive models about different attack scenarios, it is possible to design a specific quantitative measurement technique for attack risks coupled with a network settings. Then the outcome can be later used to improve the security configuration of the network. However, such a quantitative attack measurement model technique considers several technical design challenges.

First, the outcome of a quantitative measurement model must have clear semantics which could permit for the development of deterministic algorithm for generating the expected results. Second, the quantitative analysis of different security risks must be able to generate useful conclusions even in the absence of sensitive sampling security data. Finally, very large networks are usually highly dynamic in nature.

Generating attack graphs are normally a well-known method which can offer the expected information about attack scenarios and its dependencies of a malicious executable. A malicious code attack graph G= (V, E) is a dependency graph in which E represents the relationship between and V represents a state transition. Although there were many quantitative assessment models exist to depict attack scenarios of a specific network, this kind of static analysis suffers from several limitations to handle current state of art on security.

- An attack graph model offers only a partial interpretation about attack scenarios of a network. The existing methods lack of hard theoretical foundations.
- Existing design approaches to quantify network attack graphs fail to consider the dynamic nature of a networked environment.
- For larger networks, quantitative analysis of attack graphs fall under NP-complete problem which is always non-trivial.

Dynamic analysis based defense mechanisms have been developed to overcome these issues. Such approach works by utilizing the execution flow of legitimate applications to discover the presence of a malware. However, attacks such as mimicry and shadow attack weaken the dynamic malware analyser. Most OS kernels often enforce only a limited access restriction on the application program permitted to carry out its execution. As a result, malicious software program which runs as a stand-alone process abuse system resources for its execution. Once installed on the victim computer, malicious executable can freely run to execute privileges associated with the current user account running the process. This may lead to affect the entire network. Therefore, securing both user-mode and kernel-mode of an end-system is very important. Dynamic malware analysis based user-mode malware detection techniques and anti-malware detection tools have been studied and developed. As advanced malware incorporate rootkit techniques to evade detection, different algorithms have also been developed to optimize the malware detection system. But malware attacks that target kernel level compromise is an issue.

Therefore, to protect both user-mode and kernel-mode of the OS, techniques such as access control, authorization mechanism, and authentication mechanism have been implemented. One of the key foundations of most modern operating systems involves employing an appropriate level of access control protection through Discretionary Access Control (DAC). In DAC, the owner of an object state which subject(s) can access the object. Although this kind protection measures improve security in a time sharing or multi-tasking environment to certain extent, it suffers to impose firm security policies for individual application executable. Certainty secure applications always demand secure OS, and preventing application compromises at the kernel level by enforcing strict access control policies are generally considered more attractive and an effective approach. In computer security, the ACL technique improves the security level of the OS to certain level by enabling either the system or system administrator to specify necessary security access rights to resources objects in a file system.

The ACL mechanism offers a strong separation of application programs that builds secure execution of trustworthy applications from un-trusted applications. Therefore, it guarantees security for applications by defending and bypassing against the tampering with secured applications. Most malicious executables attempt to run with the same privilege level as a user-program or system and tries to increase its privileges level to attain its designated goal. Furthermore, the access control policy based mechanisms maintained by the OSs are so rudimentary. This permits nearly all privileged applications and system services running with root privileges than the program what actually needs. Thus, exploitation in any of these running programs can lead to complete system compromise. ACLs is appropriate on supervising disclosure of information through embedding strict security levels to different system objects and subjects, thus limiting access controls into the system. In opposition DAC concentrates on fine-grained access control of system objects through different object level permission modes and Access Control Matrix (ACM). Limitations in each of this mechanism can fail to satisfy one or more of the following three characteristics of an ultimate security mechanism:

(i) Certain incapability to cessation of predefined security policies by evading access controls, policies by the mechanism

(ii) Obscure continuous privileged interaction with the mechanism

(iii) Implementing cost-effective and real-time mechanism

The above discussed security solutions fall under the category of authorization mechanism. However, authorization mechanisms are not alone sufficient to guarantee a secure system. Several methods such as public-key cryptography or password can be found in a networked environment or multi-user system for user authentication. In a multi-user OS, depends on managing a huge amount of diverse applications, providing authentication to basic operating data is turning out to be more important. With modern stealthy malware attack technique, system information which is trusted for granting access needs to be rechecked and reassessed.

An authentication mechanism that used data provenance proposed in [30] pointed out the importance of verifying the kernel level authenticity and originality of data flows which are consumed by the system. Dai et al. [31] described a digital signing method for ensuring integrity and authentication of a signing agent on a system for generating digital signatures. It also described how a program signer differs from a human signer and also listed the system challenges that are associated with trustworthy of the program signer. With stealthy malicious code attack techniques, human's timely reaction to intrusion detection is not possible. Also, Microsoft Windows is the most popular and widely used OS, attacking a considerable number of systems in local area network by compromising a single system is possible. Therefore, the research work is carried out with the following objectives.

1.3 Objectives of the Research work

The objectives of the research are as follows:

(i) To provide a solution to the graph-based static malware detection approach, by devising new graph-generation and graph-matching algorithm to generate better detection rate than existing graph-based malware detection approaches with lesser false positives.

(ii) To develop new dynamic malware detection approach for protecting the user-mode of an end-system to provide better results than the existing user-mode malware detection techniques.

(iii) To devise a new cross-check based algorithm for detecting hidden footprints of a malicious executable to optimize user-mode malware detection approach to provide better detection rate than existing hidden process detection algorithms.

(iv) To develop a new mechanism for authenticating unauthorized processes of an executable during runtime before being serviced by the kernel and thus ensuring system assurance.

1.4 Thesis Problem Statement

Based on the objectives, this research work is focused on designing and developing a security enhanced mechanism named, PAM, by detecting and preventing malicious code attacks against system services by validating the originality of the suspicious processes of an executable application during runtime.

(i) Static malware detection techniques can effectively detect and prevent known malware attacks. Therefore, a graph-base malware detection approach has been proposed to provide better results than existing solutions. However, static malware detection systems are not effective against unknown malware attacks. In addition graph-based approaches belong to NP-complete in nature.

(ii) In order to detect and prevent both known and unknown malware threats that target API hook attacks, a user-mode malware prevention system has been proposed. In order to initialize and carry out illicit operations on a victim computer, a malicious executable can create multiple duplicate processes. Therefore malwares that incorporate rootkit techniques pose a serious challenge to the defenders.

(iii) A new novel cross-check based algorithm is also designed and developed to discover hidden processes and services. Whenever a new process is created, first it will be validated by this algorithm to check whether it is suspicious or not. The implementation of the proposed algorithm and its outcome can greatly helps to reduce the overall performance overhead of the proposed mechanism.

(iv) The process authentication technique is extended and incorporated with the kernel on determining whether each incoming service request of a process is legitimate or not. If suspicious then it will be authenticated by the kernel of the OS during run time. Therefore, improvement in authenticating of the processes of an executable application is incorporated and it is directly improving the trustworthiness of kernel level information.

1.5 Summary of the Research Contributions

In this thesis work new static malware detection approach and user-mode malware detection approach have been developed. Also, list its advantages and security barriers associated with them which fail to detect hidden footprints of unauthorized processes a malicious executable. Therefore, a novel hidden process and service discovery algorithm for optimizing user-mode malware detection has also been developed. Furthermore, the process authentication mechanism is extended to guard against kernel level unauthorized processes malicious code of an executable at runtime on Windows. This thesis work enforces a mandatory authentication on all suspicious processes of an executable whereas legitimate processes are directly being serviced by the kernel.

(i) The graph based malware detection approach namely, GraMD has been designed and implemented to detect malware hook attacks. To model an API function call as a graph, a new algorithm namely, API Call graph Algorithm (ACA) has been developed. In addition, a modified graph edit distance algorithm namely, Graph Matching Algorithm (GMA) to compare two given graphs has also been proposed. Real-time malware samples were collected from reputed websites and used for evaluating the proposed GraMD. The collected are classified into three sets, namely, rootkit, worms, and viruses for evaluation purpose and the experiments are conducted for each set separately. GraMD approach is compared with the existing approaches for the detection of malware attacks and the experimental results show that GraMD consistently outperforms the existing techniques with an average of 98.84 % detection rate and 0% false positives. However, such statistical approach can effectively deal known malware attacks but fail to prevent stealthy unknown attacks. This leads to the development of dynamic based malware detection approach

(ii) Detecting and preventing malicious code hook attacks in user-mode namely, User-mode Malware Detection (UMDetect) has been developed. Unlike other malware hook detection techniques, the UMDetect involved dynamically analyzing the behavior of Windows native API hook attacks. A new DLL Classification Algorithm (DCA) has been proposed to validate whether the DLL files to be imported/exported is malicious or not. For conducting experiments, a dataset is obtained from public resources and collaborative researchers. The evaluation results show that the proposed UMDetect has successfully discovered all malicious hooks and achieved 95-100% detection rate with 0% false positives against the existing user-mode malicious code detection approaches. Although the implemented technique detect and prevent malicious code attacks against important user-mode data structures, it cannot deal hidden entries of a malicious executable. Hence a solution needs to be devised to discover the hidden footprints of such malwares.

(iii) A cross-check based algorithm namely, Concealed Processes and services Discovery Algorithm (CoPDA) has been developed to discover all suspicious processes and services of a malicious executable. A dataset consists of 100 rootkit malware samples and 50 benign programs are collected from public resources. Experimental results show that, the proposed CoPDA algorithm detected all hidden processes and services of a malicious executable effectively and surpasses some real world anti-rootkit detection tools and existing solutions with 99-100% detection rate with 1.82% false positives.

(iv) Finally, a novel idea to enforce kernel level process authentication namely, PAM has been proposed for protecting system services against malicious code attacks in Windows. PAM provide strong security against malicious code attacks by combing user-mode information through the implemented CoPDA algorithm which is responsible to discover all suspicious entries of an executable and kernel-mode information by authenticating its originality during runtime. As PAM imposed mandatory authentication to confirm the originality of all suspicious processes of the executable, it guarantees prevention of malicious processes which fail to succeed during the authentication phase, thwart its subsequent attacks.

The effectiveness of the PAM is compared with the existing approaches and widely used anti-rootkit detection tools. With the advantages of improving kernel-security and generic overhead impact of 2%, PAM can become a practical solution in the current environment to prevent malicious code attacks.

1.6 Organization of the Thesis

The present chapter explores the background information for converse of the research dissertation. The motivation and challenges, objectives and the thesis problem statement of the research work are also devised and explained.

Chapter 2 presents the survey on recent works related to the work presented in this thesis. The dissertation also analyzed different works to detect and classify malware attacks by using behavioral analysis methods to address and avoid the limitations of traditional defenses. A detailed discussion and comparison of different antivirus techniques can also presented. The challenging research issues in the current research are pointed out. Finally, the extract of the literature survey and the summary of the survey are also discussed.

Chapter 3 describes the proposed graph-based static approach for the detection of malicious code attacks, GraMD. Followed by the description of the algorithms and performance comparison of the ACA and GMA algorithms with existing algorithms are reported. We designed and implemented GraMD and its simulation results show better detection rate than existing quantitative malware analysis methods. Although this technique offers few advantages, its demerits in dealing malicious code attacks force us for the development of preventing such attacks rather than detecting them.

Chapter 4 presents the proposed user-mode prevention of malicious code attacks against unauthorized process attacks, UMDetect. Subsequently, the operation of the DCA algorithm is described. The experimental results with an objective to detect API hook attacks that target user-mode data structures and its comparison with existing techniques and tools are also presented. However such prevention focused on monitoring only limited

user-mode data structures. As a result, kernel level attacks can easily bypass and abuse system resources. This clearly pointed out the requirement of kernel level runtime verification mechanism.

Chapter 5 presents the proposed CoPDA algorithm developed for ascertaining suspicious entries of a malicious executable application. The performance comparison of CoPDA algorithm with the existing algorithms is presented. The experimental results are compared with the existing algorithms and reported.

Chapter 6 describes the proposed PAM, general kernel level process authentication mechanism for general Windows based desktop computers. An introduction about the experimental result analysis of the proposed mechanism and existing mechanisms based preventing unauthorized processes attacks of an executable application are discussed. The simulation results of PAM obtained by various experiments are also illustrated with the aid of different graphs. The overall performance comparison of PAM and the existing techniques for the detection of unauthorized malicious processes during runtime with the help of graphs are also illustrated.

Chapter 7 concludes the research work by summarizing and highlighting the findings that are facilitated to accomplish the objectives. The limitations of the research work have been identified and presented to carry out the possible future work to improve further.

CHAPTER 2

LITERATURE SURVEY

This chapter briefly presents an overview of different types of malwares, common classes of malicious functions, its trend, two important classes of malwares that target Windows platform, and discuss some of the prior work that deals with quantitative analysis of malware attacks using graph based approach. Additionally, the prior work on securing an end-system in each category such as static analysis, intrusion detection techniques, policy enforcement mechanism and system call analysis are pointed out. In addition, the advantages and limitations of each technique with the requirement of protecting a workstation using user-mode and kernel-mode information are also presented. This chapter also presents the challenging issues in the current research which is clearly discussed.

2.1 Preamble

Both computer security and information security became more important since the introduction of Morris worm which was the first malware released in 1988 and shut down 10% of the computers on the Internet [32]. Since then many organizations have designed and implemented the information security to protect their valuable data. Security defenders are under increasing demands to not only defend devastating data breaches but to also prevent against attackers who are using advanced techniques to conceal their attacks. As stealthy malicious malwares become increasingly advanced, it is become more vital than ever to improve defensive techniques and methods constantly. Basically the term malicious code includes viruses, worms, Trojan horses, spyware and botnets that can be used to gather information about a computer user and access to a system without their permission. These malware can appear such as scripts, active content, code, or other software. Two general classes of malware programs are: first class of malwares needs a host program (viruses, Trojan horses, logic bombs, trapdoors) and second class of malwares are independent programs (worms, zombie). Malwares are classified based on their characteristics; some malwares do not replicate (activated by trigger) and others that producing copies of themselves.

2.2 Malware

There are different classes of malware or malcode that have varying methods of infecting computers and propagating themselves. The damage being caused by malcode into computers varies from fairly innocuous to stealing sensitive information, destroying information, and compromising and/or completely disabling computers and networks. Traditionally, malwares are categorized into different classes based on their function, authorship, and delivery mechanisms. The following list some important malware types:

(i) Viruses and Worms

A computer virus is a type of individual self-replicating software malcode that must have the ability to propagate on its own by inserting into other applications on an infected computer, leaving infections as it travels from one computer to another. Almost all viruses malcode may exist on a system as part of an executable program, but they will not perform its malicious operations or able to propagate until a user executes the host application. Not all viruses are malicious – few of them are written to help discover vulnerabilities that may exist on a computer. Computer worms have similar behavior as virus malcodes in that they are self propagate and can cause similar kind of damage to the victim computer. Compared to viruses, worms are standalone file which do not require any assistance for propagation. A worm can either exploit vulnerabilities on the victim computer or use certain type of social engineering tricks for execution.

(ii) Exploits

An exploit is a methodology, a command, or a software malcode that can be used wither to demonstrate to attack a security vulnerability that may exist on a computer or to attack a particular vulnerability. However, exploits are become a common component of malcode and make use of software vulnerabilities to permit privileged execution of malicious executable.

(iii) Downloader and Droppers

Downloaders are usually allowed downloading additional malwares from a remote server, while droppers embedded with malwares. However, both install additional malware on the compromised system.

(iv) Backdoor

In the current network environment, it represents the undocumented method of accessing a computer surreptitiously bypassing the legal predefined authentication mechanisms. Recently, almost all attackers make use of back doors to gain and maintain complete administrative access to a computer after it has been successfully compromised and permit clandestine remote access over them.

(v) Ransomeware

Ransomeware prevents service to authorized users by restricting or disabling normal functions, or hiding data. They are typically used to harvest money from tainted computer users.

(vi) Bots

Botnet has become the most serious security threat on the current internet infrastructure. A botnet (BotNetwork) is an interconnected collection of compromised infected computers (bots) which is remotely controlled by its originator (called botmaster or botherder) under a common and control infrastructure. Bot is a new type of malware which is designed for malicious activity. After the bot code has been installed into a computer, the computer becomes a member of the bot network. Here all the bots are under the controlled of BotMaster. So if bot exist in computer, it is not harmful until it receives command from BotMaster. After receiving the command from BotMaster, it is dangerous for system. These bots are not self-propagate from one system/network to other system/network. A botnet enrolls its soldiers using social engineering techniques or by exploiting software vulnerabilities.

(vii) Trojans

A Trojan is a type of harmful computer software that are defined to look like legitimate or useful program, but contain hidden code that can perform a variety of malicious operations on a computer. It may trick users into loading and executing.

(viii) Rootkits

A rootkit is a technique which is designed with the intent of allowing the remote attacker to maintain highest privilege over the resources in the victim operating system.

2.2.1 Malware Motivations

Except kiddies, advanced malware writers designed their code for performing many illegal activities such as monetary gain, ideologies, and politics. One such malware family is, advanced persistent threats which were written against politics and ideologies [33]. Advanced persistent threats such as Stuxnet used different delivery techniques to gain sufficient access privileges over the victim computer. Malicious malwares are intentionally designed to harvest money illegally. For example, Zeus bot is sold in the black market as crimeware kit which is used for producing customized malware variants. A Botnet is an interconnected collection of compromised computers under remotely controlled by BotMaster. A Botnet can be used for massive Distributed-Denial-of-Service (DDoS) attacks, installing key-logging that can steal victim's password and data, and compromising computers to prepare them for infection by future attacks. Certain malwares are purposefully designed to steal sensitive information such as financial information and user credentials which can later sold in the black market. Ransomeware averts service to legitimate users by hampering or stopping customary services, or hiding data. This type of malwares is used to harvest money forcefully from tainted computer users [34].

2.2.2 Malware Deliverance Mechanism

Malwares are designed with the intention of infecting hosts by exploitation of unknown vulnerabilities, social engineering, and negligent security practices. Security vulnerabilities permit malware authors to freely run their code with necessary privileges. These comprise include code injection, input validation, privilege escalation, cross-scripting vulnerabilities, and input validation. During the release a software, designers unintentionally left some of the uncertain vulnerabilities and patches, the malware authors make use of these openings before they are known. For example, zero-day vulnerabilities are mainly troublesome for many software, because they allow malware writers to profitably taint several susceptible hosts. In addition, malware utilizing zero-day vulnerabilities can compromise many hosts freely until such vulnerabilities are ascertained and patched.

Malware usually targeting hosts with slipshod security policies seek computers with weak passwords. These include scanning an entire network for discovering hosts which are running common network services. Once succeed the malware cracks the discovered network services by attempting dictionary attacks. Another tricky technique which support remote attacker in exploiting a host is by convincing an end-user through social engineering. For example, Trojan horses may trick end-user to download and execute malware. One such malware family is Zeus or Zbot which has been sent to targeted email crusades, in turn sent to targeted victim computers as electronic greeting card from shipping invoice or white house. Of these three delivery mechanisms, the use of exploiting security vulnerabilities is very rarely used. This is because, the effort required to exploit such security breaches. Rather than discovering security vulnerabilities, remote attackers typically buy them on the black market [33]. However, exploits are only possible as when the targeted vulnerabilities remain open.

2.2.3 Malware Trend with Hook Techniques

More than 50 million of new malware counts were discovered in the fourth quarter of 2014. Typically, an average of six new malware samples discovered every second. By the end of 2015 [35], the McAfee Labs project has collected more than 500 million malware samples. Malwares primarily infect computers through social engineering via exploitation of lack of security policies. Malware authors make use of such security vulnerabilities to enable privileged execution. Today, most recent malwares are designed with advanced techniques depending on the operations which they try to execute. To evade detection, most malwares often embed some of the operations at the time of executing their code. First, they inject their malicious code into legitimate processes of an application to initialize and carry out its illicit operations. Second, they disable all currently running security software applications to evade its detection. Thirdly, they can hide its existence by accessing hidden file features illegitimately. There are many forms of malicious software that can constantly affect a user's computer. Today, more advanced malicious software is incorporated with rootkit techniques to make detection more difficult. It has been in the wild for more than 15 years [26]. Different malware adopts the different masquerading method to avoid its detection [36]. As a result, rootkits can dynamically defy detection either by hiding from view or messing AV software. Because of these characteristics, rootkits are potentially dangerous to the integrity of user data.

Rootkits can be used for either legitimate purpose, such as debugging or malicious purpose when combined with malicious software. In order to execute different pre-coded tasks, malicious software needs to perform some initial operations such as enumerating processes and services, opening a port, or establishing a network connection on the victim computer. A malicious rootkit can use either user-space API hooking or kernel-space API hooks in order to remain hidden. Hooking is a set of code which alters the normal behavior of the operating system by intercepting the system API functions or information exchange passed between different system resources. Hooking can be used for either legal purpose, such as debugging and extending functionality or to host many illegal activities with the use of rootkit technique. Hooking can be used by malicious code such as rootkits, which try to hide themselves. As mentioned earlier, rootkits use different types of hooking techniques in order to remain hidden.

In order to bypass malware preventive measures such as AV software, advanced malware writers create a new type of malware by encrypting, signing, reordering, padding, compression, or otherwise simply changing its code without modifying its functionalities. Such changes can be applied easily by using software tools such as packers and encrypters. Few malware variant modify them automatically at the time of propagation. For example, metamorphic viruses modify their code structures when they disseminate [37]. The computing environment continues to evolve both in complexity and size as illustrated by emergence trends such as ultra large scale systems. This certain increase in complexity might introduce new and unknown security vulnerabilities which also increases the complexity of its detection. All these points motivate the defenders community to find a solution which is capable of discovering all malwares that avoid traditional defense systems.

2.2.4 User Mode Malware

User-mode rootkits work in Ring 3 mode, which infects the operating system outside the kernel level. They replace drivers, dynamic linked-library files and various processes with their own versions, which don't show the rootkits' presence. They also intercept system calls between the kernel and software programs, making sure the forwarded information doesn't include any evidence of the rootkits. User-mode malicious rootkit can able to hook user-mode applications, data structures and system library files through API functions to evade its footprints.

2.2.5 Kernel Mode Malware

Unlike user-mode malware, kernel-mode malware tries to manipulate either kernel level APIs or other system resources. Sometimes an attacker may inject malicious code into kernel and misuse control data structures and non-control data structures to obtain appropriate access rights over the system objects. Furthermore, a remote attacker may target vulnerable software to tricky end-user to download and execute malicious code into the victim computer. Such attacks might use rootkit technique to compromise a single operation on the victim computer, thus it becomes sufficient to compromise the entire system. Therefore by monitoring or inspecting the behavior of each API function call, it is possible to discover bot like malicious software. An attacker usually look for single vulnerable point on the victim computer but the defender needs to monitor thousands of both user-mode and kernel-mode resources.

2.3 Analysis of Malware Detection and Prevention Techniques

Traditional defense systems against malware attacks include the adaptation of security related automated tools, the design of trusted computing environment, and the awareness of secure computing practices. Making cognizance about safe computing practices to computer users is essentially important in avoiding intrusions, offering less chance to phishing attempts, and preventing other intrusive attempts.

Techniques such as trusted computing ensures protecting sensitive data through signing code which permits users to verify that whether it came from trusted party. Different anti-malware software is designed uniquely with specific detection time to be spent for each stage in the malware life cycle. Each anti-malware technique has its own advantages and challenges. Figure 2.1 shows various techniques to be applied during the life cycle of a malware to detect malware attacks, it limitations and challenges. Malware defense techniques can be classified into three types such as static malware analysis, dynamic malware analysis techniques and automatic malware analysis.

	Intrusion	Initialization Execution	Illicit Activities
Techniques	Signature based Detection	Behavioral Monitoring	State Monitoring
Challenges	Obfuscation and encryption	Human Assistance	Subverts
Limitations	Undetected Unknown malware	High False Positives	Circumventable

Figure 2.1 Malware Life cycle versus malware Detection Techniques

2.3.1 Static Malware analysis

Static malware analysis technique is mainly focused on understanding the internal structure of its executable part or analyzing it to discover its functionalities without executing them. Methods such as AV scanning, string analysis, identification of scripts, reverse compilation, and hashing falls under this category. The AV software is no longer a match for today's threats. Because it alone does not provide complete protection and does not offer enough protection. It cannot be designed to detect, defend and remove all kind of malware attacks at any given time. As a result, current anti-malware defense software is not completely sufficient.

Though Antivirus software can detect only known malware types, one of its noted advantages is that they generate low false positives. However, such signature-based malware detection software's are suffered from several weaknesses. First, AV software is inherently reactive i.e., it finds malware attacks only after a computer has been infected. It also requires signatures of unknown malware to be analyzed and discovered prior to detection. Second, AV software fails to locate a malware variant which has been modified from its predecessor. Thirdly, such security software uses heuristic techniques to discover as many common set of malware variants as possible. Recent AV software designers focused on reducing the required number of signatures of malware variants and reforming its analysis [38] but not against detecting, preventing, and removing zero-day attacks.

2.3.1.1 Network Level Malware Analysis

The number of security vulnerabilities that target the Internet and computer networks is increasing more and more over time. An IDS is a security application like access control mechanism, antivirus software, and/or firewalls, which is developed to prevent communication system and information against unusual pattern types.

These scarce are commonly referred to as peculiarities, exceptions, outliers or anomalies in different application domains. The main reason for launching such patterns by outside attackers is to purposefully disrupt the computer network, unauthorized access to the network and/or steal sensitive information. Several IDS research has been developed in the past to improve the detection precision rate and detection stability. As an intruder's exploit is conspicuously vary from the predefined system security policies, they can be detected. There are many advantages of making intrusion detection as a component of the all-inclusive defense system. However, various traditional computer system and its associated applications were designed to situate in a location where security was not a major concern. Therefore, such system and applications are targeted by malware writer when mounted in the modern network scenario. Additionally, security design flaws or bugs in computer system and application become targets by a malware developer to attack them. As a result, many existing preventive solutions may not be work well as expected.

A Host based IDS (HIDS) can monitor a computer for discovering intrusive activities, whereas a NIDS checks network related activities or events such as IP addresses, network packet traffic, network protocol exercised in those packet, service ports, etc. Depending on the type of acquired information to be analyzed, an IDS can be further classified into either signature- based, behavior-based or hybrid-based. The characteristics of various IDSs are listed in Table 2.1.

(i) Anomaly based NIDS Techniques

Network intrusion detection technique has been in the wild for more than few decades. In a real time scenario, an intranet is connected to the outside world through the Internet. Therefore, by installing the Network IDS (NIDS) in a suitable place, it can read all network traffics to find out suspicious packets. Though signature-based NIDS are good at detecting well-known vulnerabilities, they fail to detect new, unknown vulnerabilities, even if they included minimal variants of its predecessor.

Table 2.1 Characteristics of various IDSs

Type of IDS	Characteristics
Anomaly Based	(i) Assumes that all intrusive events are inevitably anomalous. (ii) It builds an activity profile for legitimate operations and validate whether any system state deviates from the pre-established profile. (iii) Threshold value need to be set precisely to avoid false positives (iv) Computationally expensive, because a large number of profile matrices need to updated to minimize system overhead.
Behavior Based	(i) Detection capability of the underlying system depends on the set of unique signature given to detection engine. (ii) Capable of detecting only known attacks. (iii) Specifying a unique signature for a malware sample that covers all of its possible variations is a challenging issue.
Hybrid Based	(i) Aim to improve the detection accuracy of the IDS by combining both Misuse – and Anomaly – Based techniques. (ii) Capable of known and unknown malicious intrusive activities.

The other type of IDS, called, anomaly based NIDS often attempts to determine the natural behavior of the system to be secured, and raise an alarm whenever the current observation deviates a predefined acceptance threshold. The noted benefit of this method is its potential to discover new and previous unfamiliar attack types. Anomaly detection has been widely used by security applications such as monitoring enemies' activities in military, detection of cyber intrusions, and online credit card fraud detection.

In the recent past, researchers developed and presented many important and effective anomaly-based intrusion detection techniques that might act as additional line of defense in a computer network. There are many literature surveys exist in the field of network anomaly detection which gives most useful information about its challenges and issues to the defenders community [39-46]. Misuse based IDS types are commonly used to explore known intrusive malicious samples but anomaly based IDSs try to discover unknown malicious samples. Anomaly detection system is receiving more and more consideration from both real time application implementation and theoretical point of view.

Based on the type of techniques applied in the 'behavioral' classification of different events of the underlying system, Anomaly based NIDS (ANIDS) methods can be classified into four various classes such as statistical based, machine learning based, knowledge based, and combination learners based.

Statistical based ANIDS

In the statistical based methods, the normal behavior of the given data is determined whether an unseen sample is an anomaly or not using statistical inference test. Few existing research papers of this scheme are discussed below.

Tong et al. [47] and his colleagues proposed an anomaly detection system which can detect anomalous network packets using kernel component classifiers. For features that are associated with some major component, the values are extreme large. The non-linearity issue of network traffics was well addressed. Wattenburg et al. [48] presented a model to detect computer network traffic that might contain anomalies. This model importantly relies on statistical inference technique and first order alpha-stable model. The alpha-stable modeling is mainly used to classify online network traffics to detect anomalies that are associated with flooding and flash-crowd attacks. To achieve promising detection accuracy, the model used the generalized likelihood ratio test. Lee at al. [49] presented an anomaly detection method based on online over sampling principal component analysis algorithm was presented. It attempts to detect the presence of outliers from a huge data samples by updating through online. The process of oversampling the victim fragment and extracting the direction of the fragment, the proposed approach allows determine anomaly activities using the dominant eigenvector. Expert knowledge system plays a vital role for handling uncertain pieces of information.

Though statistical ANIDS have many advantages, its limitation includes the following: First, most statistical based techniques rely on the basic assumption of a quasi-process being stationary which is not realistic in real time. Second, ANIDS can be susceptible to blending attacks. Third, time taken to report a detection alarm is directly propositional to the time requirement of building models. Finally, choosing the right statistic for complex distribution is not straight forward.

Soft computing based ANIDS

Soft computing based ANIDS techniques rely on the construction of implicit or explicit model which are capable of categorizing the patterns to be analyzed. Soft computing system is thought of include techniques such as Artificial Neural Network, Artificial Immune System, Fuzzy logic, Genetic Algorithm, Rough set and clustering & Outlier because often no single technique can offer exact solution. Research works that belong to different categorizes is presented below.

The outlier aspect of clusters might be sometimes used for quantifying the deviance degree of a specific cluster [50] for the detection of cyber intrusions. The Nearest neighbor method was applied for data classification. The complexity of proposed unsupervised IDS is directly proportional to the size of test dataset and its attributes. Zhuang et al. [51] proposed a system named PAIDS (Proximity-Assisted IDS) with the goal of identifying the new and fast propagating worms. PAIDS has been trying to obtain enhanced performance by working collaboratively with existing anomaly-based IDS. Their approach assumes that during the worm-propagation starting phase, the infected victim hosts can be grouped based on IP address and DNS used. Jabez et al. [52] proposed an outlier based intrusion detection approach to detect cyber intrusions. A specific dataset was taken to measure the presence of intrusions by using the neighborhood outlier factor method. The use of limited dataset and training model are the two weakness of this approach.

Liu et al. [53] presented an approach that monitors malicious activities at the network to prevent known and first-hand attacks using unsupervised neural networks. This real hierarchical intrusion time solution uses Principal Components Analysis neural nets to avoid the limitations of sing lelevel structures. Conditional based anomaly detection is presented in [54]. It relies on finding difference among data attributes which are classified into environmental attributes and indicator attributes. This method detects anomalous if any deviation in the predefined value of indicator attributes. However, it does not consider environmental attributes in few cases. The precision of this method precisely depends on its learning phase. Adetunmbi et al. [55] proposed a rough set theory and then a k-NN classifier mechanism to determine network intrusions with the intention of increasing detection rate of the system and minimal false alarm rate.

Classification based Network intrusive detection techniques can achieve better results than clustering based mechanisms as they use labeled training instances. In conventional classification methods, additional information can be integrated by retraining the whole test dataset. But it is a time consuming process. Incremental type of classification algorithms [56] make use of such retraining more powerfully. Although these mechanisms are effective, they cannot identify known intrusions until sufficient training data is supplied to retraining process. Tajbakhsh et al. [57] aimed at constructing a model which produces fuzzy association rules with reference to classifiers and use them for detecting general network intrusions.

The fuzzy sets theory provides an effective way to categorize different classes of normal and/or anomalous. A training dataset that belongs to a particular type is validated by using matching parameters produced by the proposed approach. If the compatibility of a test sample falls the predefined threshold, then it is considered as anomalous. Geramitaz et al. [58] presented a network intrusion detection system that rely on fuzzy rules to recognize the occurrence of specific or general exceptional network patterns. However, training instances play a vital role to decide the detection accuracy of the system. The research paper [59] is devoted to the development of network intrusion detection system that uses genetic algorithms to construct detection rules. A chromosome of individual genes mapped to various aspects such as the root-user attempt, type of service attempt to use, or logged in or not. The author concludes that malware attacks that are common can be traced easily compared to unusual characteristics.

Visconti et al. [60] presented a performance based Artificial Immune System for detecting specific anomalous behavior. This system monitors anomalous activities by examining the detailed and different states of specific parameters. An intervaltype-2 fuzzy sets hypothesis is applied to dynamically produce different system status. To deal with issues of avoiding scale of a large network that can gather flow statistics information collectively, Duffield et al. [61] presented a machine learning algorithm to transform packet-specific measurement into measurements for discovering unusual anomalies. Specifically, the authors make the relationship between network alarms and feature vector which was built by extracting flow statistics on the same network traffic. And then a unique set of rules to discover anomalous is generated.

Abbes et al. [62] presented a method an intrusion detection system that adopts protocol analysis and the concept of decision trees. First, the proposed approach generates a unique adaptive decision tree for various different application layer protocols. Then, detecting either any data record contains anomalies or not. The anomalies record might include a variety of footprints such as scans, Trojans, and botnets. However this type of system requires precise datasets for exactly detecting anomalies and also not able to pinpoint unknown intrusions. Muda et al. [63] presented a two phase prototype model for detecting network intrusive activities. At first, k-means clustering algorithm is applied to categorize test samples into three clusters that belongs different groups such as probing attack data, DoS attack data, and authentic data.

Such data collection is achieved by setting the value of each cluster centers to the mean values obtained by groping appropriate data points. Finally, the authors apply a Naive Bayes classifier approach to classify sample data into five different accurate classes such as Probing, remote to user, user to root, and normal. Palmieri et al. [64] developed a two-phase anomaly detection scheme based on various distributed sensors which are located throughout the local area network. With the help of Independent Component Analysis, mechanism, the proposed scheme extracts the essential network traffic components. These components later will be exercised to construct the standard traffic profiles which act as vital role in the next phase to classify anomalous from normal traffics. Though soft computing techniques are popular, some disadvantages of them are pointed out below.

- Most techniques suffer from scalability problems and training.
- The insufficient availability of legitimate network traffic data makes the training of these methods more difficult.
- Rule generation by rough set methods introduces proof-of-completion.
- Fuzzy association rule based methods, tasks such as dynamic rule update, and rule subset identification to be performed at runtime becomes difficult task.

Combination Learners based ANIDS

A combination learner system and method incorporates multiple techniques to attain higher detection accuracy. Examples of such systems are Ensemble-based, Fusion-based, and Hybrid-based methods. Each method has its own advantages and limitations.

Some of existing research works that apply combination learners' techniques is discussed below. Tong et al. [65] introduced a hybrid neural network secure prototype that can be used to solve problems of either anomaly detection (or outlier detection) and misuse detection. It is capable of detecting collaborative intrusive attacks using memory of historical events. Folino et al. [66] presented the idea of ensemble paradigm that uses a distributed data mining algorithm with the intention of improving detection accuracy rate when discovering malicious or illicit network activities using genetic programming. The presented framework rely on distributing data across various autonomous sites and some useful knowledge is extracted in way from data and uses the pre-generated network profiles to forecast anomalous behavior.

HMMPayl [67] is a fusion-based intrusion detection model where the sequence of bytes identifies the payload part of a network packet and Hidden Markov Model (HMM) can be used to analyze them. First, the algorithm is used to extract features of network packets and then applies HMM assure the same sensitive power of n-gram analysis. HMMPayl uses the idea of Multiple Classifiers System to produce better classification rate and to avoid evasion of IDSs.

Knowledge – based ANIDS

In knowledge-based ANIDS techniques, network events or host operations are validated against predefined set of patterns or rules of attack. The objective for the design of such systems is to discover each known attack uniquely, thus handling of occurrences of intrusive activities become easier. Knowledge based methods can also be categorized into different approaches such as expert systems, logic-based, and ontology-based. Benferhat et al. [68] presented an intrusion detection method and alert correlation scheme by combining the expert knowledge with probabilistic classifiers. Especially, the presented approach uses three decision tree classifier algorithms, namely, Naive Bayes, Hidden Naive Bayes, and Tree Augmented Naive Bayes. The authors claimed that their approach achieves better results than existing benchmarking intrusion detection tools.

Ontology based model with soft computing technique for malware behavior analysis is presented in [69]. The proposed system contains two main stages. During the first stage, it collects information such as the event logs of network connections, registry entries, and local memory activities from the victim system to extract more information about unknown malware behavior.

Then the extracted information is used to build a unique ontology to discover malicious executions. The important advantages of knowledge-based ANIDS are those of flexibility and robustness. Their noted drawbacks are listed below.

- The design of high quality knowledge-based ANIDS is often time consuming and difficult task.
- Such methods may not be able to discover unknown intrusions.
- Non-availability precise signature of legitimate and attack data increases the overall false alarm rate of the system.
- Dynamic updating process of knowledge – base is very costly operation.

The taxonomy of existing network intrusion detection system for malware detection with their type of strategy, nature of detection, type of attacks detected along with the observations and limitations of the network level malware analysis are tabulated in Table 2.2. Another suitable place to detect and prevent malware attacks is the end-system. Compare to network analysis of malware detection, malware analysis at the end-system permits the defenders to closely monitoring the behavior and functionalities of a malware instance in a sandboxed environment. So that defenders can design an effective malware prevention system. As this thesis work focused on end-system part, network intrusion detection techniques have been reviewed but not considered for implementation or comparison.

Table 2.2 Taxonomy of various existing NIDS techniques

Exiting Strategy	Type of strategy / system	Nature of Detection	Attacks Handled	Limitations / Observations
Liu et al. [53]	Centralized	Non-real time	All Attacks	Defining rules to detect various types of malware is a challenging problem
Tong et al. [65]	Centralized	Non-real time	DoS & Probing	Suffer from false positives
Tajbakhsh et al. [57]	Centralized	Non-real time	All Attacks	Training the system to classify anomalous is needed
Su et al. [56]	Distributed	Non-real time	All Attacks	Cannot identify known intrusions until sufficient training data is supplied to retraining process
Folino et al. [66]	Others	Non-real time	DoS	Used pre-generated network profiles to forecast anomalous behavior.
Wattenberg et al. [48]	Distributed	Real time	Floods & Flash Crowd	Fail to handle encrypted packets.
Khan et al. [59]	Distributed	Non-real time	MS-SQL overflow attempt	Constructing rules to detect all possible malwares is a tedious task
Muda et al. [63]	Others	Non-real time	All Attacks	Few malware pretend to be legitimate which is hard to identify
Ariu et al. [67]	Others	Non-real time	DoS & Probing	The encrypted part of payload pose a serious challenge
Geramitaz et al. [58]	Centralized	Non-real time	Probing	Training instances play a vital role to decide the detection accuracy of the system
Lee et al. [49]	Centralized	Non-real time	All Attacks	Expert knowledge system plays a vital role for handling uncertain pieces of information.
Huang et al. [69]	Centralized	Real time	Advanced Persistent Threat	Not possible to detect all malwares. Inferred similarity level is not sufficient when dealing unknown malware

Challenging Issues of NIDS

However, despite the inaccurate specification of attack signatures tend to increase the false positive rate of anomaly-based system than in signature-based method. The most challenging issues of anomaly-based NIDS are:

- Despite the inaccurate specification of attack signatures tend to increase the false positive rate of the system than in signature-based method.
- To reveal detrimental vulnerable attacks those impel to generate inconsiderable traffic. Actually this type attacks initiated be an entity inside the secured network.
- Encryption can be used to prevent content based technique.
- Blending attacks may traffic to appear legitimate.
- It is difficult to identify malicious code that does not send or receive any traffic.
- Sheer packets/second reduce the NIDS ability to keep up and running effectively.
- It is required to have larger memory to analyze large amount of TCP connection fields to discover a wide range of malware attacks.
- These systems may also be required to track IP fragments, ARP packets, and other sensitive information.

2.3.1.2 Host Level Malware Analysis

In addition to network level malware analysis technique approach, another suitable place to supervise and investigate the malware's behavior is at the end-host. It is possible to detect a malicious code attack even before it gets executed in the victim computer.

However, current host-based malicious code detection techniques do not use effective models. As a result, these models cannot capture essential properties of a malicious executable. Traditional AV software principally relies on either file hashing or unique byte sequence of a malware [70]. But malwares with code polymorphism and obfuscation can able to bypass these techniques without detouring the natural execution flow of a system call. Static analysis based techniques mainly rely on using features of malware to identify its attack. Few existing work based on static analysis concentrate on byte code analysis using machine learning and data mining have been proposed as an alternate to traditional signature based technique [4].

In addition, an entropy based unique byte-code analysis technique also exists to detect encrypted, packed or embedded malware attacks [71]. Because, static analysis based techniques, importantly, relying on features of each malware authors embedded encryption or compression technique to obscure such analysis, graph matching, and clustering techniques [72-73]. Although this kind of malware defense technique works effectively against known malware samples but completely fails to deal unknown malware executable. This is because, new malware variants can be easily crafted by integrating techniques such as obfuscation, encryption, and self-modification into existing malware [74-75].

There have been few ideas proposed to detect native API hooks in Windows. Wang et al. [76] described an association mining based technique to analyze API execution flow. By associating API sequence using Portable Executable (PE) parser, they construct association rules and finally the malicious malware is identified. But this approach did not focus on various characteristic of a stealthy rootkit. Liu et al. [77] presented a review of rootkit detection techniques. Also, the authors developed X-Anti, a multi-way based detection method to detect different rootkits. In order to maintain their system, each node's information needs to be updated frequently and timely. Yi et al. [78] presented a review to analyze Windows rootkits and various stealth techniques to attack the Windows system. They also discussed various detection techniques that have been used by the detection tools today. Unfortunately, these techniques also bring new challenges to the detection and defense against rootkits. White et. al. [79] developed a plug-in to effectively identify the contents of all user allocations. But it will not describe every possible allocation. Additionally paging issues, data structure invariant and some undocumented APIs in Windows environment were not discussed.

Hejazi et al. [80] reviewed API calls on the stack to locate some data structure, especially those which handles encryption. Their approach works without knowing the structure of data which was in user space. This limits their ability to retrieve user data. Deng et al. [81] developed IntroLib, a tool to reveal user-level library call and behaviors which are generated by a malware based on hardware virtualization. In order to intercept library calls made by malware, IntroLib used page-table mechanism at the hypervisor level. This however fails to detect malware that could obfuscate its memory structure and library calls directly invoked by malware.

Researchers have proposed different techniques to protect misuse of Windows APIs functions. Because, malicious code can interact with Windows OS through Windows APIs function calls. Wang et al. [76] presented a static analysis method to detect malicious programs. This method collects the calling sequences of native APIs from legitimate programs and sets up a data model using Support Vector Machine (SVM). Then, the method proposed by Wang et al. detects malicious code by analyzing its calling sequences. Unfortunately, this method is unable to stop malicious code in real time and malicious code an easily mimic a legitimate calling sequence.

Rabek et al. [82] presented a static analysis approach to monitor system calls at run time and to identify software executables. This approach is simple, practical and effective for user land malware detection. But Rabek et al. approach failed to detect the malicious code directly invokes the kernel level service request. Wagner et al. [83] proposed a method to handle mimicry attacks in Linux environment. Their method records addresses of system call services into Interrupt Address Table (IAT). Whenever a process is waiting to get system service it was intercepted by their framework and checks whether the caller address is in the IAT. This method was not tested over Windows systems. Method such as the idea presented in [84] can also be used to understand the activities of a malware that comes from untrusted outside network using Honeypot.

With an increasing amount of malware adopting rootkit techniques to evade AV software, further research into defenses against rootkit attacks is absolutely essential. The taxonomy of existing solutions for detecting malware using static analysis with their type of malware to be detected, level of detection, performance overhead, along with the observations and limitations of the static analysis of malware detection technique are tabulated and given in Table 2.3.

Challenging Issues of Static Malware Analysis

- This method completely fails to deal unknown malware executable. This is because, a new malware variants can be easily crafted by integrating techniques such as obfuscation, encryption, and self-modification into existing malware.
- Malicious code can sometimes mimic a legitimate calling sequence.
- Because static analysis techniques rely on using features of malwares to identify them, setting precise feature set is a difficult task.

Table 2.3 Taxonomy of various existing Host level static malware Analysis techniques

Exiting Strategy	Performance Overhead	Type of malware detected	Level of Detection	Observations
Yi et al. [78]	High	Rootkit	Network	Packets with encryption, compression, etc pose a real challenge
Park et al. [73]	Acceptable	Worm	Host	Focused only on limited behavior of malwares
Deng et al. [81]	Acceptable	Malware	Host	Kernel level hooking is hard to detect
Mansoori et al. [84]	High	Intrusive Activities	Host	Malwares that pretend to be legitimate is difficult to detect
Cesare et al. [72]	High	Packed and polymorphic malware	Host	Unpacking such malware for analysis will increase time.
Canzanese et al. [74]	Acceptable	Malware	Host	Malwares with techniques such as obfuscation and self-modification is really difficult

Graph Based Malware Analysis

To eliminate the shortcomings of signature-based approach, defenders have utilized graph-based models [70] [85-86]. Graph based model is used to solve many complicated problems in engineering and technology. Graph mainly reflects the relationship between real world entities and attributes. A graph is an attractive method for analyzing malware attacks efficiently [87]. In order to analyze malware attacks in the Internet, Red team has been manually generating graphs. But their work has either error-prone or complex for a malicious malware that adopts API hook technique. So researchers are opting different technique such as code graph and call graph, control flow graph, data flow graph to build and analyze malware attacks.

An API Call Graph (ACG) is a candidate solution which is a suitable data illustration of the data and control flood of software programs. Additionally, it offers information about local data usage of a procedure and global data that can be exchanged between different procedures. Call graph acts as a suitable model either to study the behavior of a program or for tracking the flow values between different components of a program. ACG can also be used to recognize programs that are never invoked. The API function calls of an malicious executable can be extracted either through static analysis using binary code disassemble tools such as IDAPro [88] or by executing the malicious executable in a sandboxed environment and monitoring them using tools such as APIMonitor [89].

Graph Construction and Graph Matching Algorithms

A potential solution for constructing an ACG is through using static tools which can generate a multipath graph automatically. However, such tools fail to consider the actual API function calls being invoked by a malicious executable program. This is because, recent malware authors use techniques such as obfuscation and packing to evade detoured malware function calls. An ACG for a program can be easily constructed without considering the parameters associated with the calls. This can be achieved by constructing a table containing all function calls to be raised which represent nodes of the call graph and the reference between calls showing that represent all the edges of the graph. An API function call analysis must only do once and the order in which it is analyzed is not important. However, the construction of the call graph depends on the order of each function call to be analyzed when presenting the OS resources i.e., parameters of the function call. In programs, it is possible to discover invocation of many distinct API call references from single API call that contains OS resources. It is therefore, important to ascertain all API function call from such a reference API call to construct a complete ACG. In order to use ACG to detect the presence of a malware in a computer, it is important to compare call graphs that were generated based on malware samples against the ones representing legitimate programs. To compare two graphs, it is required to have a graph matching algorithm which is basically classified into two types such as exact and inexact. The exact graph matching algorithms have been used only when both given graphs have same number of vertices, whereas inexact graph matching is useful even when the number of vertices is different in both graphs.

There are three classes of graph matching techniques; namely, graph isomorphism, Longest Common Subgraph (LCS) matching, and graph edit distance. Both graph isomorphism and LCS are proven to be NP-complete [90] and computationally expensive to calculate edge weight. As a result, the research community has focused to devise fast approximation algorithm to avoid such issues. The graph edit distance matching algorithm is the best solution to solve inexact problems, but its complexity increases its overall execution time. There are many different techniques that could be used to generate an ACG and compare two given call graphs.

Guo et al. [91] proposed a binary translation approach to analyze and detect malware execution. The authors generated control flow graph based on malware's behavior and then another API sub-graph was generated to compare its activities. Lee et. al. [85] generated a call graph using malware's Portable Executable (PE) file format in which each node represents a system call and each edge represents a call sequence. The call graph is then converted into a code graph to analyze them. Li. et al. [92] presented a compiler based rootkit prevention technique which cannot permit the kernel level control data with arbitrary points. They prevented rootkit attacks by transforming kernel control data into indexes of jump tables in which only legitimate jump targets are allowed by the kernel's control flow graph. Since rootkit can inject jump instruction in any table, scanning them is a tedious and time consuming process.

Zander et al. [93] presented a graph theoretic framework for detecting one of the dangerous malware known as botnet. A graph portioning algorithm is used to separate botnets in a tainted network. But processing encrypted network packets pose a serious challenge. Karbalaie et al. [94] presented a malware detection system based on API functions call analysis. Every API call is depicted as a graph and then the Longest Common Subsequence (LCS) algorithm is applied to compare two graphs to determine the similarity between them. This method can able to confine system calls in execution and then generates behavioral graph. The graph which had highest similarity value is concluded as malicious. But LCS algorithm can be solved in NP-complete time which increases its computational time. Riesen et al. [95] proposed a framework for malware detection that rely on hybrid signature using API call graph. The proposed method solves the disadvantages of both signature and behavior based methods. This method can able to detect both known and unknown malwares with low false positive rate.

Few existing research works [90] [96-97] used different techniques such as signature matching, pattern matching, packet sampling approach for malware detection but such techniques suffer from computational complexity. However, this method generates more false alarms. Bai et al. [98] discussed a call graph clustering approach for malware detection. The authors represented malware sample as a call graph and compare these call graphs based on graph edit distance. Afterwards similarity score to cluster these malware samples is calculated. K-medoids and density based clustering algorithm DBSCAN techniques are also proposed for clustering malware samples. The authors stated that K-medoids clustering technique is not able to address to all malware families whereas DBSCAN technique can able to address almost all the malware families.

The investigations of existing research works have been shown to adopt different approaches to generate API call graphs. A major issue of the precise generation of an API call graph is its incomplete construction. This is because of the exclusion of API call and its associated resources during the construction of the API call graph which may result into inaccurate. Jaikumar et al. [99] presented a graph theoretic approach for the detection of different kind botnets present in a computer network. For graph generation, the nodes which represent an infected computer can be added into the set V if a new computer which is not in V exhibit malicious activity. A noted point this approach is the representation of weighted edge which is derived from the exploit co-occurrence of malevolent operations across the entire network.

The cost involved in the bipartition process is the weights of each edge that move towards from node in set P to the node in set Q. To make bipartition optimal, the cost involved with P and Q is minimized using normalized cut algorithm. The graph bipartition is a recursive procedure. The complexity of the graph construction algorithm is O(|V||E|). The pseudo code of the graph construction and graph partitioning algorithms are given below:

/* *Graph Construction Algorithm* */

1. begin
2. for (each compromised computer) do
    a. add compromised computer into V
    b. edge (e) ← assign edge weight that ranges from 0 to 1
    c. if (new infected computer found) then

begin

i. increase weight of new ‘e’

ii. goto step 2

3. end

4. else

begin

i. remove a node from V when infected computer postpone its malicious activities

ii. create a new ‘V’

5. end

6. end

/* *Graph Partitioning Algorithm* */

Step 1. Estimate |V| dimensional vector to bipartition the given graph

Step 2. Determine if the bipartite graph can be further bi-partitioned

Step 3. If step 2 is true then goto step 1.

Elhahi et al. [100] proposed an API call graph technique for the detection of malware attacks. First, the function calls to be executed by a malicious executable and its dependency parameters will be extracted to model an API call graph. The constructed graph is then compared with a database of malware call graph samples using a graph matching algorithm. However, the runtime complexity of the graph matching algorithm depends on the number of nodes in the query graph. The graph matching algorithm that relies on graph isomorphism is given below:

/* *Finding Optimal Subgraph Algorithm* */

1. begin
2. Similarity ← 0
3. for (each subgraphs (*a* and *b*) of API call graphs (Q and G))
4. for (each edge in *a*)
5. for(two vertices in *a*)
6. best path ← if two nodes belong to the same dependence subgraph

endfor

endfor

endfor

7. Similarity (a, b) ← maximum (similarity value)
8. Similarity = similarity + Similarity
9. end

Let Q be a query graph and G be a data graph. Let '*e*' be an edge in Q and two vertices x and y belong to G. In addition, *x* and *y* belong to the same subgraph in the API call graph. Then a modified greedy algorithm namely, graph edit distance is applied to find the best path (P) that involves e with x and y using Eqn. (2.1).

$$\text{Similarity}(Q,G) = \max_i \sum_{e \in E(Q)} \max_j \text{Similarity}(e,P) / |E(Q)| \qquad (2.1)$$

The complexity of the algorithm is O(|E(Q)||P|) where |E(Q)| is the number of edges in the query graph and |P| is the number of best paths.

Zhao et al. [101] presented a graph based approach to detect known as well as unknown malware. The function call graph of a malicious executable is extracted and analyzed through machine learning technique to identify unknown executable files. But representing the complete control flow of programs is a tedious task. Park et al. [73] proposed an approach for the construction of a behavioral graph that represents the execution behavior of a set of known malware instances. The behavioral graph is generated by clustering or grouping a set of unique behavioral graphs that represent kernel level objects and its features based on system call traces Even though this method produces 0% false positives, malware writers obfuscate legitimate system calls by rewriting the binaries or source code itself. Table 2.4 presents the taxonomy of graph construction algorithms and graph comparison algorithms developed for detecting malwares along with their complexity, observations and limitations.

Table 2.4 Taxonomy of various existing graph-based malware detection approaches

Exiting Technique	Complexity	Graph construction	Limitations / Observations	Graph Comparison	Limitations / Observations
Lee et. al. [85]	O(|V||E|)	V=system calls, E=relationship between system calls	Does not consider information about call graph parameters	Using graph union and intersection	Few edges are omitted
Park et. al. [73]	O(|V||E||μ|| |)	V=kernel objects, E=Dependency between two kernel objects	Discards potential information such as dependency between system calls	Using weighted common behavioral graph	Few edges are omitted
Zhao et. al. [101]	O(n × D(lg(|D|)))	V=system calls, E=Dependency between V	Do not consider parameter information aspects of system calls	Data mining and Feature selection	Lacking of training
Elhadi et al. [100]	O(|E(Q)||P|)	V=API calls and its associated resources, E=Dependency between V	Take longer time to construct a call graph	Graph Edit Distance algorithm	Takes longer time to compare call graph and model graph
Jaikumar et al. [99]	O(|V| |E|)	V=Infected computers E=Liklihood of similar activities between two Computers	Takes longer time to identify all infected computers in a network	Graph partitioning is applied. Graphs are separated based on the behavior of infected computers	Fail to detect a bot is which designed to perform its pre-programmed behavior in random order.

Challenging Issues of Graph based Malware Detection

- The runtime overhead of graph based approach is usually NP-complete
- Graph based approach is more robust and evade detecting unknown attacks
- Some malware which pretend to be legitimate is a challenging issue
- An attack graph model provides only limited view of security of a network and for a large network, analyzing a hug volume of attack scenarios is a tedious process

2.3.2 Behavioral Malware Analysis

One of the techniques proposed to address the weakness of traditional static malware defense mechanisms are based on behavioral analysis or dynamic analysis. In behavioral analysis method, the predefined properties of executing legitimate software are used to discover the presence of malware. Behavioral monitoring of each system call innovation made by processes of an executable is often utilized in recent anti-malware software. This approach works effectively, as it terminates malicious actions when it discovers such actions [6].

A complex problem which forces the anti-malware software to run longer time is the huge quantity of data being generated by malware on a daily basis. The cross-view validation technique for detecting hidden traces of stealthy malware have been learned and implemented for testing user- applications [102], within the kernel of the OS [77], inside the virtual machine [103] and using coprocessor hardware techniques [104].

2.3.2.1 User-Mode malware Detection and Prevention

Malicious rootkits refer to a collection of software routines designed to hide their presence and other malicious activities and enable the attacker to take control of the victim computer. Moreover, rootkits can also be used as backdoor to spy user or system's activities. The attacker can then capture sensitive information about either end-user or computer. As Windows is the one of the most popular and widely operating system, today much malicious software is being developed with the intention of affecting Windows OS. In order to launch malicious activities, Windows rootkits adopt a mechanism called 'hooking' which can modify the predefined execution path of a system call. However, rootkits need to access native APIs to accomplish their tasks. There exist two different rootkits such as user-mode rootkits and kernel-mode rootkits. The former types of rootkits work in ring 3 i.e. infect the victim computer outside the kernel and try to the original

system related files with fake detoured code. Unlike user-mode rootkits, kernel-mode rootkits affects the OS core and thus can permit the remote attacker to take complete control of the victim computer. Both type of rootkits intercept the pre-installed anti-rootkit software's in the victim computer and make sure that they does not include any footprints of its own. As mentioned earlier, rootkits use different types of hooking techniques to misuse both user-mode and kernel-mode data structures such as IAT/EAT and SSDT to remain hidden and evade anti-malware software.

Import address table hooking

The Import Address Table (IAT) is the most important call table of the user space modules. The IAT keeps the references of all routines exported by a particular Dynamic Link Library (DLL). And each DLL that an application is linked with, particularly at load time, will have its own IAT. Many executable files have embedded one or more IATs in their structure that are used to store the addresses of existing libraries that they import from DLLs. Most of the user land rootkits use the IAT hooking technique to intercept the API function calls. IAT entries are filled by the Windows loader at boot time. Thus, to maneuver an IAT, it is mandatory to access the address space of the request. One way to achieve this is by using DLL injection technique. Normally rootkits use DLL injection techniques to modify the address of the specific function in the IAT to point to the address of the rootkit function where it is presented. Therefore, when the application calls a specific function, the rootkit function is called instead.

Inline Hooking

Detour patching is another technique to divert the predefined execution path to malicious code without altering IAT call table entries. This technique is implemented by inserting a JUMP statement into the target routine to divert the execution path. Therefore, whenever the currently executing thread executes this jump instruction, the control is transferred to a detour routine. The original portion of the code from the target function which is deposited, in coincidence with the jump instruction returns back to the target code, is known as 'trampoline'. Therefore, the initial jump in the trampoline replaces a certain code when it is inserted and at the end. Using this technique it is possible to arbitrarily intercept the flow of execution. Intercepting every system service calls that use native API is a tedious and time consuming process.

Forrest et al. [105] expressed the idea of using a profile for system calls of high privileged processes. This approach offers many advantages compare to user-mode behavior based detection techniques. First, kernel root processes are more vulnerable than user-level processes onto a computer system. Second, they have predefined set of behavior that is more firm over time. Ideally, each kernel service request is mapped to a predefined set of system call chain of execution paths that it can spawned. Deng et al. [81] presented IntroLib, a framework for tracing user-mode library function calls made by malicious executables. IntroLib is enabled by hardware virtualization and residing outside the guest OS. In order to monitor the control flow transitions between library functions and malware, IntroLib utilized shadow page table technique in hypervisors. But IntroLib is virtual machine based in nature and cannot detect system call reordered attacks.

Lutas et al. [106] proposed a hypervisor based method of protecting user-mode processes against malware attacks in Windows. This method is also based on hardware virtualization. In order to protect user-mode processes against malware attacks, page-fault execution is injected in the guest OS to monitor all swap-in and swap-out memory operations. However, signature of different functions must be extracted from each OS separately to locate malware attacks. Aboughadareh et al. [107] presented a framework named, SEMU that combines both user-mode and kernel-mode analysis outside the guest OS to analyze malware attacks. The OS that runs on the virtual machine introspects all kind of operations between the OS and malware. At user-mode, SEMU logs all kind of activities such as system calls, input output controls, and information exported by DLLs.

Ahmed et al. [96] presented a runtime malware monitoring and detection system that rely on API call arguments (spatial information) and its dependence sequences (temporal information) information and machine learning algorithms i.e., malware detection rely on spatial-temporal information available in the API function calls. This malware detection approach requires to define the accurately and also cannot prevent evasion attempts. Another malware detection approach that relies on anti-debugging function which mainly used to prevent malware form analyzing a malicious program is proposed by Yoshizaki et al. [108]. If the behavioral patterns of an application differ from the behavioral patterns of legitimate application, then it is detected as malicious.

Table 2.5 presents the taxonomy of user-mode only malware detection by monitoring few important user-mode data structures along with their type of detection, samples taken, performance overhead, data structures monitored, observations and limitations.

Challenging Issues of User-Mode only Malware Detection Approach

- Malware that target to evade detection cannot be detected by only monitoring user-mode data structures
- The behavior of few malicious programs may behave and appears to be legitimate programs.
- Malware that directly invoke API function calls through kernel level cannot be detected.
- Advanced stealthy malware might duplicate its name and identities similar to benign programs and try to forge the kernel of the OS to get service

Table 2.5 Taxonomy of various existing approaches for detecting and preventing User-mode malware attacks

Exiting Technique	Samples Taken	Type of Detection	Performance Overhead	Data Structures Focused	Limitations / Observations
Hejazi et al. [80]	9 API functions	API hook attacks	Not validated	Sensitive API functions	Tracing API function calls and its flow is not resolved
Ahmed et al. [96]	416 Malicious samples 100 benign samples	Malware attacks	0 %	API function calls	Fail to encounter evasion attempts
Kumar et al. [102]	Legitimate applications	Memory resident attacks	Not validated	None	Scanning the code and data segment of memory is a tedious process
Deng et al.[81]	93 Malicious Samples	Hypervisor based Library function calls	< 15 %	None	Virtual machine based in nature and System call reordered cannot be detected
Aboughadareh et al. [107]	3 Malicious Samples	Hook Attacks	> 20 %	DLL files	Virtual machine based technique
Yoshizaki etal. [108]	Ago bot	Malware attacks	0 %	API call parameters and its sequences	Did not tested in real time

2.3.2.2 Detection of Hidden Entries in User-mode

In order to execute different pre-coded tasks, malicious software needs to perform some initial operations such as enumerating processes and services, opening a port, or establishing a network connection on the victim computer. A malicious rootkit can use either user-space Application Programming Interface (API) hooking or kernel-space API hooks in order to remain hidden. All the detection methods found in the literature implemented different techniques with the intention to assist the defenders in ascertaining rootkit footprints. These techniques range from identifying for unique signature pattern in the impending malware sample to supervising system behavior. The important issue with live analysis is the authentic information such as files and functions returned by the OS. The crosscheck-based comparison approach that aims to ascertain hidden processes and services concealed by stealthy malicious executables by comparing two different list of information. Blacklight, one of the Windows rootkit detection tools use cross-check based approach for discovering hidden footprints of malware. Its pseudo-code given below.

/* Hidden Process Detection Algorithm using cross-check based Approach */

Step 1. Start looping from 0 to 0x41DC valid Process Identifier (PID)

Step 2. Call OpenProcess() function on every PID. The OpenProcess function calls NtOpenProcess () function.

Step 3. The NtOpenProcess function calls PsLookupProcessByProcessId to verify whether the process exist in the list. PsLookupProcessByProcessId uses the PspCidTable to verify the same.

Step 4. NtOpenProcess function calls ObOpenObjectByPointer to obtain the handle of the process being checked.

Step 5. If successful then store the information about the process in a list.

Step 6. Goto step 1

Step 7. Obtain another list of information by using CreateToolhelp32Snapshot which extract information about all currently running process.

Step 8. Compare the two different list of information. The discrepancy between them discovers the hidden entries.

From the introduction of VICE, the first rootkit detection tool, many researcher proposed and implemented different cross-check based solution in the form of either algorithm or tool. Few existing solution that talked about discovery of hidden entries in Windows were discussed below.

In order to detect hidden processes, Kumar et al. [102] presented an approach to crosscheck two different process lists generated by calling higher-level user-mode APIs and lower-level APIs. The discrepancy between the two lists shows hidden processes. However, if the application runs under limited privilege rights then it would fail to manipulate system related resources and also cannot access protected memory areas. Saur et al. [109] discussed an algorithm to locate paging structures of impending processes which are concealed by malicious software. Schuster et al. [110] developed a search prototype to scan an entire memory dump to reveal hidden or terminated processes and threads. Burdach et al. [111] described an approach to enumerate unseen processes. This approach is actually implemented in the Windows memory forensics toolkit.

Betz et al. [112] developed a tool, called MemParser to enumerate the active running processes of the underlying operating system. Their tool can also dump the process memory. George et al. [113] programmed Kntlist to analyze and evaluate kernel's internal data structures such as list and table to extract important processes, threads, and other data. The Windows kernel keeps many tables and list to manage all of its resources. By inspecting them, it is possible to catalog all items which can help for detecting for malware footprints. However, this approach cannot detect kernel objects that are influenced by the OS and processes have already been terminated but not entirely erased from the memory.

As many rootkits adopt a hooking technique to hide their traces, Yin et al. [114] programmed HookFinder, a tool to find hooked activities of unfaithful binaries using fine-grained crash analysis approach. But, it has not been revised since 2008. Another similar tool, HookMap [115] has utilized backward data segmentation technique to trace address of memory pages which can be abused by malicious rootkits to embed hooks. The crosscheck view approach was used in GhostBuster [116] to detect rootkits by comparing two different sets of information, inside-the-box and outside-the-box. However, rebooting the OS during an external scan would produce a complexity overhead.

Microsoft's Rootkit Revealer [117] is a rootkit detection tool which generates two different lists of information in the same underlying system to reveal the presence of rootkits. However, it can detect persistent rootkits that can only hide files and registry-related settings in Windows operating systems. But it does not detect hidden processes and services. To detect hidden processes, they compare the process scheduling list to the list of all processes within the kernel. Though their discussion is on Linux, the same concept can be applied to Windows.

Rootkit detection systems require inspection from outside the potentially compromised operating system. For example, virtual machine introspection [118] runs a security service within a privileged domain and uses memory introspection APIs exposed by the hypervisor to analyze the state of a guest system. Most existing anti-rootkit detection tools crosscheck information generated by tainted system calls against system information generated by its own for identifying rootkit traces. A stealthy malware conceals its footprints by controlling OS function calls which cannot be hidden. But, using offline investigation to reveal hidden traces of a malware is very difficult. In short, most existing techniques suffer from issues such as lack of integration, high false positive rate, overhead produced by complex configurations and scalability and performance issues.

Jones et al. [119] described and implemented a tool named, Lycosid, a virtual machine monitoring method for the detection of hidden processes and services. Lycosid uses cross-view validation approach to compare the information obtained about processes in a guest OS and information obtained at lower - level using Ant farm VMM component. However, Lycosid obtained both information from virtual machine which is not always produces trusted output. In addition, guest level component is vulnerable to malware corruption.

Richer et al. [120] presented a system named, Linebreaker which can able to detect hidden rootkit footprints by comparing hypervisor level information and OS level information obtained from guest OS. Similar work presented in [121] also used cross-view comparison approach for detecting hidden entries of rootkits. This approach compares the VM extracted states and hypervisor extracted VM's execution states. The taxonomy of detecting hidden processes and services of malicious programs in user mode for optimizing or improving the a malware detection approach samples taken for testing, performance overhead, platform implementation, type of detection, and observations and limitations is presented in Table 2.6.

Table 2.6 Taxonomy of various existing approaches for detecting hidden entries of a malware

Exiting Technique	Samples Taken	Performance Overhead	Implementation Platform	Type of Technique	Limitations / Observations
Wang et al. [116]	10 File hidden malwares and 120 spyware	< 10 %	Windows	Cross-view for detecting ghostware programs that hide files	It cannot detect malware that hide processes in user-mode
Jones et al. [119]	50 Processes	0.7 – 5.3 %	Linux	Cross-check based approach	It obtained both information about processes from the VMM and guest
Kumar et al. [102]	50 Samples	< 10 %	Windows	Scanning memory	Malware affected memory pages are used
Fu et al. [122]	2 rootkit malwares	Nil	Windows	Cross-view	Few malware can bypass the anti-rootkit detection tools used in the experiment
Xie et al. [121]	3 applications	2.5	Linux	Cross-check based approach	Trust hypervisor level information
Richer et al. [120]	13 Rootkit samples	9.5 %	Windows	Cross-check based approach	Rely on hypervisor level information

2.3.2.3 Combination of User-mode and Kernel-mode Protection

Attackers may also tricky end-users through drive-by-download attacks to compromise a vulnerable computer or network. For launching drive-by-download attacks, remote attackers might use vulnerable browser or its plug-in. Hsu et al. [123] presented a scheme namely, BrowserGuard to detect and prevent drive-by-download attacks by analyzing each and every downloaded objects. Based on the outcome of the analysis phase, BrowserGuard either permit or block the object being downloaded without user's consent. Without analyzing the runtime state of the malicious object being downloaded and its source file, the BrowserGuard can able to block the execution of a malicious application. However, BrowserGuard was designed to support only IE 7.0 browser that runs in a Windows platform. Malware that incorporates rootkit technique may also become a serious threat to system security. Baliga et al. [124] developed an anomaly based prototype namely, Gibraltar for automatically discovering kernel level rootkits that target modifying kernel level data structures. Gibraltar mainly applied the concept of data structure invariants to identify kernel level malicious rootkits. Though Gibraltar was effective against kernel-mode malware attacks, it was designed to identify only 23 rootkits.

Remote attacks might also use vulnerable software to inject malicious code with the intention of hooking and compromising system services. Sun et al. [125] proposed a behaviour-based method for analyzing the behaviour of an API function call both in Windows and Linux platform. This approach only blocked malicious API function calls and permit all legitimate system services being directly serviced by the kernel. But identifying malicious operations that directly calling lower level API functions pose a serious challenge. Similar to remote attacks, insider attacks is also a serious threat to system security. Rajagopalan et a. [19] presented a policy-based mechanism namely, Authenticated System Call for the purpose of discovering compromised applications in Windows. The authenticated system call mechanism used an extra argument in addition to system call arguments to check whether an application is malicious or not. However, such authenticated system call mechanism required to incorporate precise set of policies to locate malicious operations.

The system proposed by Nguyen et al. [128] captures malicious code attacks by hooking kiSystemService Dispatch Table (SSDT) in the kernel mode. Therefore, any malicious code which does not follow the predefined route execution will be detected as unauthentic code. The idea was implemented without modifying the kernel of the Windows OS, which results ease implementation. However, such proposed approach had certain limitations. First, invocation of a system service request with incorrect dispatch ID results into system crash. Second, implementation difficulty was neglected which weaken their system's security strength. Third, guessing attack can easily compromise their solution. As an API call indicates how a particular task is executed, the values that are supplied to it may also important to detect malware attacks. The behavioral operations of binary files are extracted by executing them in a controlled environment. The feature sets are defined through API calls and its parameters which may then used to create vectors. The feature that deviates from legitimate executable is detected as malicious.

Wang et al. [115] proposed a scheme namely, HookMap with the aim to monitor and analyze flow of execution of an application to discover the kernel level hooks that could be possibly hijacked by malware for evasion. However, HookMap had challenges such as accurately identifying the kernel level hooking with relevant run time context information. In addition, dissimilarity in the kernel-mode is also a serious problem. A similar approach was proposed [126] to protect SSDT through monitoring user-mode data structures. The taxonomy of detecting hidden processes and services of malicious programs in user mode for optimizing or improving the a malware detection approach samples taken for testing, performance overhead, platform implementation, type of detection, and observations and limitations is presented in Table 2.7.

Table 2.7 Taxonomy of various existing approaches for detecting and preventing malicious code attacks at Kernel-mode

Exiting Strategy	Test Samples taken	Accuracy Rate	Performance Overhead / False Positives	Type of Malware Detected	Level of Detection	Limitations / Observations
Rajagopalan et al. [19]	3 samples	96 %	0.73 – 7.92 %	Compromised applications	User-mode and kernel-mode	Vulnerable to frankestin attacks
Nguyen et al. [128]	5 applications	95 %	3 – 9 %	Code injection attacks	User-mode and kernel-mode	Supply of incorrect PID lead to system crash
Wang et al. [116]	8 Rootkit malwares	98 %	5 – 7 %	Kernel hook attacks	Kernel-level	It fails to detect kernel non-control data hooks
Sun et al. [125]	8 applications	97 %	8.8 – 9.10 %	Malicious code attacks	User-mode and kernel-mode	Malwares that target higher level APIs cannot be detected
Hsu et al. [123]	7 – 18 antivirus terminators	98 %	0.42 – 1.77 %	Drive-by-download attacks	User-mode and kernel-mode	It can support IE 7.0 on a Windows System
Salehi et al. [127]	385 benign samples 100 malware samples	98.40 %	3 %	Malware attacks	Kernel – mode	It cannot detect unknown malwares effectively

Challenging Issues of Behavioral Malware Analysis

- The increase in computational overhead and false positive rate limit its real-time applicability
- Recent malwares can mimic as benign and react later which is very difficult to handle
- System call based dynamic analysis technique rely on the assumption that the predefined execution flow of an executable can be coarsely
- Attacks such as mimicry attack and shadow attack etc. can succeed against process level signature based system call detectors.

2.4 Extract of the Literature Survey

Even though advanced anomaly detection approaches can detect unseen type of intrusions in real time, it is still relatively immature in the field of network security. The computer network traffic seems to be a complicated dynamical system, triggered by many factors. Though there were various different schemes have been discussed in the past to detect exceptions, they are mostly based only upon traditional statistical results. In these schemes, all network factors are combined to examine the dissimilar network traffics. Additionally, detection approaches based on deep packets analysis have reached their limits. If attackers implement packet-encryption, then network level malware detection becomes a very challenging problem and therefore, network based malware analysis technique has not been focused in this research work but surveyed to understand exiting techniques and to understand the functionalities of different malware variants.

In addition to network level malware detection technique, another appropriate spot to detect and prevent malware attacks is at the end-system. Though static malware analysis technique is effectively applied to detect malwares, it is well suited for detecting malwares with known signatures. Because such technique relies on signatures or properties of malwares, malware writers embedded techniques such as encryption and compression to complicate its detection. Many existing graph based approaches were effectively only to detect certain malware activities at the end-system. Such approach needs human assistance and suffers from false positives when dealing a huge amount of attack scenarios. Additionally solving graph based problem falls under NP-complete problem.

Also, malware analysis technique such as AV software mainly relies on using features of existing malwares to discover its malicious activities. Therefore static malwares techniques are not suited preventing stealthy unknown malicious code attacks.

Many different malware defense mechanisms were proposed based on dynamic analysis in the past to detect malware attacks at user-mode in Windows. Although these techniques were proposed to address the weakness of static malware analysis techniques, attacks such as lower level API hook attacks and malwares that incorporate rootkit techniques to evade AV software and can bypass them.

In order to discover the hidden entries of a malicious executable to optimize the user-mode malware detection approaches, many different cross-check based algorithms have been developed and proposed by the research community. In addition, much familiar and widely used anti-rootkit detections are also available. But optimizing the detection rate, accuracy rate, and false positive is a challenging problem.

Though many existing policy based solutions overcome some of the problems of dynamic malware analysis technique, setting exact policy set for each and newer malware variant is a difficult task. In addition, the existing process authorization techniques are not sufficient to prevent kernel level malicious code attacks. As a solution, combining user-mode information and kernel-mode information can provide stringer security against stealthy malicious code attacks.

Hence, to overcome the limitations of the existing malicious code detection approaches and algorithm and to provide best detection rate, the following approaches are to be devised.

(i) Devising new graph-based malware detection approach using two new algorithms to model API function call as a graph and comparing two given graphs respectively to rid of the drawbacks of the existing graph-based static malware detection approaches and provide better results than existing approaches reported in this thesis.

(ii) Devising a new user-mode malware detection approach for detecting and preventing malicious code API hook attacks.

(iii) Devising a new cross-check based algorithm to discover the hidden entries of a malicious executable. The proposed algorithm can also be used to optimize the user-mode malware detection approach.

(iv) Devising a new security enhancement mechanism using both user-level information and kernel-level authentication to detect and prevent malicious code attacks that target hooking system services in Windows platform.

2.5 Summary

In this chapter, various categories of malware detection and prevention techniques such as intrusion detection systems, graph based approaches, static malware analysis, behavioral malware analysis, and security policy enforcement techniques have been reviewed. The advantages and limitations and challenging issues of well-known and widely applied malicious code defense techniques in each category are pointed out. Further, the need for mandatory runtime authentication on all suspicious system call invocations made be processes of an executable which improves the security strength of the kernel is pointed out. Finally, this chapter is concluded extract of the literature survey and challenging issues in the current research of a literature survey.

CHAPTER 3

PROPOSED GRAPH-BASED APPROACH TO DETECT MALICIOUS CODE ATTACKS

Today, many modern malware developers is taking the advantage of API hook technique to take the control of the victim computer which making it difficult to detect their presence. Because of the sophistication of rootkit tools, a remote attacker can use native API to compromise any computer which can later be used for many illegal activities such as sniffing network lines, capturing passwords, sending spam and DDoS attack, etc. Thus to protect end-system by identifying and preventing native API malicious code hooking is a challenging problem to the defenders. Today, many different malware-analysis tools incur specific features against malwares but manual and error-prone. Therefore, a behavior-based monitoring detection system has been proposed to effectively detect native API hooks in user-mode. Unlike other malware identification techniques, this approach involved dynamically analyzing the behavior of native API call hooking malwares. A brief preamble about the significance of graph based malware detection is discussed in the subsequent section.

3.1 Preamble

Though malicious computer software can be referred to with different names such as virus, worm, Trojan, spam, botnet, etc., their ultimate goal is to cause damage either to the end-computer or end-user. The advancement in computer technology allows the malware writer to integrate obfuscation technique to evade detection specifically API hooking. Unfortunately, signature-based detection approach such as anti-virus software on the end-computer is not effective against attacks such as system call reordering. To overcome this shortcoming, many different behavior-based approaches have been offered. However, these approaches bear limitations such as higher false positives; fail to detect zero-day attacks, fails to improving the accuracy rate from past experience, etc. In this chapter, an API call graph approach has been proposed to capture detouring activities to be performed during malicious software execution.

As graph based approach can be effectively applied to replica complicated relation between entities, it is opted to visualize malicious rootkit behavioral activities by monitoring system API calls during runtime. This will help the defender to optimally capture malicious system calls from benign calls.

3.2 Proposed Graph-based Approach

It is assumed that most malicious malwares are developed by inheriting characteristics from its previous version. For example, the various versions of TDSS rootkit are: TDL1 which was designed to load and run at the time of booting the operating system which was designed with the intention of infecting system drivers. TDL2 appears to be same as TDL1. However, it includes different names with random string and also imports new technique to avoid detection and removal. In order to obtain control over the victim computer, TDL3 patches the disk controller driver. Some features of TDL2 were updated to make detection and removal more difficult. The aim of TDL4 variant is the same as that of TDL3, but patched master boot record to make infection of computers with 64-bit processor. The overall flow diagram of the GraMD approach is given in Figure. 3.1.

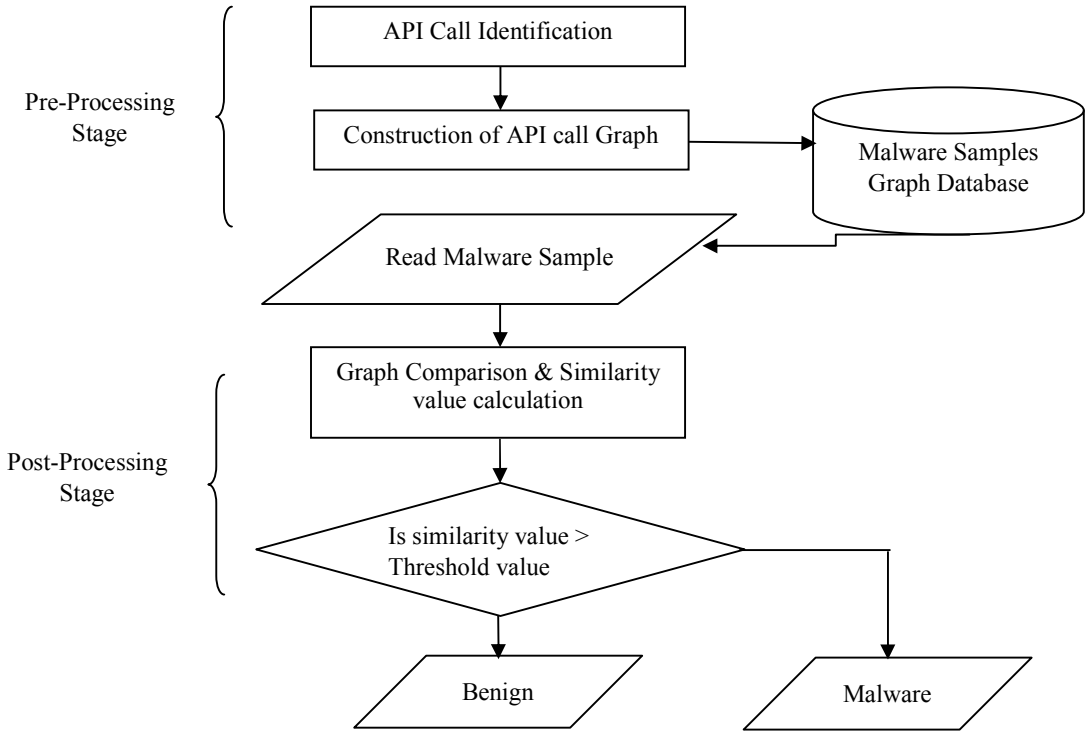

Figure 3.1 Flow diagram of the proposed GraMD approach

API call Graph Generation

The proposed GraMD consists of two important stages which are referred as pre-processing stage and post-processing stage. The pre-processing stage is responsible to collecting necessary resources of an API function call and integrating them as a graph. The API call graph is generated using the proposed call graph construction algorithm using the extracted API resources. Which act as graph nodes. The edge in a graph represents the relationship between two nodes. A directed graph G=(V, E) is visualized as a call graph in which V is a set of vertices that represent a function call of an executable program and E is a set of edges which depicts the relationship between two system calls. A directed edge (u,v) in E represents a function call of the program, u→v. GraMD attempts to discover the malicious code attacks which integrate API detouring technique to launch their illegal activities using an API call graph approach. As API function call is a finite set of sequence of invocations with ordered parameters and also they communicate with the use of handles (Unique identifier), GraMD can identify all necessary resources to construct an API call graph for a corresponding function call. For example, Figure 3.2 shows the OpenProcess API function call that visually approached as a call graph. The graph shows all internal function of the OpenProcess function and all its subsequent calls.

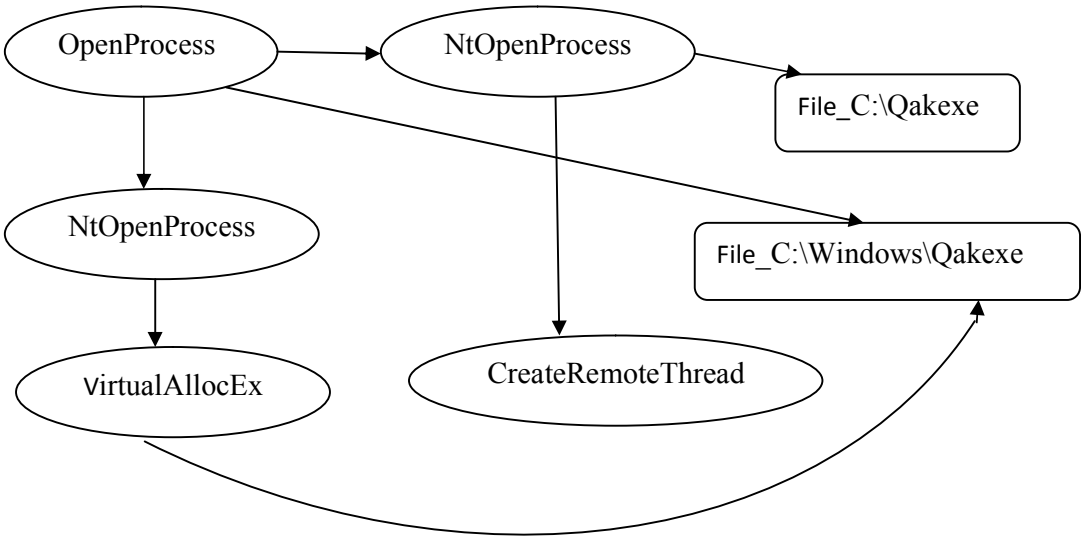

Figure 3.2 API call-graph

Algorithm for computing Graph Edit Distance

For given a large set of graphs, the difficulty for the computation of the exact subgraph of the call graph is to determine exact edit distance that best matches to the approach graph. Because new malware can be created from its predecessor and by approaching the malware sample as a call graph, the difficulty of categorizing malware variants is to discover subgraphs that best match with high similarity. If the call graph has more than one edge in common with the approach graph then measuring the similarity of best neighbor matching edges is chosen. For two graphs Call Graph (CG) and Approach Graph (MG), the problem of determining approximate subgraph is to discover the best match subgraph S_a using Equations 3.1 and 3.2

$$S_a = para \max_G . \ \text{sim(MG,CG)} \quad (3.1)$$

S_a = Maximize {Simval(CG,MG) = 1} then CG is isomorphic to MG. (3.2)

where $CG \subseteq G$ and sim(CG,MG) represents the level of matching between CG and MG.

Pre – Processing stage

The pre-processing stage starts by identifying and gathering all API calls of a running executable along with all its associated identities such as registry enumeration, process manipulation, network resources, memory management, etc. This phase is indented to identify all the nodes and edges of an API call graph. And then, a data dependent edge of a pair of nodes is generated using the parameters connected with two API calls. A malicious program is executed in an isolated environment and instructed to identify the resources of a function being executed. Based on the function call traces, GraMD identified the relationships between two function calls. An API call and all its associated parameters are used to generate a graph using ACA algorithm which is given in Figure 3.3.

To construct an API call graph, all the functions associated with an executable will be identified by referring the Import Address Table (IAT) and Export Address Table (EAT). If an API function call is raised then its corresponding name and its parameters are extracted to construct an API call graph in which each node contains the function name and an edge is established using its parameters list. If two parameters in the parameter list are same, then it reflects the dependence between the current and previous API call. Finally, all the API call graphs are kept in a database.

```
/* Algorithm for Generating API call dependent graph */
1    begin
2        Extract all function calls of an executable
3        Select a function
4        if (Native API call)
5         {
6           s.node ← function_name;
7          Get parameters(function_name);
8          d.node ← recursive(pointer call analysis);
9        // Recursive function call
10         Anynode.generategraph();
11        }
12       endif
13      Store it in database
14   end
```

Figure 3.3 Pseudo code for ACA algorithm

Post – Processing stage

Today, a malware writer can develop a malware by updating new features and techniques with its predecessor rather than coding from scratch. This information can help the defender to reduce the complexity of considering all kinds of addiction while inquiry the approach graph. The objective of the post-processing stage is to generate a subgraph of the data graph by referring the approach graph. The intricacy of a graph comparison approach has increased when all kinds of dependencies between the edges and nodes in Call Graph (CG) and Approach Graph (MG) are accounted. In order to avoid this issue, each graph is simplified by finding the best matching subgraph (SG) that can be used to exactly identify a malicious API hook attack. The idea of graph comparison is to generate a subgraph of CG by best matching the MG. The GED technique [95] is applied to determine the similarity between CG and MG. The value of each edge is normalized between 0 and 1.

The modified graph matching algorithm given in Figure 3.4 mainly used two compare two given graphs and to compute the similarity value. Since both the CG and the subgraph of MG contain only limited number of edges, the graph matching algorithm performs better. This will also help us to prove its correctness.

```
/* Algorithm for matching two graphs */
1     begin
2         simval← 0;
3         Get the paths of (P1, P2)
4         if (Paths  P1 and P2 has same label for all edges) then
5             for each path find similarity using GED do
6             simval ← simval+simGED(P1, P2)
7             s.node ← function_name;
8         endfor
9         p← number of paths in Q
10        simval(Q, G) ← simval|P|;
11        if (simval(P1) = =  simval(P2) )
12            Msg “malicious API call found”;
13            Alert();
14        endif
15    end
```

Figure 3.4 Pseudo code for GMA algorithm

3.3 Experimental Setup

All experiments are carried out on ACER Core Duo with 2.93 GHz processor with 4 GB RAM and the host machine runs windows 7 operating system. For each system call, a corresponding CG is generated and the same is compared with MG. By analyzing many root malware attacks, the threshold value of 97% is set to determine whether a generated call graph imitates malicious activity or not. If the calculated similarity value of any malware exceeds the predefined threshold similarity value, then it can be suspected as a malicious malware.

By and large, researchers have opted a graph based approach for comparing two graphs that aims to detect a malware attack with their own malware datasets against various assessment techniques such as LCSA, N-gram, etc. In order to evaluate the robustness and effectiveness of the GraMD, a malware dataset that includes different families of attacks is collected from some reputed websites [129-130]. Table 3.1 lists the malware samples along with the technique integrated used for testing and training against various existing techniques. Each malware sample is run in an isolated environment to identify and extract API calls and its parameters using the API Monitor tool [89].

Table 3.1 Various Malware Families used for evaluation of GraMD

Malware Family	Hook Type	Hook Technique
Rootkit.win32.Agent.	Kernel	DKOM
Rustock.A.	User	Hook
Rustock.B.	User and kernel	Hook
Rustock.C.	User and Kernel	Hook
AFX rootkit	User	Hook
FU rootkit	Kernel	DKOM
HideProcessHook	Kernel	Hook
Phide	User	DKOM
Shadow walker	Kernel	DKOM
YYT rootkit	User	Hook

The API calls of an executable are identified by analyzing binary files statistically using tool like IDA Pro [88] or by executing the binary files dynamically in an isolated environment using a tool like API monitor. To dynamically analyze a malicious executable files the following three operations are performed. First, the obfuscation cover is removed. Secondly, unpacking and decryption are performed over the executable. Finally, functions are extracted with all its parameters. Finally, the call graph is generated for each API function call using the algorithm given in Figure 3.3. In order to utilize call graphs to exactly locate API hook attacks, it is necessary to compare a call graph that reflects the API hook behavior against those that reflect benign behavior. To compare two call graphs, a modified GED algorithm i.e., GMA algorithm is applied to determine its similarity by matching CG with MG. When two graphs have the similarity value either greater than or equal to the predefined threshold value, then it is said to be exact matching or suspicious malicious call.

3.4 Experimental Results and Discussions

The evaluation results are obtained by conducting simulation experiments for comparing the proposed GraMD method against existing methods using some common parameters such as true positive, false positive, detection rate, and accuracy rate. These parameters are defined and calculated below.

- True Positive (TP) occurs when a malware is correctly detected as a malware.
- False Positive (FP) occurs when a legitimate sample is caught to be a malware.
- Detection Rate (DR) = $\frac{(TP)}{(TP+FN)}$ (3.3)
- Accuracy Rate(AR) = $\frac{(TP+TN)}{(TP+TN+FP+FN)}$ (3.4)
- False Positive Rate (FPR) = $\frac{(FP)}{(TN+FP)}$ (3.5)
- Receiver Operating Characteristic (ROC) curve – It is a two dimensional graph used to visualize the performance of the proposed approach by plotting TPR on the X axis against FPR on the Y axis.

The effectiveness of the proposed GraMD approach in detecting malware hook attacks is evaluated using a dataset consists of 200 malware samples and 50 benign applications. The benign applications are gathered by freshly installing application on a computer that runs a fresh copy of Windows XP OS. The malware dataset is downloaded from a publicly accessible website called 'VX Heavens' and divided into three groups namely, Rootkits, Worms, and Trojans, which a group on the average contains 70 malwares and 15-17 benign programs. Then, 70 percent of the dataset is used to train GraMD and 30 percent for testing it.

The accuracy and detection rate of the proposed GraMD approach is further investigated by conducting the experiments twenty times and the averaged minimum and maximum similarity value of all malware samples from each group is calculated and averaged. Table 3.2 shows the averaged similarity values and detection capability of each group of existing approaches including the proposed GraMD.

Table 3.2 Comparison between similarity value and detection capability

Technique	Maximum SV	Minimum SV	Number of malware samples not detected
Family: Rootkits			
Park et. al. [73]	87.74	31.28	4
Zhao et. al. [101]	92.68	40.56	2
Elhadi et al. [100]	97.73	58.62	1
Proposed GraMD	98.23	58.08	1
Family: Worms			
Park et. al. [73]	82.21	12.18	2
Zhao et. al. [101]	88.10	19.23	2
Elhadi et al. [100]	81.69	12.01	1
Proposed GraMD	92.08	42.34	0
Family: Trojans			
Park et. al. [73]	80.86	31.90	2
Zhao et. al. [101]	94.79	18.64	2
Elhadi et al. [100]	79.32	33.83	1
Proposed GraMD	90.29	43.56	0

Table 3.2 shows that the proposed GraMD method has achieved an average of 93.20 similarity value. Among all, the Park et al. method failed to detect 8 malware samples in total produces lowest performance. Whereas Zhao et al. method undetected only 6 malware samples. The Elhadi et al. method failed to detect an average of one malware sample but GraMD undetected only one malware sample and surpasses the method proposed by Elhadi et al. Another important consideration of GraMD approach is to evaluate its effective against the detection rate benign samples.

Figure 3.5 shows that the methods proposed by Park et al., has achieved an average of 15.9% of FPR, whereas, Zhao et al., with 11.9 %. In case of rootkits malware samples, the method proposed by Elhadi et al., and GraMD failed to detect one instance. The reason is, more advanced stealthy malware with rootkit technique intelligently evade its detection. The proposed GraMD approach detected all instance of Worms groups and Trojans groups correctly and surpasses the existing approaches.

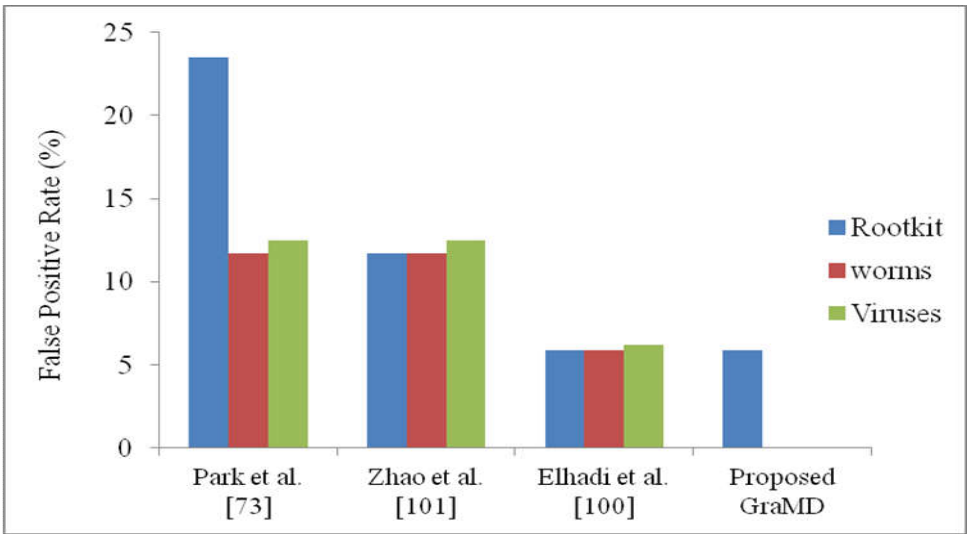

Figure 3.5 False Positive Rate of the proposed GraMD and existing Approaches

Figure 3.6 shows that the proposed GraMD approach achieved 100% detection rate in the worm and Trojan groups but only 97.68 % in rootkit group. This is because, before execution, few malware rootkit samples checks whether it is running in a sandboxed environment. If so, then they postpone its near future activities. The method proposed by Elhadi et al. achieved an average of 97.59% against the same dataset which is the second highest in the list. Moreover, the method proposed by Park et al. has achieved the lowest DR in all malware datasets.

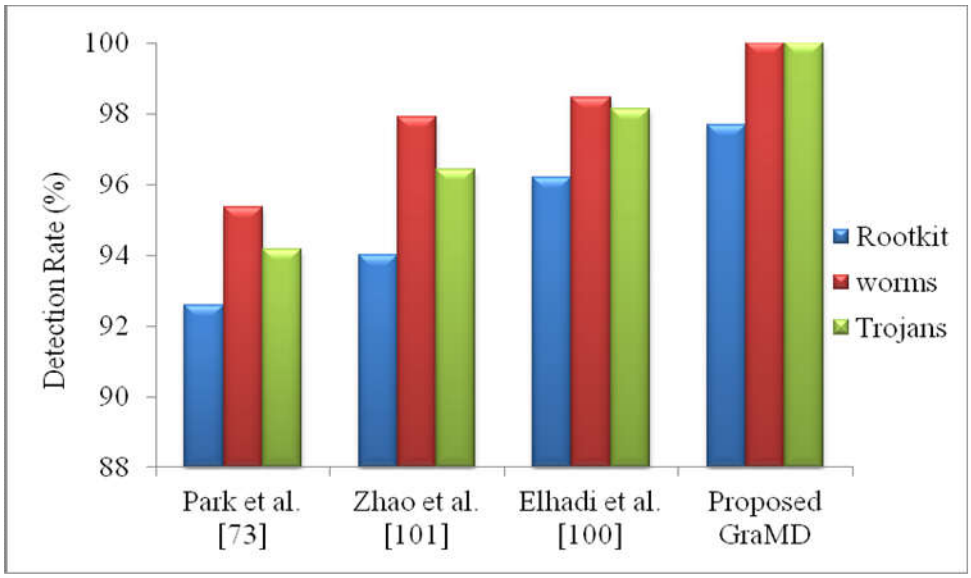

Figure 3.6 Detection Rate of the proposed GraMD and existing Approaches

Figure 3.7 shows the overall accuracy rate of GraMD approach and other existing techniques against all 250 malware samples. The GraMD approach achieved the highest accuracy rate of 98 % against all 50 rootkit malware samples, but achieved an average of 100 % in the Trojans and worms groups. Elhadi et al. method has achieved next best result than other comparable techniques with an average of 98.21% AR. Moreover, the methods proposed by Park et al. and Zhao et al., have achieved an average accuracy rate of 95.09 % and 96.53 % respectively.

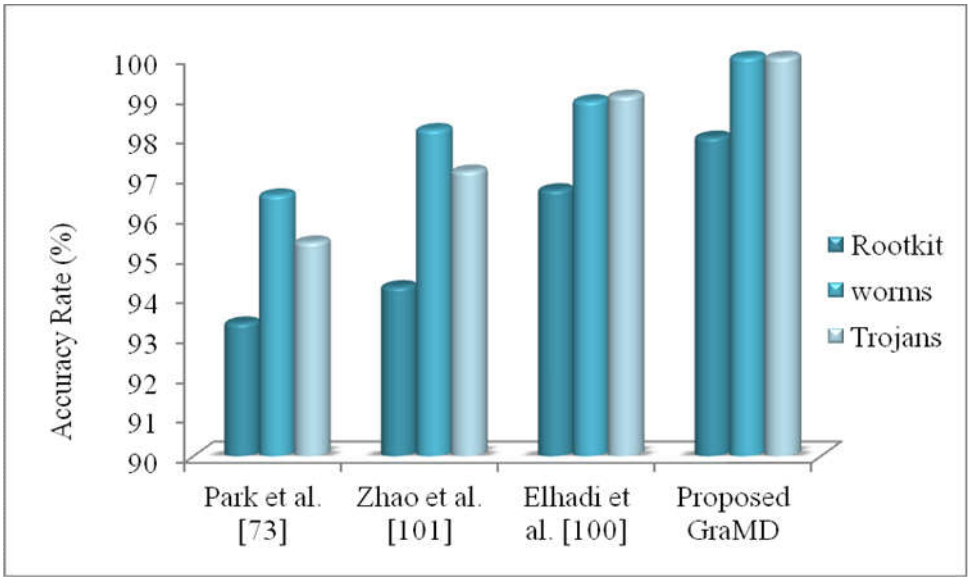

Figure 3.7 Accuracy Rate of the proposed GraMD and existing Approaches

From the above experimental results and discussion, it is clear that GraMD approach outperforms than the rest of discussed existing approaches in all aspects. A game-theoretic approach is also used to ensure the optimization of resources consumed by the GraMD approach. The game theoretic model dynamically selects a specific API targeted by stealthy rootkit malware based on the expected attack scenario.

3.4.1 Mathematical Verification

A two-player repeated non-cooperative game approach is selected, since malicious code attacks are trying to compromise the victim computer repeatedly. More specifically malicious code attacks that target hooking user-mode data structures during runtime is considered. It is assumed that the game is played between the two players: the GraMD and the Malicious Code (MC). The MC is the attacker and the GraMD is the defender. The objective of MC is to use ‘n’ number of processes or APIs (e.g. in worst case) from the

victim computer with the intention of performing and launching some illegal activities. A malicious rootkit attack is successful when atleast ‘m’ APIs out of ‘n’ APIs is utilized. It is assumed that a malware attack will not be achieved in single step. Therefore, the GraMD detects the initial infection and predict the next attack to be launched in the near future based on past experience or historical information. Then, the GraMD immediately monitors additional APIs for ‘t’ additional time period as the attack is expected to be launched.

Game Strategies

Let (AS,CA,AR,T), where AS is the set of APIs equipped with PAM which is referred to as defender, CA is the system cost for monitoring additional API, AR is the set of attackers and T = {1, 2,…n} is the set of target computers. To minimize the intermediate calculations, a two-player, non-cooperative game is selected in which the number of repetition depends on the number of the attacking steps. Ait is also assumed that both players are known about the strategies and utility function they have.

The possible strategies for GraMD are {no_attention, monitoring}. If the GraMD detects an API hook attack, it can select either to ignore the current task or to monitor. In case of monitor, the GraMD will select more additional important APIs to monitor. The type of the API to be monitored and the length of the monitoring time highly depends on the information base of GraMD. The monitoring time will be chosen from the information base, based on the attack scenario. When GraMD detects initial rootkit attack which will be carried out in multiple steps, it will be able to detect the next possible attack action. At the same time, the GraMD will choose to increase additional important APIs being monitored. After completing additional task, the GraMD resumes monitoring the standard predefined number of APIs.

On the other hand, the available strategies for the rootkit attacker i.e. MC are {end_process, proceed, waiting}. The strategy ‘end_process’ indicates to abandon the attack in order not to be detected; strategy ‘proceeds’ means proceeding with the predefined next step and strategy ‘wait’ indicates launching the next attack step after certain period of time. Since predicting the delay time for every attack action is difficult, the delay time to derive the approach is also considered.

Formulating the Game

The set of APIs to be monitored by GraMD is denoted as Ac = {a1, a2,…an}. The APIs are continuously monitored by GraMD as long as it is in live state. Let C_m denotes the additional computational system cost needed for monitoring single API at time '*t*', which is represented as: C_m=r × a, Where 'a' is a single API to be monitored and '*r*' is the number of clock pulse to monitor a single API and the time deviation is calculated using two local timestamps of the two events.

Now the total increased system resource cost α is computed in case when the GraMD choose 'monitor' strategy. This can be represented as:

α = Summation of drift in monitoring period of each additional API

= Time taken to monitor all protected APIs

$$= \sum_{a \in A} C_m(t_m) \tag{3.6}$$

Where t_m is the additional time to monitor single API. Then, converting these normal functions into utility function will be easy for us to apply game theory concept and it is given as:

$$R = f(\alpha(t_m, a)) = \gamma(\alpha(t_m, a)) \tag{3.7}$$

Where γ is the weight parameter to describe the system cost of GraMD. If GraMD detects an API hook attack, it will have utility gain of ω, and $-\omega$ is the cost of damage to be caused. Hence the utility function of GraMD with its possible outcomes is:

$$U_{GraMD} = \begin{cases} \omega - C_m & \text{detect and stop} \\ -\beta\omega & t_m \le t_d \\ \omega - C_m & t_m \ge t_d \ \ where\ t_d > 0 \\ -\beta\omega - C_m & t_m < t_d \ \ where\ t_m \ge 0 \end{cases} \tag{3.8}$$

Where t_m is the increased monitoring time because of the additional APIs to be monitored and t_d is the delay in launching the near future attack or current attack and β is false alarm rate of the GraMD. On the other side, when the attack is successful, the attacker will gain ω' and $-\omega'$ otherwise. Let C_a is defined as the security APIs consumed by the attacker to make the next step to be successful. This is represented as:

$$C_a(t_d) = r \times a \tag{3.9}$$

Where r is the number of clock pulse to monitor a single additional API and t_d is the delay time. Then, the cost-Utility function for the attacker is defined as below:

$$R' = f(C_a(t_d)) = \gamma'(C_a(t_d)) \tag{4}$$

Where γ' is the scale factor to define reward for a successful attack. Hence, the utility function of RM is given by:

$$\mathrm{U_{MC}} = \begin{cases} 0 & stop \\ \omega' & t_m \le t_d \\ \omega' - C_a & t_d \le t_m \ \ where \ t_m \ge 0 \\ -\omega' - C_a & t_d \le t_m \ \ where \ t_d > 0 \end{cases} \tag{4.1}$$

Table 3.3 shows the increased system resources for both the GraMD and the MC in general format. The S_i indicates different strategies to be selected by the GraMD in which the monitoring time and the API being monitor will vary.

Table 3.3 Utility Derivation

Players	**End_process**	**Proceed**	**delaytime variation**	
			$t_d \le t_m$	$t_d > t_m$
no_attention	$\omega, 0$	$-\beta\omega, -\omega'$	$\omega - C_{mi}, -\omega' - C_{ai}$	$-\beta\omega, \omega' - C_{ai}$
$S_i(t_d, a_i)$	$\omega - C_{mi}, 0$	$\omega - C_{mi}, -\omega'$	$\omega - C_{mi}, -\omega' - C_{ai}$	$-\beta\omega - C_{mi}, \omega' - C_{ai}$

Performance Evaluation

A case study is conducted to evaluate and simulate the game approach. Since delay time in launching near future attack will vary from attack to attack, let's set it from the historical data. A dataset of 100 different rootkit samples is collected which adopt both user-mode hook and analyzed them in windows XP virtual machine. By observation, most of the rootkit samples affect common native API functions to perform illegal activities in the victim computer. The GraMD's information base contains most commonly affected native API functions and their respective DLL file.

The evaluation describes the game after the initial attack action i.e., it is infecting all executables in the victim computer which is being detected by the GraMD. Now, the GraMD will take narrow action based on its intelligence about the impending attack following the infecting executables in the attack scenario i.e. from (no attention, increasing the additional APIs to be monitored by 50, and monitoring time duration by 400sec, 900sec, and 1500sec). Also, set γ=2, d=0.020, $-\omega$=3500, and $-\omega'$=2500. Table 3.4 shows the manipulated results for the first round.

Table 3.4 General IAT Hook

	end_process	Proceed	800	1200
no_attention	3500, 0	-3500, 2000	-3500, 900	-3500, 300
400	2700, 0	2700, -3000	-4300, 900	-4300, 300
900	1100, 0	1100, -3000	1100, -4100	-5900, 300
1500	500, 0	500, -3000	500, -4100	500, 4700

It is mentioned earlier that the players are rational and also the attacker knows defender strategies and will try to maximize its gain. The optimized result for P^x and P^y is calculated using Gambit tool. Table 3.5 and Table 3.6 contain the game result for Table 3.4.

Table 3.5 Optimal payoff for P^x

payoff	no_attention	20	30	50
300	½	4/35	0	27/70

Table 3.6 Optimal payoff for P^y

payoff	end_process	proceed	30	50
0	13/20	7/50	0	21/100

Finally, a payoff matrix is constructed for general malware hook detection with three different strategies as shown in Table 3.7. For simplicity only binary values are used to represent the output from rootkit detection module i.e. value 1 represents success and value 0 represents failure.

Table 3.7 General Rootkit detection payoff

	RegistryKey	IAT	Inline
No_attention	1,0	0,1	0,1
IAT	1,0	1,0	0,1
IAT, Inline	1,0	1,0	1,0

Simulation Result

For every scenario, a game theory approach is simulated to calculate payoff matrix. From the calculated payoff matrix, the Nash Equilibrium is generated using Gambit tool. Each scenario is simulated 10 times in our approach to calculate its corresponding utility resource. The average of every scenario is found and is plotted in a graph using MatLab software. In addition, the same 10 different samples are simulated in traditional approach and corresponding resource values are calculated to discover the difference. Figure 3.8 shows that GraMD approach takes less resource consumption than traditional approach.

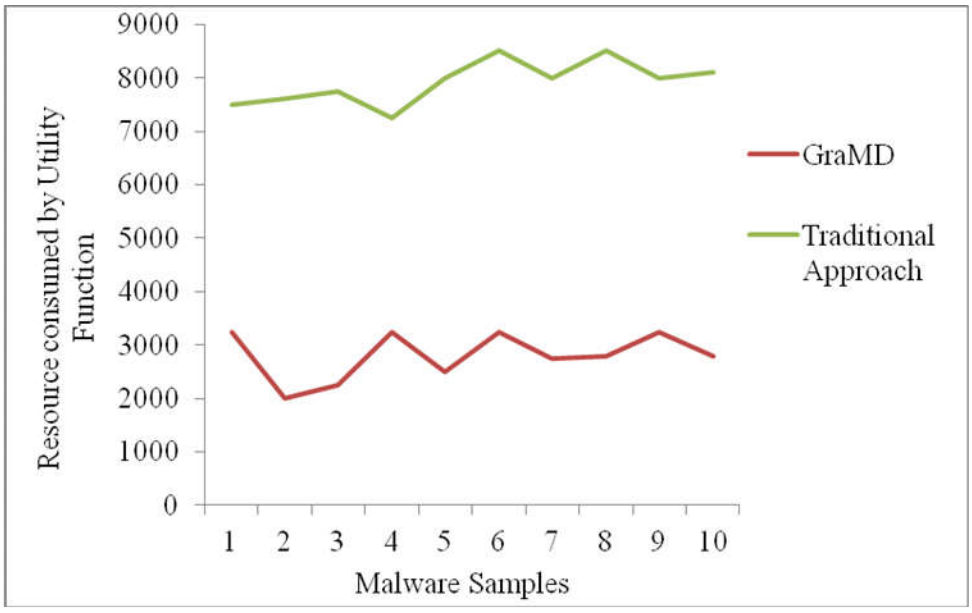

Figure 3.8 Resource Consumption

3.4.2 Analysis of the proposed GraMD approach

As GraMD divides an API call graph into simple paths and does comparison based on these paths instead of comparing the API call graph as a whole, it greatly reduces the complexity problem and also avoids scalability problem. In nature, graph matching algorithms belong to NP-complete problem and have a computational complexity due to slowness. Additionally, many such algorithms do not efficiently solve scalability problem. But all these issued are resolved in GraMD. Suppose the sum of the length of a path is '*n*' with size '*d*', then , the time complexity of GMA algorithm is $O(n \log p + d^2)$ which takes less time with reduced space requirement compare to existing graph based malware detection techniques.

3.5 Summary

Today, most malware authors have integrated API hooking technique with their code to evade detection. In this chapter, a graph based GraMD method has been presented to discover API hook attacks which are based on suspicious system call traces and the relationship between these calls. In turn, these system calls are represented as a call graph and then graph mapping technique is applied. Finally, the system determines the similarity rate using GED technique. The experimental evaluation results conducted on the malware samples prove that the proposed GraMD method incurs an average of 98-100 % detection rate which replicates a significant optimization over the existing methods.

Though a graph based approach can be effectively detect known malware activities but analyzing a hug volume of attack scenarios for a large network is a tedious process. In addition, graph based static approach failed to detect and prevent stealthy unknown malware. To overcome these limitations, a user-mode malware detection and prevention technique using dynamic malware analysis has been proposed and implemented which is described in Chapter 4.

CHAPTER 4

PROPOSED USER-MODE MALWARE DETECTION AND PREVENTION APPROACH

It is conspicuous from Chapter 2 that the graph based methods are generally preferred for detecting malwares based on the attack scenarios rather than preventing malicious attacks. Therefore, a behavior-based malware defense technique, UMDetect has been proposed for detection and prevention of malicious code attacks at user-mode. The architecture of the proposed UMDetect approach, test-bed and dataset used for evaluation, and comparison with the existing techniques are discussed in this chapter.

4.1. Preamble

Today, almost all modern malware writers have been incorporating hooking technique into their code either to compromise an end-system or evade its footprints from malware detection software and tools which make its detection more difficult. After the victim computer has been compromised, it can be later used for launching many illicit operations such as stealing important user information, sniffing network lines, sending spam, etc,. Therefore, detecting and preventing advanced API hook attacks is a challenging problem. Many existing solutions can be effective against API hook attacks but error prone and manual. Therefore, a behavior-based monitoring detection method namely, UMDetect has been proposed to effectively detect and prevent native API hooks in user-mode. Compare to existing malware identification techniques, UMDetect has been designed to dynamically monitoring and analyzing the behavior of API function call hooking attacks. The experimental evaluation results show that the proposed UMDetect approach produced better performance than the existing anti-hook detection tools and approaches with no false positive.

As discussed in Chapter 1, WOW64 subsystem permits executing 32-bit applications over 64-bit versions of Windows and thus malware writers are taking this advantage to exploit 32-bit versions of processor through 64-bit versions. Figure 4.1 shows the flow of execution of a user-mode application that invokes WriteFile() system service routine which is implemented in the kernel mode.

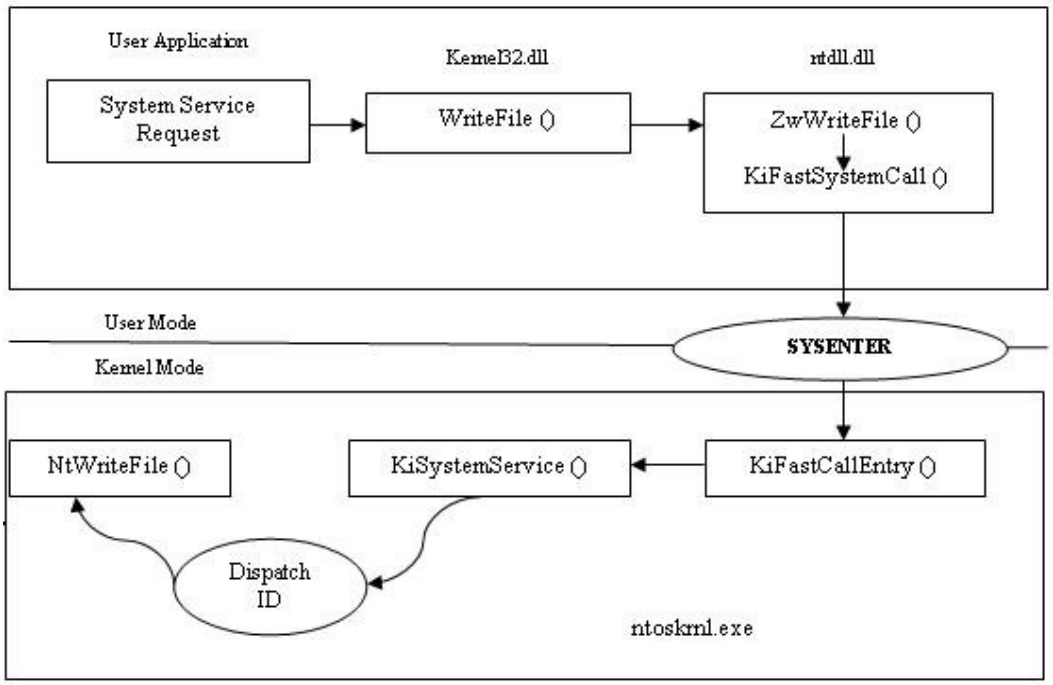

Figure 4.1 Execution flow of WriteFile() API function

The order of executing each step is explained below:

i. The user application calls the user mode WriteFile() function.

ii. The WriteFile() calls ZwWriteFile() native APIfunction which has a stub in ntdll.dll.

iii. Then, ZwWriteFile() calls KiFastSystemCall function which in turn executes the SYSENTER instruction.

iv. In response to SYSENTER, the program control is transferred to KiFastCallEntry() which is located in ntoskrnl.exe as executive service.

v. This will cause the KiSystemService dispatcher to call NtWriteFile() function using the dispatch ID.

Rootkits use several variations of hooking techniques during its lifetime. There have been many anti-rootkit detection tools are available for the purpose of detecting rootkit malwares. Each time such a tool is run, a log file is generated to keep a list of detected hooks. The amount of data in these log files is overwhelming as they hold information about each and every hook that had been detected on the system. On an average, each of these log file contains several hundred lines of code and data.

In order to perform malicious operations over the victim computer, malicious codes need to interact with the OS through Windows subsystem API libraries. The actual implementation of the native API functions resides in ntoskrnl.exe which is located in the kernel. Each native API has a reference inside ntdll.dll which is isolated in the user mode. After the malicious instructions are deposited in a victim computer, code-injection attacks must use native API calls to do further damage. Hooking various API functions into the victim computer is an important attacking technique employed by sophisticated malware. To defeat current hook detectors, modern malware writers maintain discovering new hooking mechanisms. However, the existing malware analysis technique is typically manual or error- prone. Therefore, UMDetect, a behavior based monitoring mechanism that does not require prior information about hooking method to defeat user-mode hook attacks has been proposed.

Import Address Table Hooking

The IAT is the most important call table in the user-mode which keeps references of all routines exported by a particular DLL files. IAT entries are filled by the Window automatically during load time. And each DLL that an application is linked with, particularly at load time, will have its own IAT. Many executable files have embedded one or more IATs in their structure that are used to store the addresses of existing libraries that they import from DLLs. Most of the user land rootkits use the IAT hooking technique to intercept the APIs at boot time. Thus, to maneuver an IAT, it is necessary to access the address space of the request. Normally rootkits use DLL injection techniques to modify the address of the specific function in the IAT to point to the address of the rootkit function where it is presented. Therefore, whenever the application calls a specific function, the rootkit function is called instead. Hooking a module's IATs using DLL injection can be accomplished by calling HookAPI () function as shown in Figure 4.2.

```
HookAPI (File *fptr, char* apiName)
{
DWORD bAddress;
bAddress = (DWORD) GetModuleHandle (NULL)
return (walkImportLists (fptr, bAddress, apiName))
}
```

Figure 4.2 HookAPI Function

Figure 4.3 shows IAT hook through CreateFile API function. The rootkits had managed to create a hook by overwriting the address of the CreateFile function in the IAT of the user application. If the entry point of CreateFile in the IAT is successfully modified with the address of rootkit routine, all native API calls in the target process are rerouted to rootkit routine.

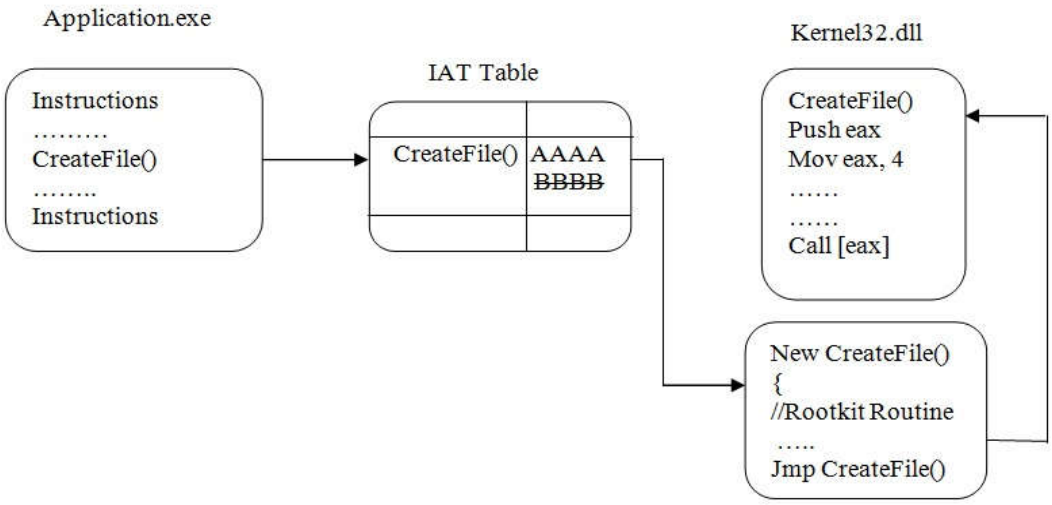

Figure 4.3 IAT Hook by a Malicious Rootkit

The walkImportLists() function checks the module's PE signature by adding a Relative Virtual Address (RVA) to the base address. Then checking each import descriptor will list all routines that are imported from the corresponding DLL. If Import Lookup Table (ILT) and IAT contain entries, then the names in the descriptor's IAT are compared against the name of the function that needs to be restored. If there is a match, substitute the address of the hooked function.

Inline hooking

Inline hooking or detour patching is another technique to divert the predefined execution path to malicious code without altering IAT call table entries. This technique is implemented by inserting a JUMP statement into the target routine to divert the execution path. Therefore, whenever the currently executing thread executes this jump instruction, the control is transferred to a detour routine. The original portion of the code from the target function which needs to be redeposited, in coincidence with the jump instruction returns back to the target code, is known as 'trampoline'.

Therefore, the initial jump in the trampoline replaces a certain code when it is inserted and at the end the necessary instructions might be executed which were replaced and then bounce back to the target code as shown in Figure 4.4.

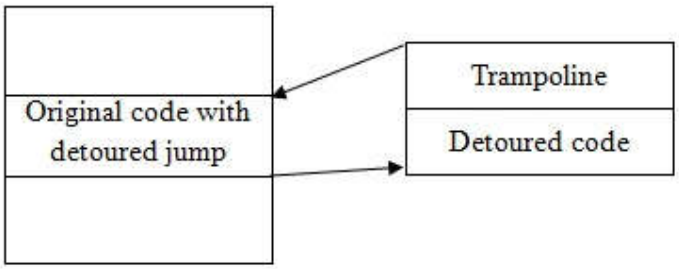

Figure 4.4 Hooking Inline Function

4.2 Architecture of the proposed UMDetect

To monitor and detect native API hooking in the user space, UMDetect intercepts native API calls in user mode and looking the traces of IAT entry modification and inline hooking. The proposed UMDetect approach is supposed to be installed in a clean system and it seizes native API system calls in user mode before they get service from the kernel of the OS. Figure 4.5 shows the proposed architecture of the UMDetect system.

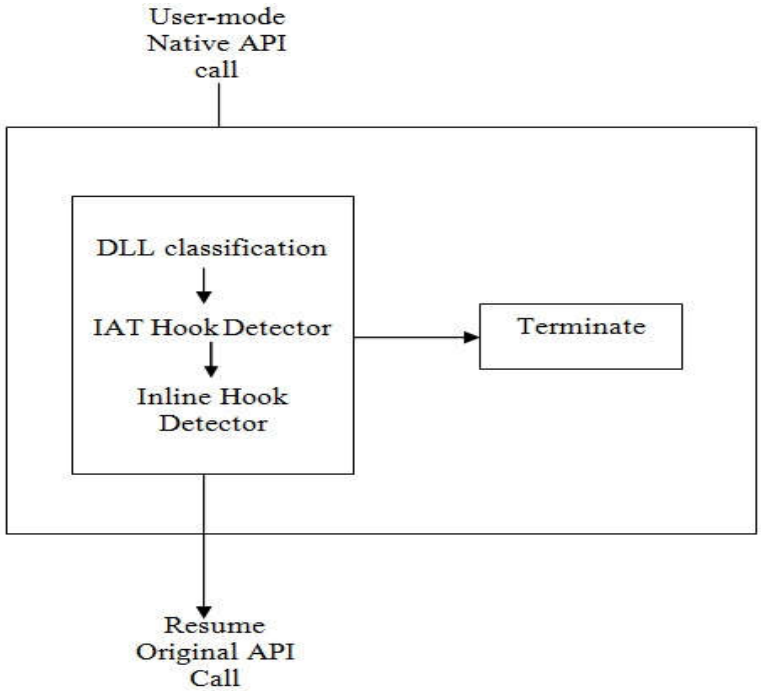

Figure 4.5 Overall flow of the proposed UMDetect approach

DLL classification Algorithm

The proposed UMDetect approach consists of three important modules: DLL classification, IAT hook detector and inline hook detector. Intercepting every system service calls that use native API is a tedious and time consuming process. Therefore, in order to allow legitimate system service calls to be serviced as normal, a new algorithm namely, DLL Classification Algorithm (DCA) has been developed. The pseudo code of the DCA algorithm is given in Figure 4.6.

```
/* Algorithm for DLL Classification*/

1.    begin
2.      Get DLL Name and handle using GetFileVersionInfo()
3.       if ((szDllName, dwHandle, dwCount, pBuffer) != 0)
4.          {
5.           Extract DLL file information and Store it in a file;
6.          }
7.       else use  VerQueryValue()  extract  VarFileInfo  and ValueLen;
8.        if (bVer && dwValueLen != 0)
9.         {
10.        Store DLL file informations;
11.       }
12.      if (extracred DLL informations are valid)
13.       {
14.         legitimate DLL;
15.       }
16.     malicious DLL and terminate application;
17.   end
```

Figure 4.6 Pseudo code for DCA Algorithm

Since most of the malicious code cannot include properties such as vendor name, description and version details in its DLL file, the DCA algorithm verifies these information to check whether the processes associated with the DLL file is malicious or not.

To get the vendor name of the DLL file, the GetFileVersionInfo() API function is called to get the file version information buffer which contains all the property values of a DLL file. To get the specified property value of the DLL file, the VerQueryValue() function is invoked. Finally VerQueryValue() function identifies whether the DLL file to be imported into IAT is either legitimate or suspicious.

IAT Hook Detector

The IAT Hook Detector (IHD) is the first level of defense module against native API hook in the user-mode. All processes that are associated with the suspicious DLL file are are given as input to this module. To detect native API hooks in IAT, the IHD performs the following steps:

a. The IHD obtains a list of currently running processes by calling the EnumProcesses function.
b. For each process, the PrintProcessNameAndID function is called by passing it to the process identifiers which in turn call functions OpenProcess to get the process handle, EnumProcessModules to extract the module handles and GetModuleBaseName to find the name of the executable file along with process id.
c. Then IHD compares each process with unknown processes. If legitimate, the LoadLibrary function is invoked to load the process into memory. After reading the MS-DOS header (MZ), PE, PE extended and section header from the executable, IHD determines DLL of an application which has been loaded and also the address range of each DLL in memory. Then IHD examines the IAT of the executable to examine the entries in each IAT.
d. Finally, if any entry drops outside of the module's address collection, the IHD stop executing the DLL; otherwise it will be serviced as a legitimate system service call.

Inline Hook Detector

As an alternative approach to IAT hooking, many malware writers keep call table entries within the requested range and instead modify the code that it points to. Inline Hook Detector (LHC) is another level of defense to strengthen the proposed UMDetect approach. First, LHC reads the executable file to reach the Export Address Table (EAT).

And then LHC calls the ExportAddrTable API function to read the addresses of each function in EAT which are responsible for invoking native API functions. To detect the detour patches of each function in the DLL file, LHC uses the CheckForOutside function to trap 0xE9 or 0xEA which will be opcode for the unconditional near and far jump respectively, in the first five bytes of the DLL's API function. Finally LHC identified the address where the CPU will jump to the function and then checks the CPU jump address to determine whether it lies outside the pre-allocated address range for the DLL file. If any function which are not in the predetermined address range is considered to be malicious.

Table 4.1 Malware Family with Hook Type

Malware family name	**Type of Hook**
Papras	IAT
Bacalid/DetNat	IAT
Haxdoor-B	INLINE
Agent	IAT
Feebs-A	INLINE
Qukart	INLINE
Virut	INLINE
Alman-B	IAT
ProAgent	INLINE
Alman-A	IAT

4.3 Experimental Test bed

A sandboxed experimental environment setup was arranged with a computer that runs Microsoft Windows 7 SP3 OS, Core 2 Duo 2.93 GHz of processor and 2 GB of memory and the guest machine runs Windows XP OS. In order to evaluate the effectiveness and performance of the proposed UMDetect approach, a set of 20 user-mode malware samples (two samples from each family) has been collected from some reputed websites, in addition to 10 benign samples. Table 4.1 lists the test samples which are taken for evaluating UMDetect.

4.4 Experimental Results and Discussions

Each malware sample was run in a controlled environment to detect the hooks generated by them in the victim machine. Few malware executable does not hook library functions when it starts running, but, as time increases, the number of hooks also increased. Figure 4.7 shows the result of the total number of hooks generated by each malware sample.

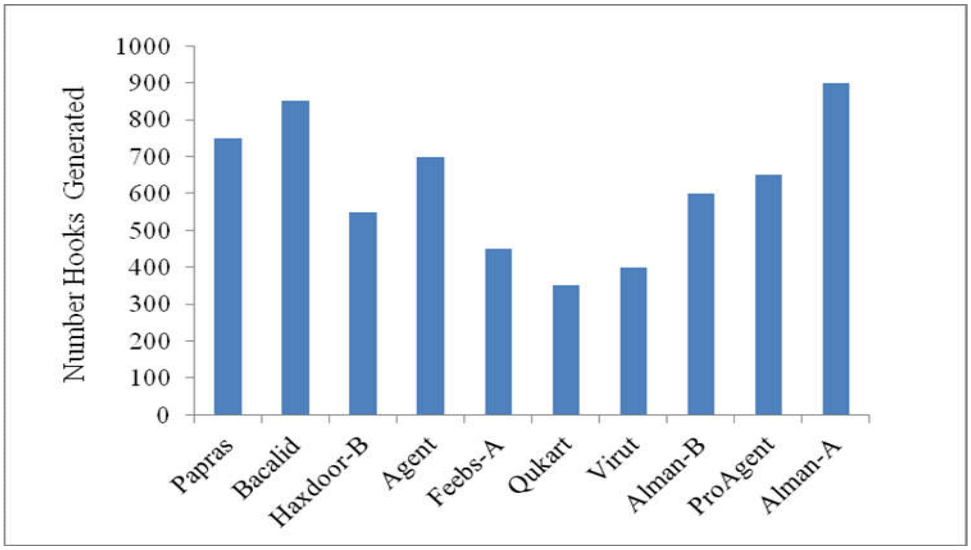

Figure 4.7 Number of hooks generated by different malwares

To show the precision rate of the proposed UMDetect approach, each malware was run separately against few standard hook detection tools such as BlackLight [131], IceSword [132], VICE [133], R3 Hook scanner [134], and UMDetect. Figure 4.8 shows the values of DR and FPR of various testing tools including UMDetect. Figure 4.8 indicated that the proposed UMDetect method outperforms than other existing with 95 % DR. This is because, few malware behaves liked legitimate. When the testing platform reboots again UMDetect detects those malware behavior correctly. Therefore the DR of UMDetect falls between 95-100 %. In order to determine FPR of UMDetect, 10 legitimate API hooks are implemented as a C++ file in Microsoft Visual studio. Every time a C++ file is executed, the proposed UMDetect method identifies all with no false positives.

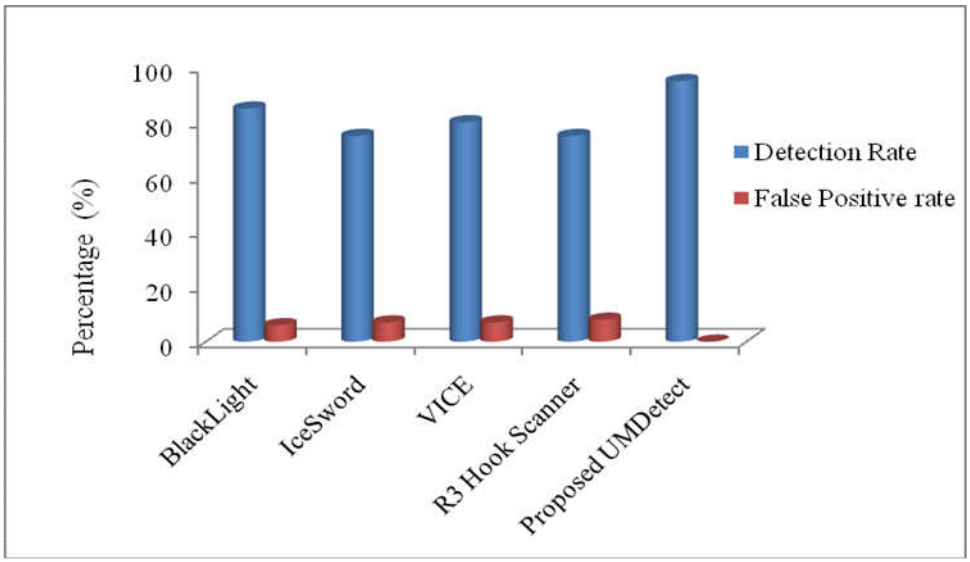

Figure 4.8 Comparison between Detection Rate and False Positive Rate

However, IceSword and R3 Hook Scanner have achieved the same DR of 75 % with different false positives. But, both IceSword and VICE has achieved 7 % of FPR. In addition the CPU cycles taken by various existing techniques such as the methods proposed by Deng et al., Abonghadareh et al., Yoghi et al., and the proposed UMDetect have been determined to estimate the overhead to be caused. Figure 4.9 points out that the proposed UMDetect approach has caused minimum overhead (2359 CPU cycles before executing UMDetect and 3659 CPU cycles after executing UMDetect) than the existing approaches.

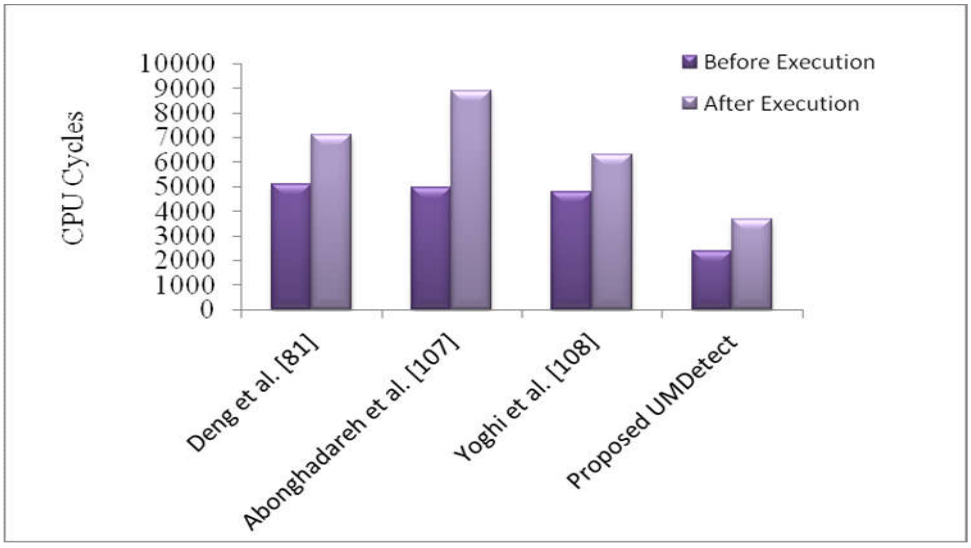

Figure 4.9 Performance comparison with the existing Approaches

4.5 Summary

With an increasing amount of malware adopting rootkit techniques to evade AV software, further research into defenses against malware that integrate rootkit technique to evade its detection is absolutely essential. A behavior-based method to trace and prevent user-mode malware that target hooking user-mode data structures has been implemented on Windows. There are few existing works that aimed to detect and prevent user-mode malware has been found in the literature. In addition, several user-mode hook detection tools are also available which can be used to detect these types of hooks. The proposed UMDetect has been evaluated using a dataset that consists of real-time malware that aims to misuse user-mode data structures on Windows. The experimental evaluation results indicated that the proposed UMDetect approach incurs 5% better detection rate than existing techniques.

The proposed malicious code detection and prevention approach surpass the existing approaches concerning the prevention of malicious code attacks in user-mode. However, UMDetect failed to identify and prevent the foot prints of malicious rootkit malware. Therefore, the task of discovering the hidden foot prints of malicious rootkit malware can really improve and optimize the detection rate and performance of the entire malware detection system which is resolved in Chapter 5.

CHAPTER 5

PROPOSED HIDDEN PROCESS AND SERVICES DETECTION ALGORITHM

It is manifest from the literature that the hidden process detection algorithms are generally preferred for optimizing the malware detection system. A new algorithm namely, Concealed Processes and services Discovery Algorithm (CoPDA) has been proposed and developed for discovering the hidden entries of malicious rootkit malware. In the subsequent sections a brief preamble about the hidden process detection algorithm is given and then the proposed CoPDA algorithm is validated by conducting various experiments using a real time dataset. The true positives, false positives, precision rate, detection through hindrance, and detection rate of the algorithm are derived and presented. The effectiveness of the algorithm is evaluated by comparing it to the widely used anti-rootkit detection tools and existing hidden process detection algorithms using various performance metrics.

5.1 Preamble

There are many forms of malicious software that can constantly affect a computer. Today, more advanced malicious software is incorporated with rootkit techniques to make detection more difficult. A rootkit is a technique which is designed with the intent of allowing the remote attacker to maintain highest privilege over the resources in the victim operating system. It has been in the wild for more than few decades. Different malware adopts different masquerading methods to avoid its detection. As a result, rootkits can dynamically defy detection either by hiding from the view of AV software. Because of these characteristics, rootkits are potentially dangerous to the integrity of user data.

Rootkits can be used for either legitimate purpose, such as debugging or malicious purpose when combined with malicious software. There are two basic classes of rootkit which are classified based on the mode of operation, such as user-mode rootkits and kernel-mode rootkits. As the latter operates in the Windows kernel, they are more powerful than the former.

In order to execute different pre-coded tasks, malicious software needs to perform some initial operations such as enumerating processes and services, opening a port, or establishing a network connection on the victim computer. A malicious rootkit can use either user-mode API hooking or kernel-mode API hooks in order to remain hidden. Table 5.1 lists some important API function names which are targeted by malicious rootkit families to execute their operations.

Table 5.1 API functions hooked by malicious rootkits

Rootkit Malware	Hooked Functions
Zbot	ZwCreateThread
Win32/Cutwails	ZwOpenKey, ZwEnumerateKey ZwSetValueKey
WinNT/Omexo	ZwReadFile
Win32/Ursnif	CreateProcessA CreateProcessW
WinNT/Ramnit.gen!A	ZwOpenKey ZwCreateKey
Win32/Dorkbot	CreateFileA/W CopyFileA/W ZwEnumerateValueKey
Win32/Eyestye	ZwEnmerateValueKey

There are many different techniques [26] [135] which have already been proposed to reveal a rootkit footprint.

i. Signature based – This approach uses unique signature, i.e., the byte sequence of known rookits. Additionally, heuristics and behavior models based on certain actions are used to identify a certain family of rootkits.

ii. Detecting detours – Windows operating systems are comprised of many important data structures such as Import Address Table (IAT), Export Address Table (EAT), and Interrupt Descriptor Table (IDT) in user-space and System Service Descriptor Table (SSDT) in kernel-space which allows a programmer to carry out task execution. Rootkits can either modify or alter these data structures to execute their own code. As a result, when there is a request for system related activities, it will always be executed by the detoured rootkit code.

iii. Crosscheck approach – In order to decept rootkit presence, they may alter particular data when returned to the defender. Therefore, one of the common approaches to detecting rootkit traces is by comparing the results returned from a high-level system call and low-level system call.

iv. Integrity check – A digital signature is created using a cryptographic hash function when the system is installed with a clean operating system. Then, each library call is checked for code alternation.

v. Alternative trusted medium - A rootkit can dynamically conceal its existence only when it is running. Therefore, the finest trustworthy technique for kernel-level rootkit revealing is to shutdown the infected computer and then checks its storage space by booting from an alternative trusted medium.

vi. Memory dumps – In order to capture a live rootkit, this method analyzes the virtual memory of the underlying operating system in offline state using a debugging tool.

All the detection methods listed above implemented different techniques with the intention to assist the defenders in ascertaining rootkit footprints. These techniques range from identifying for unique signature pattern in the impending malware sample to supervising system behavior. The important issue with live analysis is the authentic information such as files and functions returned by the OS. Most existing anti-rootkit detection tools crosscheck information generated by tainted system calls against system information generated by its own for identifying rootkit traces. A stealthy malware conceals its footprints by controlling OS function calls which cannot be hidden. But, using offline investigation to reveal hidden traces of a malware is very difficult.

In short, most existing techniques suffer from issues such as lack of integration, high false positive rate, overhead produced by complex configurations and scalability and performance issues. Therefore, to overcome the drawbacks of the existing algorithms, a new hidden processes and services detection algorithm is developed and presented in the subsequent sections.

5.2 Concealed Processes and services Discovery Algorithm (CoPDA)

From the survey, it is identified that almost all malware hide their footprints such as processes and services including the entries in registry, files, etc., to stay undetected and perform their illegal activities for a longer time. Therefore, focusing on locating these footprints to stop them before causing damage is important. This crucial point is mainly used to develop CoPDA effectively. As many data guard programs might use techniques such as encryption or access control to hide files locally, these issues are not considered. Rootkits may hide their traces and activities either from the user or AV. This vital point is mainly used to determine whether the underlying operating system includes hidden rootkit traces or not. The hidden character of a rootkit can be formalized as follows.

Let $G_a(t)$ be the set of system-wide objects of type a at time t, and $V_a(t)$ be the set of visible objects of type a at time t. Let o be an object of type a. At time t, if thereexist o such that $o \in G_a(t) \wedge o \notin V_a(t)$, then o is caught as hidden at time t. Otherwise, if $(G_a(t) \wedge V_a(t) = \phi)$ then the underlying system is in a stable state (with respect to a at time t). Let A represent all the available objects in the underlying system. Then,

$G = \bigcup_{a \in A} G_a(t)$ and $V = \bigcup_{a \in A} V_a(t)$ where G and V are the global view and visible view of objects in the underlying system, respectively. If any object is in the global view, but not in the visible view, then it is caught as a hidden object. As a result, by comparing G and V, the list hidden objects are discovered in the target computer. The proposed algorithm, CoPDA, is a practical tool that point out the user-level hidden rootkit processes and services. In order to produce hard evidence about the hidden processes and hidden services of a malicious rootkit, it is mandatory to have a global view and visible view of objects in the core system.

CoPDA is a crosscheck-based approach to discover and listing hidden processes and services by comparing the global view and visible view. One of the methods user-mode rootkits can use to hide their detection is to hook a native API function and filter its traces. A Windows process will open many handles associated with a process which can be used to detect hidden processes. One way to enumerate handles is to scroll through the process handles of CSRSS.EXE which holds the handles to all running processes. A list of the global view (G) is generated which contains the handles to all running processes by going through the process handle of the CSRSS.EXE process of the physical operating system. As rootkits hook normal process enumeration functions, the finest way to identify all processes is to make use of the NtQuerySystemInformation function. The pseudo code of the global view of the CoPDA algorithm is depicted in Figure 5.1.

```
/* Algorithm for GlobalView */
1.  begin
2.        Allocate necessary memory to handle table
3.        Loop through the range of valid handles
4.       begin
5.          Pid←handles.UniqueProcessId
6.          if Pid belongs to CSRSS.EXE
7.          begin
8.            Get Current ProcessId using OpenProcess and GetCurrentProcessId
9.          end
10.      end
11.      repeat loop
12.     begin
13.       Enumerate all Processes using NtQuerySystemInformation function
14.       Separate all child processes created by Services.exe
15.     end
16. end
```

Figure 5.1 Global View of CoPDA algorithm

Furthermore, in Windows operating system, services can only run with highest privilege mode. As a result, rootkits integrate services not only to keep control over the entire system, but also to manage its tasks and to remain undiscovered. Since services are visible to the end user, it is important for a rootkit to conceal its services to prevent itself from being detected. CoPDA has the ability to locate all hidden services by extracting information from Services.exe which is an important process in Windows operating systems for revealing rootkit footprints.

In order to produce a list of the visible view (V), a list of all running processes is generated by taking a snapshot of currently running processes in the system using CreateToolhelp32Snapshot in the guest operating system with which is used to discover hidden rootkit processes and services. Then the child processes that are manipulated by Services.exe are separated. The pseudo code of the visible view of the CoPDA algorithm is depicted in Figure 5.2. Finally, the entries in the global view are compared against the entries in the visible view. Any difference in these two data lists reveal hidden traces of malware.

```
/* Algorithm for VisibleView */
1.  begin
2.      Take a snapshot of all processes in the system using Createhelp32Snapshot
3.      Set the size of ProcessEntry32 structure
4.      Loop through the range of valid process in Process32Entry
5.      begin
6.          Walk the snapshot of processes and retrieve the information about each process
            using OpenProcess, Process32First, and Process32Next functions
7.      end
8.      repeat loop
9.  end
```

Figure 5.2 Visible View of CoPDA algorithm

5.3 Experimental Results and Discussions

All the experiments are conducted on a physical machine built with an ACER Core Duo machine with 2.93 GHz processor, 2 GB of memory and Microsoft Windows 7 as the physical operating system. The guest operating system runs Microsoft Windows XP SP1 with no service pack. CoPDA has been implemented using the Microsoft Visual Studio 2008 development environment. Different anti-rootkit detectors are opted for in to verify the strengths of CoPDA algorithm using various metrics. Table 5.2 summarizes a description of the version and revealing techniques of various anti-rootkit detection tools including CoPDA which were used for the evaluation procedure.

Table 5.2 Characteristics of testing tools including CoPDA Algorithm

Sl. No	Tool Name	Version	Hooking	Cross-View
1	GMER [136]	1.0.15.15281	Yes	No
2	HeliosLite [137]	1.005	Yes	Yes
3	IceSword [132]	1.22en	Yes	Yes
4	Rootkit Unhooker [138]	3.8.388.480 SR2	Yes	Yes
5	HiddenFinder [139]	1.5.6	Yes	Yes
6	BlackLight [131]	2.2.1046	Yes	Yes
7	CoPDA	--	Yes	Yes

5.3.1 Performance Analysis of CoPDA

A dataset consist of 1000 malware samples are obtained that belong to different infecting technique such as virus, worms, Trojans, rootkits and backdoors from a publicly available database [130] [141-142]. Then a carefully read, analyze and record the technical details of each malware individually.

Finally, a dataset which consists of 100 malware samples that mainly utilized API hooking, registry hiding and process hiding technique has been selected for testing and training CoPDA algorithm. This study helped to design the CoPDA to react against unknown malware. In order to determine the effectiveness of CoPDA in detecting hidden rootkit footprints, mainly, hidden processes and services, different samples with cross validation technique is used. The dataset is divided into ten groups – a group on the average contains 10 malwares and 9-10 benign samples. Then the tools are trained on nine groups and the remaining one group is used for testing them which is shown in Table 5.3. The computed values of all the metrics considered are summarized in Table 5.4.

Table 5.4 Summary of computed values

Tool Name	Weighted Average (%)					
	TPR	**FPR**	**PR**	**Recall**	**F-Measure**	**DA**
Hellioslite	88.14	5.93	93.67	88.14	90.76	91.03
GMER	85.53	6.00	94.33	85.53	89.45	89.42
HiddenFinder	85.52	8.13	91.98	85.52	86.82	86.83
RootkitRevealer	0.0	0.0	0.0	0.0	0.0	0.0
IceSword	96.18	3.82	96.18	96.18	96.08	96.12
BlackLight	96.0	3.0	97.27	96.0	96.47	96.50
Rootkit Unhooker	84.53	7.55	91.85	84.53	87.95	88.50
CoPDA	100	1.82	98.09	100	98.99	99.02

Table 5.3 A brief statistical report of the considered evaluation parameters

Tool Name	Metrics	Group1	Group2	Group3	Group4	Group5	Group6	Group7	Group8	Group9	Group10	Avg.
HelliosLite	**TP**	9	8	9	8	8	8	9	9	9	8	8.5
	FP	1	0	0	1	0	1	0	1	1	1	0.6
	TN	9	9	10	8	9	9	8	10	10	9	9.1
	FN	2	1	0	1	0	2	1	2	1	2	1.2
	FAR	10.00	0.00	0.00	11.11	0.00	10.00	0.00	9.09	9.09	10.00	**5.93**
	PR	90.00	100.00	100.00	88.89	100.00	88.89	100.00	90.00	90.00	88.89	**93.67**
GMER	**TP**	9	10	9	9	9	9	10	9	9	9	9.2
	FP	1	0	1	1	0	1	0	1	0	1	0.6
	TN	9	10	9	9	8	9	10	9	9	9	9.1
	FN	1	2	2	3	1	1	2	2	1	1	1.6
	FAR	10	0	10	10	0	10	0	10	0	10	**6.00**
	PR	90	100	90	90.00	100.00	90	100	90	100	90.00	**94.00**
HiddenFinder	**TP**	9	9	8	9	9	9	10	9	9	8	8.9
	FP	1	1	0	1	1	1	1	0	1	1	0.8
	TN	9	8	9	9	9	10	8	9	9	9	8.9
	FN	2	2	3	2	2	1	2	2	1	2	1.9
	FAR	10	11.11	0	10	10	9.09	11.11	0	10	10	**8.13**
	PR	90.00	90.00	100.00	90.00	90.00	90.00	90.91	100.00	90.00	88.89	**91.98**
RootkitRevealer	**TP**	0	0	0	0	0	0	0	0	0	0	0
	FP	0	0	1	0	0	0	0	0	0	1	0.2
	TN	10	10	9	10	10	10	10	10	10	9	9.8
	FN	0	0	0	0	0	1	0	0	0	0	0.1
	FAR	0	0	10	0	0	0	0	0	0	10	**2.00**
	PR	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0.00**

IceSword	**TP**	9	10	9	10	9	9	10	10	10	10	9.6
	FP	0	1	1	0	1	0	0	0	1	0	0.4
	TN	10	9	10	10	10	9	10	10	9	10	9.7
	FN	1	0	1	0	0	0	0	1	1	0	0.4
	FAR	0	10	9.09	0	9.09	0	0	0	10	0	**3.82**
	PR	100	90.91	90	100	90	100	100	100	90.91	100	**96.18**
BlackLight	**TP**	10	10	9	9	10	10	10	9	9	10	9.6
	FP	0	1	0	0	1	0	1	0	0	0	0.3
	TN	10	9	10	10	9	10	9	10	10	10	9.7
	FN	0	0	1	1	0	0	0	1	1	0	0.4
	FAR	0	10	0	0	10	0	10	0	0	0	**3**
	PR	100	90.91	100	100	90.91	100	90.91	100	100	100	**97.27**
Rootkit Unhooker	**TP**	9	8	9	8	9	9	9	8	8	9	8.6
	FP	1	1	2	1	1	0	0	1	0	1	0.8
	TN	9	9	9	10	9	9	10	10	10	10	9.50
	FN	1	2	2	3	2	1	1	2	1	1	1.60
	FAR	10	10	18.18	9.09	10	0	0	9.09	0.00	9.09	**7.55**
	PR	90	88.89	81.82	88.89	90	100	100	88.89	100.00	90	**91.85**
Proposed CoPDA Algorithm	**TP**	10	9	10	10	9	10	10	10	9	10	9.7
	FP	0	1	0	0	0	0	1	0	0	0	0.2
	TN	10	10	10	10	9	10	10	10	10	9	9.8
	FN	0	0	0	0	0	0	0	0	0	0	0
	FAR	0	9.09	0	0	0	0	9.09	0	0	0	**1.82**
	PR	100	90.00	100	100	100	100	90.91	100	100	100	**98.09**
	DA	100	95	100	100	100	100	95.24	100	100	100	**99.02**

5.3.2 Overall Detection Accuracy of CoPDA

From the usability point of view, allowing CoPDA to generate a low false alarm rate is an important issue; otherwise CoPDA will not be competitor against existing anti-rootkit detection tools if it stopped legitimate applications frequently. CoPDA identifies malicious hidden rootkit process and service by generating and matching two dissimilar views. The creation of the visible view is manipulated in the user-space of the guest operating system. Then, the production of the global view and the matching of the two views are completed in the physical operating system. The performance metrics for determining the effectiveness of CoPDA are described below.

- The Precision Rate (PR) and Overall Detection Accuracy (DA) are calculated as follows.

$$PR = \frac{TP}{TP + FP} \tag{5.1}$$

$$DA = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.2}$$

- F-Measure – It denotes the measurement of a test's accurateness by adding recall value and precision value into a single measure of performance. Normally, the result closer to 100% is good. F-Measure is computed as follows.

$$F - Measure = 2 * \frac{PR * Recall}{PR + Recall} \tag{5.3}$$

Table 5.4 shows that all anti-rootkit detection tools achieved different detection accuracy rate, i.e. detecting hidden processes and services, with different FPR. It is realized that CoPDA achieved an average of 98.09% PR with 99.02% DA of locating hidden processes and services. As few benign applications imitate operations performed by malicious software, CoPDA produced 1.82% FPR. When restarting the system, CoPDA correctly identified all hidden processes and services with no false alarm. Overall, CoPDA had outperformed all other anti-rootkit detection tools in all measures. The feature supplied with Receiver Operating Characteristic (ROC) is commonly used to evaluate the performance of malware detection tools in computer security.

Figure 5.3 depicts the generated ROC curve of eight anti-rootkit detection tools including CoPDA. As the curve bypasses the upper-left region, CoPDA achieved 100% TPR, indicated that CoPDA offers the best result among the other anti-rootkit detection tools.

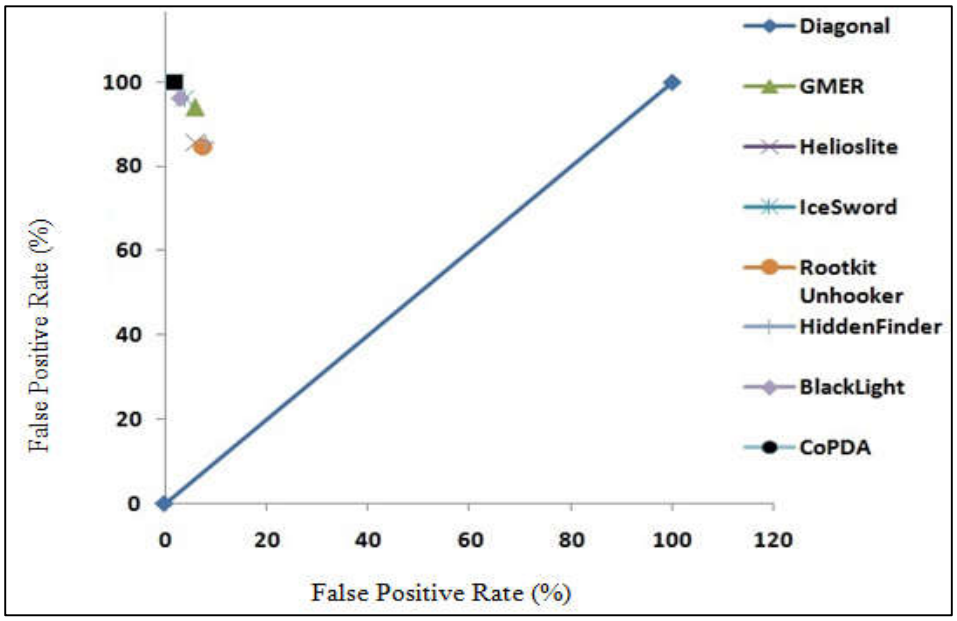

Figure 5.3 Realization of ROC Curve (Averaged Values)

From Figure 5.4 GMER scans for hidden processes, services, threads, files, as well as the detection of several different types of hooks. In our test, it had a precision rate of 94.33%. One important downfall of GMER is that it cannot allow a user to perform any other task while it is scanning the target systems. HeliosLite utilizes the cross-view approach to detect rootkits and can be executed from a USB drive. Against the rootkit samples that were taken for tests, it achieved a precision rate of 93.67%. IceSword scans the entire system to categorize hidden processes and services, files, registry settings, ports and startup items. IceSword spotted all the processes and services hidden of all the five rootkit families. It produced 100% precision as it detected all the hidden processes and services. However, while reacting to FU and FUTo rootkit malware, IceSword was unable to discover the rootkit which was responsible for hiding them. One important limitation of IceSword is that it only works effectively within the Windows XP OS, and fail to function in another environment. By scanning the entire target system, the BlackLight anti-rootkit tool identified hidden processes when executed against FU and FUTo rootkits. But fail to discover the rootkit. However, BlackLight achieved good results, detecting all the rootkits in our experiments.

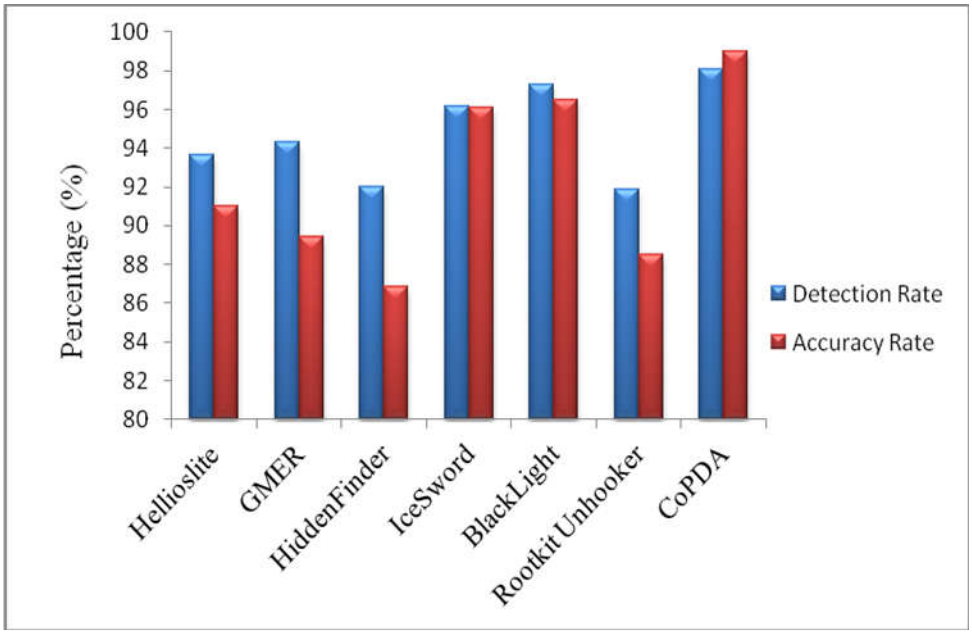

Figure 5.4 Comparisons of PR and DA

Rootkit Unhooker detects hidden processes, and files, but failed to discover hidden services and registry key values. As a result, it produced an 88.5% detection rate. HiddenFinder also uses cross-view technique to identify hidden processes, but only attained a 91.85% of the precision rate although it has not been revised since 2008. RootkitRevealer was the first rootkit detection application that used cross-view mechanism to detect persistent rootkits that can only hide files and registry-related settings in Windows operating systems although it does not detect hidden processes and services. Therefore, it has not been taken as a test tool.

CoPDA produces a total high detection precision rate of 98.09%. No other tested anti-rootkit detection tools achieved better TPR, FAR, PR and DA than CoPDA. Finding the mean for the other tools and compared with the proposed CoPDA produced higher values compared to the other tools. Mean and confidence interval for the ten set of samples are calculated against parameters such as true positive rate, false positive rate, recall, precision rate, F-measure, and detection accuracy which is given in the Table 5.5.

Table 5.5 Computation of mean and Confidence Interval

	Mean for 10 Groups(μ)	95 % CI
TPR	74.127	35.516 to 112.738
FAR	5.083	2.443 to 7.724
Recall	74.127	35.516 to 112.738
PR	78.548	38.100 to 118.994
F-Measure	76.1308	36.7545 to 104.876
DA	92.41	87.548 to 97.282

The comparison between the mean of both CoPDA and other tools are given in Table 5.6 which clearly shows that CoPDA outperforms than existing tools in terms of all features listed.

Table 5.6 Comparing CoPDA with existing tools

	TPR	FAR	Recall	PR	F-Measure	DA
Other Tools	74.13	5.08	74.13	78.55	76.13	92.41
CoPDA	100	1.82	100	98.09	98.99	99.02

Similarly, ROC curve is plotted using precision rate and detection accuracy to confirm the effectiveness of CoPDA. To achieve the same, the values between 0 to 1.0 are used. The calculated values are given in the Table 5.7. For the tabulated values, a graph is plotted using TPR and FPR as shown in the Figure 5.5 to compare CoPDA with other tools. The linear grid in the graph confirms that CoPDA achieves higher performance rate over the other tools.

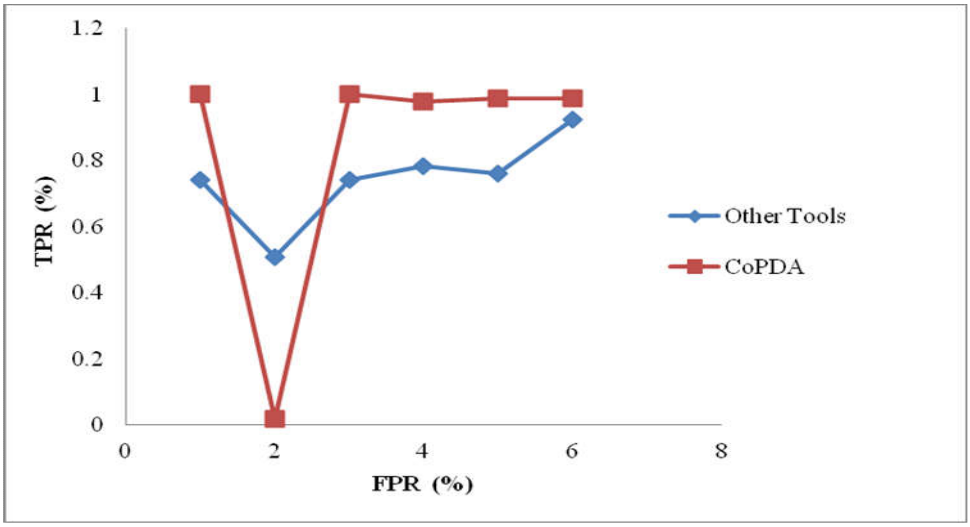

Figure 5.5 Comparisons of TPR and FPR

Table 5.7 Comparisons of CoPDA with existing tools (For ROC Plot between 0 to 1.0)

	TPR	FAR	Recall	PR	F-Measure	DA
Other Tools	0.74	0.51	0.74	0.78	0.76	0.92
CoPDA	1	0.0182	1	0.98	0.99	0.99

5.3.3 Detection through Hindrance

The timeliness and precision of CoPDA is evaluated when spotting a single hidden process. When the victim computer contained more than one hidden process, the discrepancy between the global view list and visible view list is larger which leads to much easier detection. Therefore, enabling CoPDA to detect a single hidden process implies an unconditional detection state. The synthetic process creator which is a tool for procreating processes randomly is utilized to determine the variation in detection against different amount of process load. It is shown in Figure 5.6.

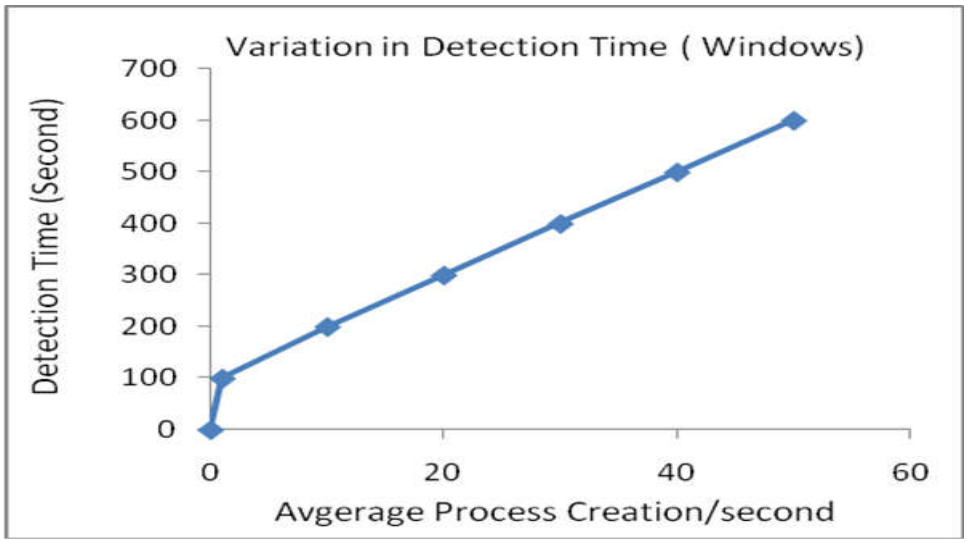

Figure 5.6 Variation in hidden process detection time depends on the process creation activity in Windows

In [140], the authors point out that process appearances are exploding than obsession. Hence, we select a Pareto distribution with k=1 (shape metric) for process inter-landing time. By varying the Pareto locality metric, the average rate of process creation is stabilized which is directed to sizable process creation. This dissemination makes the detection process difficult. The process lifetime interval is set between 0 and 1 second based on uniform process distribution to keep the test system in a stable state.

To keep hiding processes in Windows, the rootkit malware fu.exe hides its traces by directly modify a process token used by the OS. In addition, a guest process reporting tool is opted to fabricate hidden processes in Linux operating systems. The process creation time for different sets of process activity levels in Windows and Linux operating system are measured, the same is given in Figure 5.7 where the X-axis represents different process activity levels and the Y-axis represents the detection time. Figure 5.7 also states that each process creation and depart action increases the detection time. When the legitimate process consignment is reasonable, the presence of single malicious rootkit process is spotted every time. However, there is a bigger discrepancy in detection time under intense process creation that introduces false positives. However, CoPDA reduces the result of false positive identification.

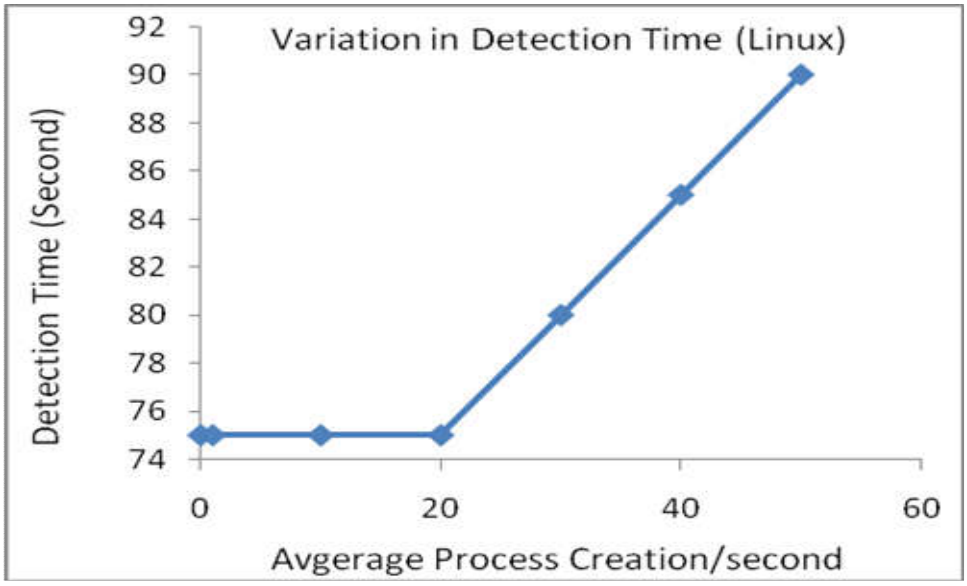

Figure 5.7 Variation in hidden process detection time depends on the process creation activity in Linux.

5.3.4 Runtime Overhead of CoPDA

While CoPDA runs continuously for detecting hidden processes, it is important to ensure that it produces significant runtime overhead. In order to assess the runtime overhead CoPDA algorithm, there are three benchmarks are executed in Windows and in the Xen hypervisor including CoPDA. After executing all five test samples and its average value is tabulated in Table 5.8.

Table 5.8 Detection of runtime overhead

Name of the Benchmark	Runtime in seconds		% of overhead
	CoPDA	Xen	
Compile	21.164	21.041	0.6
CreateProc	5.601	5.303	5.1
MemAlloc	5.523	5.220	3.5

The CreateProc is capable of creating and destroying 1000 processes rapidly. The MemAlloc allots a 200 MB fragment of memory and performs page table switching. CoPDA generates a 5.1 % runtime overhead for CreateProc benchmark and a 3.5 % runtime overhead for MemAlloc. For the Compile benchmark which includes the bash shell source, CoPDA causes a tiny runtime overhead of 0.6%. The CoPDA was tested and tweaked principally around 32-bit Windows and produced positive outcomes.

The proposed hidden process detection algorithm is compared with the existing algorithms proposed by Desheng et al., Xie et al., Richer et al., based on detection accuracy and performance overhead. Table 5.9 indicated that both the algorithm proposed by Desheng et al. and CoPDA have achieved 100 % detection rate, but Desheng et al. has selected only two malware samples for testing. However, in all aspects the proposed CoPDA algorithm outperforms than other existing algorithms with 100 % detection accuracy and true positives.

Table 5.9 Overall comparison with existing Approaches

Approaches	Overhead (%)	Detection Accuracy (%)
Richer et al. [120]	9.5	98
Xie et al. [121]	2.5	96
Desheng et al. [122]	0	100
Proposed CoPDA	0.6	100

5.4 Summary

Cross-check based algorithms have been effectively applied to discover the hidden foot prints of a malicious executable. The CoPDA algorithm, a cross-check based approach has been developed and implemented which runs in the user-space in Windows to discover hidden processes and services of a malicious executable. CoPDA maintains a separate process list that contains all running processes and services by continuously monitoring the victim computer. Then, another list is generated by dynamically analyzing lower-level processes and services and then the two lists of data are cross-checked to discover hidden processes and services. The experimental results show that CoPDA outperforms the existing algorithms which are proposed by Desheng et al., Xie et al., and Richer et al. The CoPDA algorithm gives considerable improvement over the algorithm proposed by Richer et el., and surpasses the algorithms proposed by Desheng et al., and Xie et al.

CHAPTER 6

PROPOSED KERNEL LEVEL AUTHENTICATION MECHANISM

The trustworthiness of the underlying computing environment is very important to ensure total system security. The key issue in system security is verifying the originality of an application to check whether the application is legitimate or malicious before being serviced by the kernel of the OS. A novel kernel level process authentication mechanism has been proposed for imposing mandatory authentication to validate the originality of all suspicious processes of the executable rather than verifying all. PAM is evaluated using different test-beds and datasets and compared with the existing techniques concerning the prevention of unauthorized malicious process attacks before abusing system resources. In the subsequent sections, a brief preamble about process authentication to prevent malicious code attacks, the proposed system architecture of PAM mechanism and experimental results are presented in this chapter.

6.1. Preamble

Today, hackers frequently integrate rootkit techniques into their code to compromise and evade detection. A malicious rootkit is a software tool which is designed with the intention of acquiring and maintaining privileged access rights over the resources of the underlying OS while concealing its footprints by subverting legitimate OS behavior. Most computers have included an ACL feature to prevent an authorized application for obtaining access to confined resources. But rootkits might use the vulnerabilities in the target computer or use social engineering attacks to get mounted. After successfully installed, the rootkit does not want to be identified by an existing anti-rootkit tool so as to prevent its access. One of the best ways to bypass this is to become invisible to all running software applications on the target machine. Most software applications trust the OS to provide authentic information about its environment in which it is running. The application inquires the OS, for example, files and registry keys which are necessary for configuring the application, using the API provided by the Windows subsystem. Windows comprises of many sub-system DLLs to offer many different features to the programmers.

Among all, ntdll.dll is acting as an intermediate interface to the forward Native API calls to kernel-mode and is located in user-mode. The implementation of each native API calls is located in ntoskrnl.exe which resides in the kernel. Each native API call has a stub in ntdll.dll. Figure 6.1 shows how the NtCreateFile() native API function call is requested. The parameters for accessing a particular system service are kept in the stack in advance. The system service number and the address of kiFastCallEntry stub are pushed to EAX and EDX respectively. Then kiFastCallEntry stub moves its address to register ESP and starts executing SYSENTER in ntdll.dll.

```
NtCreateFile:
mov eax, 0x17
mov edx, kiFastCallEntry_add
call edx
kiFastCallEntry:
mov edx, esp
mov SYSENTER
```

Figure 6.1. NtCreateFile API function call

Then kiSystemService, the service dispatcher in the kernel-mode use the system service number which is stored in the register EAX to lookup in the System Service Dispatch Table (SSDT) to forward the request to the actual routine which service the incoming request. After completing the system service, the kernel switches to the user-mode using the return address which is stored in ESP. In order to hide its existence and stay for an extended period of time, rootkits hijack these undocumented APIs and observe for any request an application may arise that might be incriminating.

Many advanced stealthy malware re-route the flow of a system call by modifying their address in SSDT to point to the detoured code. Therefore, its detoured code runs whenever the system call is invoked. The OS then executes the code at that address. The combination of privileged rights and stealthy technique make malicious rootkits or codes a particularly serious threat. In order to protect few important processes, Microsoft had introduced protected process feature in Windows Vista. For example, in Windows Vista, Protected Media Path (PMP) utilizes protected process mechanism to offer a high level of protection for providing stronger protection of sensitive media content. Whenever a new process is started, the system performs different level of checks as given in Figure 6.2.

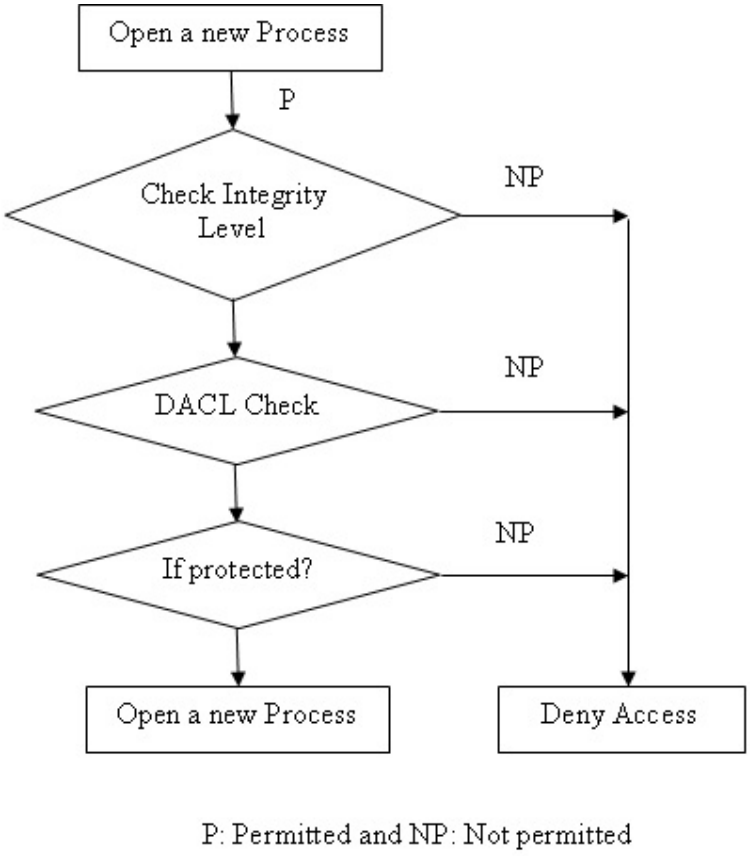

Figure 6.2 Various levels of checks of a protected process

The same validation checks are pertained for all threads that belong to protected processes. The protected process concept has evolved to strengthen the security and protect end-user. By default, only certain processes will be started as protected. Microsoft restricts third parties from accessing protected processes. Also, a protected process has unconstrained privilege access over other protected processes. However, a custom protected process, which integrates random code attack can bypass the complete system. The main purpose of introducing protected process feature is to provide an environment for protecting and enhance Digital Right Management (DRM) functionality in Windows Vista x64. But open privileged access to system process would weaken kernel level protection. Most Malware Detection System can either only detects either only detect attacks before its execution or after the victim computer has been compromised.

The first method has only limited information to detect malware attacks. On the other hand, the second method fails to prevent the damage to be caused which cannot guarantee for system security. Therefore, recent security systems need to be strengthened to ensure total system security or trusted computing environment. Developing a mechanism for improving the security strength of an OS is very essential, because the hackers target the most popular OS such as Windows rather than Linux environment. This research work exhibits the need for re-examining the system's fundamental process identification system. Therefore, the idea of kernel level runtime authentication has been developed to discover and prevent malicious code attacks that target hooking system services during runtime.

If any process or application fails to supply the kernel generated credential information at runtime, PAM labels it to be malicious and suspend or terminate their future activities. To be accurate, after detecting a suspicious process or application, PAM remarks it to be suspicious and validate its originality. PAM directly permits benign processes to get kernel service so as to improve the overall performance of the prototype. The innovation of PAM is that, it incorporates light-weight system call authentication technique to verify the authenticity of suspicious processes at runtime which is not often done by the OS kernel.

Additionally, PAM identifies the suspected processes and executable which can act as agents of remote hackers and restricts them. PAM is designed to be implemented on Windows and conduct various experiments to demonstrate its effectiveness and efficiency.

- PAM is a novel kernel level system call authentication mechanism which includes malicious code tracing and authenticating their originality to prevent malicious code attacks that directly invoking a system service call in the kernel mode on a marketable OS in a friendly manner.
- PAM has been designed and implemented on Windows OS to prevent process forging attacks at runtime without using their signature.
- The important reasons of incompatibility and low usability issues were investigated of existing anti-rootkit detection tools.

In short, the runtime mechanism, PAM, can enrich the security strength of current computing environment with the following properties:

- PAM cannot be compromised by malwares with kernel level privilege that target system call runtime hooking in contrast to the traditional security systems
- PAM, as a runtime authenticator framework, exposes malware attacks which execute data at user-mode, modifies code, and modify kernel mode data structures illegitimately.
- The dynamic malware prevention method of PAM implements uncircumventable more flexible better than that of current malware defense security systems.

Dynamic Detection and Prevention. PAM must prevent malicious executable at the moment when they illegally attempts to hook kernel level data structures during runtime.

Preventing Kernel tampering. Recent stealthy malwares attempt to access kernel level information and code to carry out its illicit activities on the victim computer. PAM must ensure the trustworthiness of kernel level data and code by restricting the behavior of each and every suspicious process. This access restriction helps to limit or completely control the range of malicious activities.

Tamper Resistent. PAM must function acceptably even when the malicious executable runs on the target computer and not offer any chance to the malware to subvert the preset protection mechanism.

Low performance overhead. The additional functionalities to be incorporated into kernel must guarantee the performance of the entire system.

Combat against recent stealthy malware attacks. PAM must detect and prevent unknown malware attacks and advanced stealthy malware that target misusing kernel level resources.

Target Malware

PAM monitors system wide process manipulation in user mode to discover and prevent foot prints of malicious executables. However the primary goal of PAM is to protect kernel data structure and the malware that attempt to hook system services at runtime.

PAM Memory Protection

- PAM disables write operation on user-mode helper components and the kernel.
- When a malware attempts to alter a kernel protected memory area, the processor identified the access breach and raises a page fault.
- PAM checks whether the address of the raised page fault is protected.
- PAM raises an alert and shuts down the computer, if it identified any access breaches that are associated with its components.

6.1.1 Assumptions

An attacker's target system is defined as a computer which permits malicious code exploiting vulnerable processes. The intention of malicious software is causing damage to the victim computer by invoking system services using either user-mode or kernel-mode API functions. It is assumed that the attacker cracks PAM if it holds any one of the following properties.

- Malicious code cannot call any system service using user-mode native API invocation. If so, it will be caught by the CoPDA Algorithm.
- Malicious code cannot be allowed to read or write the memory from user-mode stack.
- The malicious code will not be permitted to scan the code part of the legitimate executable. Otherwise, the OS triggers general protection error message.
- Malicious code is prohibited to amend read-only pages in memory. Violation of this property must call a native API which is disallowed.
- Malicious code cannot exchange their process ID and thread ID. Also, the shared memory concept of modern OS disallows to search other process's memory area.

In addition, it is assumed that the defender does not have any clue about the type of resource an attack may utilize to achieve its ultimate goal.

6.1.2 Security Models

(i) ***The design goals of PAM*** – The goal towards PAM design is to guarantee that the kernel of the OS appropriately authenticates each system service call to be raised by the application during runtime and malicious code cannot pretend to be legitimate process.

PAM design trusted elements are the Process Identifier (PID), the kernel credential information, the kernel loadable code, and the kernel's protected area of memory. It is assumed that the kernel of the OS does not include any malicious code, as it is hard to design any protected computer without this. The kernel memory security properties such as integrity and confidentiality are also preserved.

(ii) ***Malware Attack Type*** – Advanced stealthy malware code on the victim computer may run without user intervention as a user-level process. A remote attacker can inject malicious code into software and force them to abnormally execute the injected code. During code injection phase, the malware attempts to perform a certain operations at the user-space. The injected malicious code creates duplicate processes that are necessary for execution in the user-space. Then, the malicious code may pretend to be a legitimate process by spoofing process names. Hence, it is not possible to process names as unique identifier for differentiating running processes.

There are many different requirements to generate inimitable secret data for process authentication problem. Some of them are common that can be found in other credential scheme, whereas few are uncommon and new.

(i) ***Confidentiality*** – A secret Process Credential (SPC) s shall not be known to malicious user-mode processes.

(ii) ***Originality and reproduction*** – For each process, only one SPC must be supplied. The SPC is updated during re-execution of the same process. Additionally, valid SPC cannot be duplicated.

(iii) ***Anti-replay Attacks*** – As the authentication mechanism changes the credential for every system call invocation, reclaiming SPC for replay attack will be disallowed.

(iv) ***Runtime Supervising*** – Before processing each process request, the authentication mechanism needs to ensure that if this request has been already authenticated or not. This can be achieved by querying the information and status of that process which is stored in the kernel helper processes.

6.2 Proposed System Architecture of PAM

PAM authenticates each suspected process system calls, whereas all legitimate process will get kernel service directly. This actually improves the overall performance of the PAM. PAM retains kernel issued credentials and knowledge of each suspected process in the user-mode. This information is queried by the kernel during the authentication phase to determine whether to allow the system call or not. The PAM acts as a sandbox that can prevent malware from violating kernel's predefined security policies. PAM comprises of four important components, namely, the security monitor, the preservation function, the Credential Information Generation Function and the runtime authenticator is given in Figure 6.3.

6.2.1 Security Monitor

The Security Monitor (Mtr) component is responsible for monitoring process manipulation on a system-wide basis. To prevent the installation and execution of malicious programs, we control process creation in the user-mode on a system-wide basis by hooking NtCreateSection() function which cannot be easily bypassed by any process to launch a new process. Whenever a new process is created in the user-mode, it will be monitored by the Mtr and is tested by the CoPDA algortihm. Intercepting processes and verifying each and every incoming system service request during runtime is a tedious and time consuming task. Therefore, the CoPDA algorithm allows each legitimate system service call invocation to be serviced as normal and classifying the remaining as suspicious. This is achieved by comparing the currently running processes and services against the same information obtained kernel level.

After loading and initialization necessary information, the Mtr suspends the main thread of each suspected process. Next, the Mtr hooks each suspected process by inserting guidehook.dll into it by allocating necessary space using VirtualAllocEx function. Finally, the Mtr initialize the hooked dll by calling CreateRemoteThread and resume the suspended thread. Therefore, whenever the hooked is called, execution transfers to our detoured code, which is indirectly get executed, and after completion, the control is transferred back to allow the original function routine for completion.

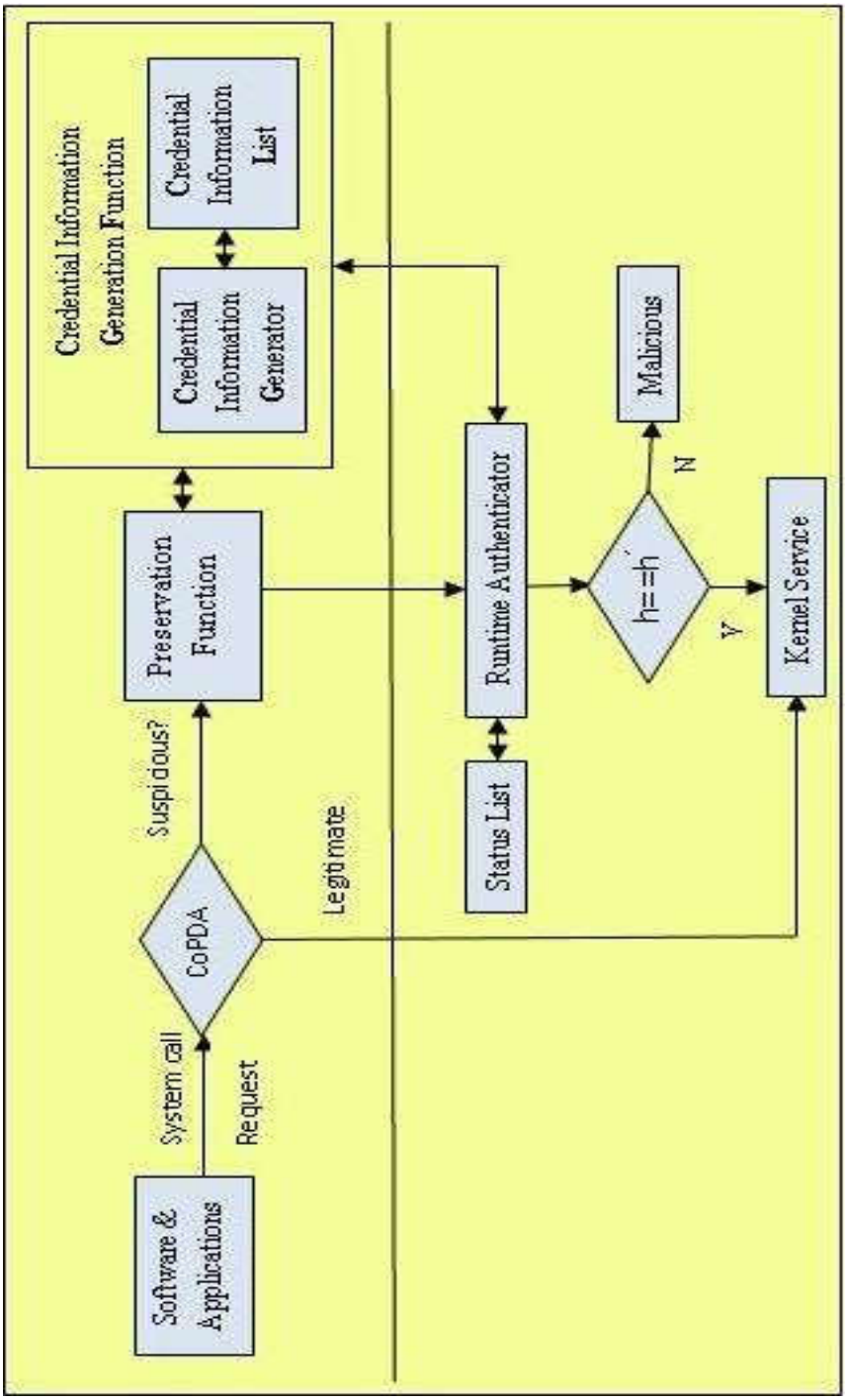

Figure 6.3 Overall flow structure **of** PAM

Operations of guidehook.dll. First, the memory is scanned to find the address of ntdll.dll which contains stubs of kernel API functions. In Windows, the ntdll.dll is the first module to be loaded, i.e., the first LDR_MODULE entry in InInitializationOrderModuleList. Since, the register EAX = PEB → Ldr. InInitializationOrderModuleList.FLink, then [EAX+0] ← List entry's FLink and [EAX+4] ← List entry's BLink. As a result, the base address value of ntdll.dll at [EAX+8] is obtained. Then, the ntdll.dll is hooked by inserting, the Preservation Function and Credential Information Generation Function into it by using WriteProcessMemory function. Then, the entry point of each native APIs is located by inspecting ntdll.dll and replaces sysenter command by jump preservation function.

Therefore, the detoured code will be executed first whenever a suspected process requests a system service. Finally, a read-only page is created in the memory by setting PAGE_READONLY protection flag of VirtualProtect function where the address of a process authentication function is stored at M. The description of notations and complex words is given in Table 6.1.

Table 6.1 Description of notations and complex words

Notation / word	Description
Mtr	Security Monitor
VirtualProtect	It changes the access protection of a process
VirtualAllocEx	This function initializes and allocates necessary memory
WriteProcessMemory	This function is used to write data to an area of memory in a specified process
LDR	It is a pointer to a PEB_LDR_DATA structure which actually contains information about the modules to be loaded for the process
EAX	Register
ESP	Stack pointer register
FLink & Blink	Forward Link and Backward Link
PEB	Process Environment Block

In addition, the diagrammatic representation of the security monitor and guidehook.dll is depicted in Figure 6.4 and Figure 6.5 repectively.

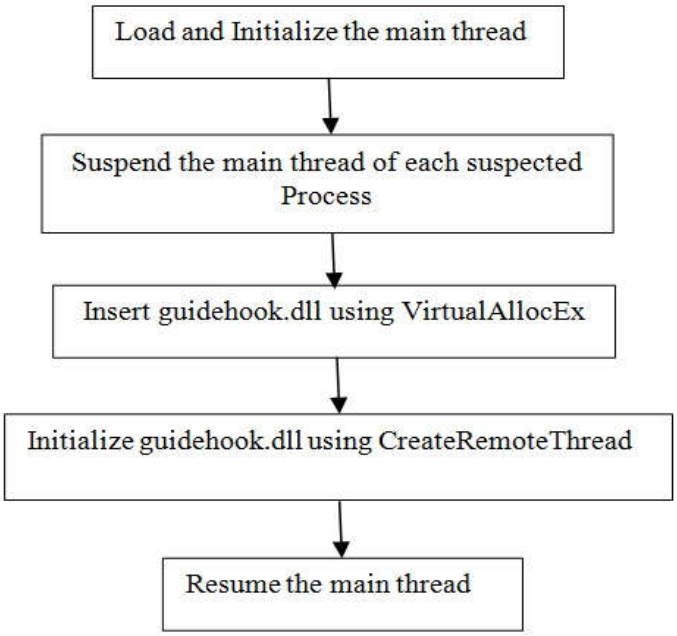

Figure 6.4 Diagrammatic representation of Security Monitor

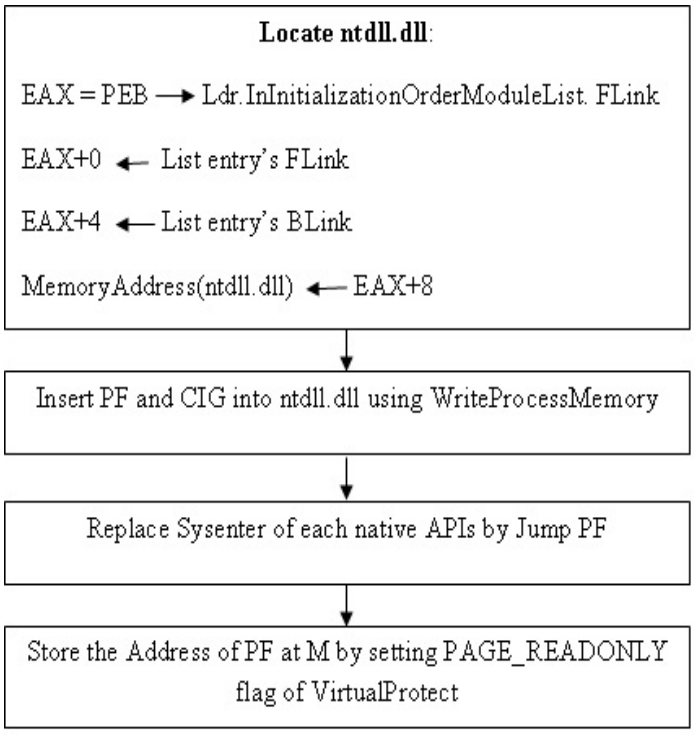

Figure 6.5 Diagrammatic representation of guidehook.dll

6.2.2 Preservation Function

The Preservation Function (PF) is one of the kernel helper processes which reside in the user-mode. To avoid data integrity problems while processing PF and Credential Information Generation Function, it is necessary to make a copy of their register values in advance. All these values are kept in the stack which will be restored later when the Credential Information Generation Function task is over. The EIP instruction pointer value is restored when the Runtime Authenticator queries the Credential Information Generation Function. To allow the Credential Information Generation Function to perform correctly, we backup the stack pointer ESP at memory address, E. If any malicious process/application tries to bypass this phase, it will fail to succeed in the authentication stage.

6.2.3 Kernel Level Runtime Authenticator

The Runtime Authenticator (RA) is the kernel-mode component which is the heart of our design. Its goal is to authenticate each suspected process at the kernel mode during runtime before being serviced. When a system service request enters into the kernel for the first time, the RA checks the Status list (S) which only maintains processes that have been successfully authenticated previously. If any request has not been authenticated, the RA queries the CGF by sending the query (Pid) to retrieve its HMAC value.

In response, the CGF reply with the h value of the corresponding Pid which is retrieved from CIL. If the returned h value is null or the delay associated with received h exceeds t value, the authentication check is disallowed. Otherwise, the R_A recomputes h'←hmac(Pid, p.srn) and compares both the values of h and h'. If matches, then the authentication check is successful. Otherwise p is concluded as malicious. Finally, information about serviced system call entries are removed from the list S and the same is reflected in the CIL.

6.2.4 Process Authentication Protocol

The process authentication protocol runs between user-mode and the kernel for ensuring secure communication. Let p represent a new user process with a unique Pid and p.srn represent the copy of p's secret information. We write hmac-req (p.Pid) for sending p's secret credential information retrieval request to R_A and a secure hash code generation function, HMAC.

1. For each suspected process, p, the CGF performs the following operations:
    a. Generates a cryptographically Strong Random Nonce (srn) with a time frame, t and forwards (p.Pid,p.srn) to R_A. The time frame will expire some (short) time afterward or if no response from p.
    b. Computes h←HMAC(p.Pid, p.srn) and stores (p.Pid, h) in CIL.
2. When p enters the kernel for the first time, R_A performs the following operations:
    a. R_A confirms whether p.Pid Є S. If so, it will be serviced by retrieving its Pid and dispatch identifier.
    b. R_A recalculates h'←HMAC(p.Pid, p.srn).
    c. R_A queries the CGF with p.Pid to send its h value. If there is such value is found in the CIL, then p is reported as malicious.
    d. If a delay in receiving the HMAC exceeds t, then the authentication request will not be processed.
    e. R_A compares h with h'. If matches, the authentication request succeeds. Otherwise, p is reported as malicious.
3. When p completes, all its corresponding entries in both S and CIL are about to be deleted.

The constraint of a process of an application in order for authentication protocol to succeed, it requires the knowledge of the kernel generated credential. For example, if a process which belongs to the Internet Explorer browser claims to be legitimate, then it must succeed the authentication phase by supplying kernel generated credentials. The process identifier which is used for generating credentials is maintained by the kernel and assumed to be unforgeable and trustworthy.

6.3 Experimental Results and Discussions

The effectiveness and performance of PAM is evaluated in using a dataset consists of 350 malware samples [129-130] [142-143] has been obtained based on its attacking techniques employed and execution environment. In addition, 50 benign samples are also obtained from two reputed websites [144]. The computation of performance overhead, true positives, and false positives of PAM are determined through conducting two test cases namely, *test case 1* and *test case 2*. *Test case 1* is conducted on a single workstation with a dataset consists of 50 malwares and *Test case 2* is conducted on a client-server model using a dataset consists of 300 malware samples.

Test case 1:

The experiment is conducted on a computer with 2.8 GHz Intel Pentium 4 processor, 4 GB of memory and running Windows 7 OS and performed a series of tests on Windows XP OS. This is because WOW64 intercepts all OS system calls made by a 32-bit application. There are two important reasons for the selection of Windows operating system for evaluating the proposed PAM. First, Windows is the most popularly used OS and malware creators ensure that their creations will work in all types of OS from Windows XP to mobile OS. Secondly, system calls and API functions of x32 bit applications will work on x64 bit OS without requiring additional settings. The Microsoft Windows Driver Development Kit [145] was used for implementing the kernel driver module of PAM.

In order to evaluate the efficiency of PAM, a standard micro benchmark namely, KeQueryPerformanceCounter is used on a work station to determine the additional time to be taken for executing the proposed PAM. The benchmark function returns the number of ticks per second (ts) i.e. additional time taken for executing PAM, using PerformanceFrequency function. The symbols used in *test case 1* are given in Table 6.2.

Table 6.2 Description of Symbols

Symbol	Description
ts	Number of ticks per second
T_1	First invocation time at which the KeQueryPerformanceCounter function is called.
T2	Second invocation time at which the KeQueryPerformanceCounter function is called.
Ti	i th invocation time at which the KeQueryPerformanceCounter function is called.
T	T_2-T_1
^{l}T	Overhead caused by KeQueryPerformanceCounter function
^{l}Tnapi	Overhead accounted by new native API
Tnapi	Execution time of a native API
t	Clock counter of KeQueryPerformanceCounter

Let Ti is the ratio of the clock ticks counter per second (ti) to the PerformanceFrequency and T represents the difference in execution time between the first invocation (T1) and second invocation (T2) of KeQueryPerformanceCounter. On the other hand, ^{l}T is the overhead incurred by KeQueryPerformanceCounter which is measured by two consecutive calls of the same function. Similarly, $^{l}Tnapi$ is the overhead caused by new native API which is computed by Equation 6.1.

$$^{l}Tnapi = (\ ^{l}T_2, napi - (\ ^{l}T1, napi) - \ ^{l}T \qquad (6.1)$$

There are four important legitimate native API functions are considered in which three of them namely, NtOpenFile, NtCreateFile, NtWriteFile and NtOpenProcess are critical and other one, NtClose, is non-critical. Each native API is executed separately and interrupted by the PAM and the same test is repeated 1000 number of times and obtained its average value. Table 6.3 depicts the statistical result of execution of *test case 1*. In order to estimate the impact of the PAM with respect to the core system, the execution time of the genuine native API (gA) and the overhead caused by the corresponding native API (oA) intercepted by PAM are computed and averaged.

Table 6.3 Performance overhead of PAM against Benign Samples

API being examined	Average API execution time (µsec)		Overhead
	gA	oA	
NtOpenProcess	779	620	0.79
NtOpenFile	5234	1924	0.37
NtCreateFile	2353	2059	0.87
NtWriteFile	6127	5479	0.89
NtClose	658	527	0.80

The results given in the Table 6.3 shows the overhead caused by four different native APIs that are intercepted by the proposed PAM. The maximum overhead of 0.89 was caused by NtCreateFile API function and the minimum overhead of 0.37 introduced when executing NtOpenFile function. This indicates that the overhead incurred by PAM before malware set has been executed on the workstations is 0.52 which is acceptable in real time.

Also, PAM has identified all the legitimate applications correctly and thus produces 0 false positives. Table 6.4 depicts the averaged value additional scores taken by the proposed PAM against different malware samples.

Table 6.4 Benchmark Scores against Malware Samples – *test case 1*

Malware samples	**Generic**	**PAM Enabled**
1 – 10	1315	1347
11 – 20	1285	1315
21 – 30	1341	1356
31 – 40	1311.7	1322.7
1 – 50	1298	1327
Average	6550.7	6667.7

In addition, assessing the efficiency of PAM is required to determine its impact on the overall system performance and the overhead caused by PAM. The measurement of PAM's performance is measured using PCMark8 benchmark tool [147]. All tests were conducted without user intervention except for executing the benchmark. The benchmark results have taken an average of 10 iterations. PAM is tested by loading all its components. PCMark8 is operated by simultaneously performing various different system level operations such as I/O operations, process creation and system call invocations. The benchmark provides the overall performance of the system relative to overall score.

The Table 6.4 also shows that when evaluating PAM on a single work station (with 50 malware samples) incurred smaller overhead (117 Scores). While performing I/O operations and system call operations in parallel, PAM incurred total score of 227 and 229 respectively. Figure 6.6 reports the false positives achieved by all kernel level anti-malware detection tools against malware samples and benign samples taken for *test case1*. Out of 4 tools, PAM closely shows superior results than all existing anti-malware detection tools with no FPR. Actually, producing no false positive tuned the performance of the PAM.

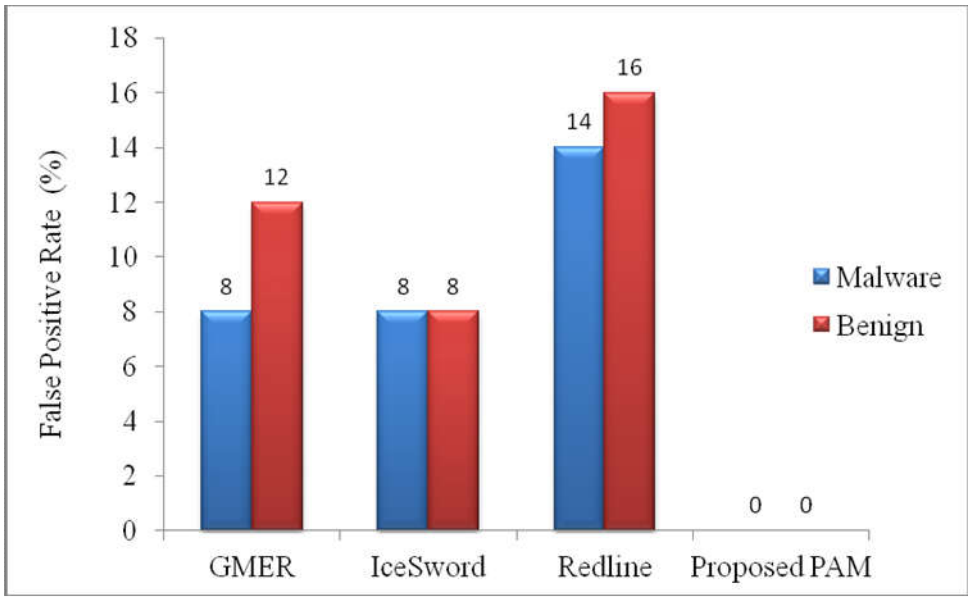

Figure 6.6 False Positive comparison (*Test case1*) with existing anti-malware detection tools

Test case 2:

The performance overhead incurred by PAM is computed by measuring the CPU cycles to be taken for executing additional tasks such as intercepting each system call, API function and authenticating their originality. To evaluate the additional overhead caused by PAM, a client-server test bed that runs with 2.8 GHz Intel Pentium 4, 4 GB RAM and running both Windows XP professional OS with SP3 as well as Windows 7 is arranged. To validate the experimental results and confirm the adaptability of PAM in real time, a client-server model as a test-bed has been setup as *test case 2*.

The noteworthy servers such as an FTP server, and IIS server, and an IRC chat server include all malicious samples. Another server runs the only web server in full trust mode which only holds benign samples to examine trusted communications. The end computer installed with Windows XP OS run different client programs such as FTP client applications, email application such as Thunderbird, IRC clients, web browsers such as IE and Firefox, newsgroups and eMule which are attractive targets of malware attacks. Certain protocols such as FTP, ICMP, IRC, SMTP and eMule are defined as dangerous. Then, few malicious samples are purposefully introduced into the host and carried out different actions between the client and server. With this setup, the effectiveness of PAM can be carefully evaluated with and without considering it.

It is also confirmed whether the malicious samples are executed on the host machine using the log files generated by the tools with and without setting security protection. The same is manually verified in the log files of process, files and registry entries. Except the file-copy operation, the behavior of few native API functions can be monitored only by intercepting one system call, for example, NtCreateFile(), and NtOpenFile(), etc. However, behaviors like code-injection into other processes consist of invocation of multiple system calls such as OpenProcess(), VirtualAllocEx(), WriteProcessMemory(), CreateRemoteThread(), etc. Therefore, it is planned to intercept the first system call itself to prevent the execution of subsequent calls which would disallow other subsequent calls. The outcome is confirmed using three popular anti-rootkit detection techniques: GMER [136], IceSword [132], and Redline [146]. All these tools are capable of detecting rootkit malware samples that target SSDT hook attacks.

Each security tool is tested against all of the samples. For each kind of samples, the total number of false positives and false negatives are computed after launching PAM. A false positive is noted when a security tool incorrectly identifies a benign activity as malicious. Figure 6.7 shows the False Positive Rate (FPR) obtained by testing all the four malicious code detection tools including PAM.

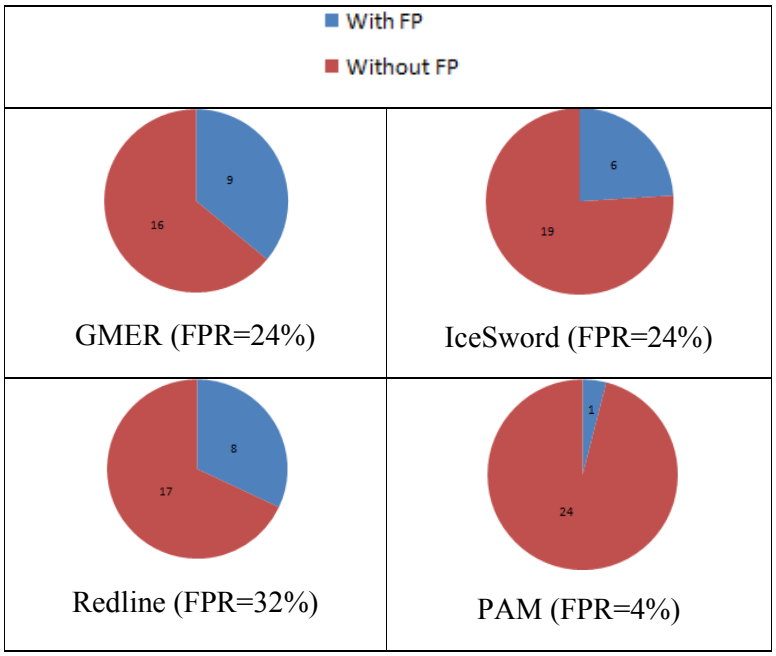

Figure 6.7 False Positive comparison (*Test case 2*) with existing anti-malware detection tools

Both GMER and IceSword anti-malware security tools incurred FPR about 24%, but in case of Redline it was 32%, whereas PAM had FPR of 4%. When the PAM is re-started againg, it identified all malware correctly, thus causes 0 % FPR. On the other hand, the False Negative Rate (FNR) of GMER and IceSword was 20% and 15% respectively. As Redline's detection capability of SSDT hook attacks depends on the type of API function to be hooked, it achieved FNR by 65%. However, the FNR of PAM was almost zero. As a conclusion of this test, the PAM can effectively block any type of malware that targets system call hooks. But none of the existing anti-malware security tools tested dealt kernel level authentication mechanism to verify the originality of a system call invocation. For all test cases, every system call and API function are invoked 100 times and their averaged CPU cycles are given in Table 6.5 With PAM enabled, the malicious executable incurred 1.3-38.1% performance overhead than native functions, while the benign executable incurred only 0-1.9%.

Table 6.5 Measurement of performance overhead (CPU Cycles)
(The columns PAM-b and PAM-m illustrate the CPU cycles acquired by the benign programs and malware programs).

Function	Native	PAM-b	PAM-m
NtOpenFile	167703	169721 (1.2%)	169823 (1.3%)
NtWriteFile	245201	249993 (1.9%)	338546 (38.1%)
NtCreateFile	334568	338579 (1.2%)	348579 (4.2%)
NtCreateProcess	206556	208945 (1.1%)	215326 (4.2%)
OpenService	6568202	6568423 (<0.1%)	6679899 (1.7%)

The result of intercepting NtWriteFile() function produced the highest performance overhead of 38.1% as an outcome of capturing file-copy operation. As each system is designed to enforce different policies, it is hard to compare the overhead of authenticated system call with other system call monitor. The generic performance overhead impact is 2% PAM as a result of intercepting the system call and API function to verify their originality which is well below of other systems and also acceptable. Table 6.6 gives a comparison of performance overhead PAM in terms of relative scores measured by PCMark benchmark against malicious code detection and prevention using the combination of user-mode information and kernel-mode information.

The performance overhead of PAM is increased by 96 scores in total (almost 1% increase compare to test case 1) when a client-server model with a largest dataset is used. This is because of attempting to run PAM by executing multiple legitimate applications, malwares samples, I/O operations and system call operations simultaneously. However, such overhead is acceptable and does not affect the overall system performance but ensures total security against malicious code attacks.

Table 6.6 Benchmark scores against Malware samples – *test case 2*

Malware samples	**Generic**	**PAM Enabled**
1 – 60	1442	1483
61 – 120	1401	1443
121 – 180	1399	1438
181 – 240	1438.5	1484.5
241 – 300	1421.6	1466.6
Average	7102.1	7315.1

6.4 Verification by Mathematical Model

Hypothesis Test. A statistical verification has been conducted to check whether there is a significant deviation in the performance of the kernel/system before and after enabling the PAM. To achieve this, two null and alternative hypothesis are defined as follows.

H0: Statistically, there is no significant difference in the kernel performance after enabling PAM.

H1: Statistically, there is some association between before and after enabling PAM.

The given data in Table 6.7 has shown the CPU cycles as obtained from the two test cases. Hence the CPU cycles in the two tests can be regarded as correlated and therefore, the t-test for paired values was opting to confirm the performance deviation between these two cases. Let $d = x1 - x2$ and $\bar{d} = \Sigma d / n$, where x1 and x2 denote the CPU cycles in the two tests and n is the number of functions tested.

Table 6.7 t-test computation

Function	PAM-b		PAM-m	
	d	**d^2**	**d**	**d^2**
NtOpenFile	2018	4072324	2120	4494400
NtWriteFile	4792	22963264	93345	8713289025
NtCreateFile	4011	16088121	14011	196308121
NtCreateProcess	2389	5707321	8770	76912900
OpenService	221	48841	111697	12476219809
	Σd=13431	Σd2=180391761	Σd=229943	Σd2=21467224255
	t_{cal} = 0.912871		t_{cal} = 1.970616	

Applying t-test, $t = \frac{\bar{d}\sqrt{n}}{S}$ where $\bar{d}$ represents the mean of the difference and S represents the standard deviation of the difference. From t-table, for γ=n-1=4 degrees of freedom, $t_{0.05}$=2.776. In both cases, i.e., PAM-b and PAM-m, $t_{cal} < t_{0.05}$, hypothesis H0 has been accepted and it is concluded that there is no significant change in the kernel/system performance after enabling PAM framework.

Security Assurance

The strength of security protection guaranteed by PAM is verified by analyzing the confidentiality of the credentials used on authentication stage and the integrity of PAM components. Without using a strong pseudorandom number generator for generating secret credentials, forging the existing credentials is impossible. Also, a malicious process's arbitrary code that might try to replace PAM generated credentials. It cannot successfully bypass the authentication stage. This is because the arbitrary code which is not generated by PAM does not appear in the record. To prevent another application, revealing the secret information generated to perform a challenge-response attack to be launched by a malware, PAM restricts read access. A malware may attempt to steal secret information from PAM components or application's memory at runtime. This issue is resolved by using the typical process memory segregation feature offered in the OS itself. PAM ensures confidentiality by disallowing the other applications that have direct access to the secret credentials except PF and CGF which are kernel helper processes.

PAM components span both the user-space and the kernel- space. The Runtime Authenticator resides in the kernel. As we trust the integrity of kernel resources, this component is trustworthy. However, the integrity of user-space components, the PF and the process CGF need to be confirmed which may be the ideal targets of malware spoofing and tampering. Only the kernel of the OS can access or modify the code segment of these components.

(i) Forge – When the malicious code directly requests the kernel service, the authenticator which resides in the kernel checks if it has already authenticated. For a given unique process identifier which can never be forged to call a system service, the authenticator will immediately asks the requests to reach the process validation function

(ii) Password Guess – The malicious code cannot bypass the process validation function. The malicious code can evade the authentication check function, if and only if, it successfully retrieve or guess the secret credential information. Recall that the secret information is 23 or 55 bits in length and assigned randomly for a suspicious process at load time which is very hard to retrieve.

As the value of this is hard-coded into the preservation function which is a kernel helper process, learning it by scanning the code segment is not permitted. When a legitimate process invokes a system service call, the secret information will be unavailable in the stack and removed after successful check. However, malicious code cannot intercept this request to learn the value of secret information. The process validation function resides in the read-only page of the code segment. Only the kernel helper process (i.e., process with highest privilege) can only access or modify the code segment of the authentication function.

Compatibility and Usability

The good compatibility of PAM is ensured by achieving a significant reduction of the false positives. PAM directly prevents malicious code attacks rather than detecting illegitimate information flooding that may result in a false positive. Additionally, we formulate different exception rules to prevent producing a large number of false positive results. This actually helps to reduce the net FPR of PAM. Software usability mainly concerns with the assessment of effectiveness and efficiency with which end-users can perform tasks with a software tool.

Assessing usability now becomes an important element in the software development process. As PAM does not include any configuration settings, it automatically detects and prevents the potential malicious code API hook attacks.

Chance of Successful attack

PAM is capable of classifying interpreted software applications functioning as a stand-alone process. However, PAM cannot reveal the malicious code that is already injected into the authenticated process and runs as a stand-alone process. The strength of PAM lies on accuracy of classification precision. The trustworthy classification of an application is a challenging and difficult task, and it inexactness may permit a malware to acquire the secret credentials of a legitimate process. In order to improve the detection capability of PAM, many advanced static and dynamic monitoring and analyzing techniques need to be combined. Figure 6.8 shows that PAM outperforms than existing approaches proposed by Sun et al., Hau et al., and Salehi et al. in all aspects with improves security level with 100 % detection rate.

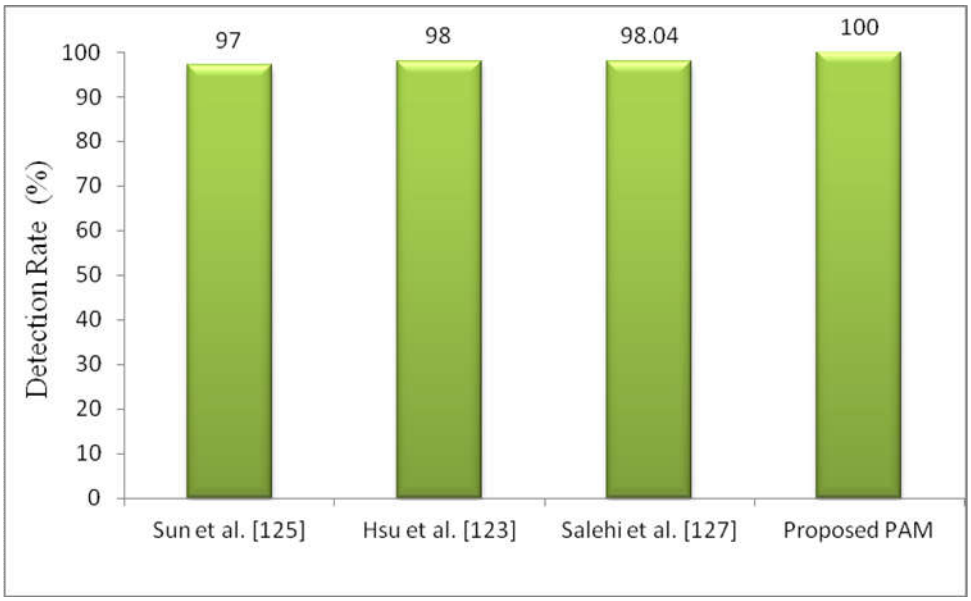

Figure 6.8 Overall Detection Rate comparison of PAM

6.5 Overall Comparison of existing Approaches and PAM

The notable strength of PAM is that it will directly prevent the malicious code hook attacks rather than locating them that many results in a false positive. There are various existing works have been proposed in the past by researchers to strengthen the

security of the kernel, but only few of them dealt kernel level authentication to prevent malicious code attacks such as hooking kernel level data structures.

Figure 6.9 depicts the overhead incurred by existing approaches while detecting and preventing malicious code attacks at the kernel-mode including PAM. Almost all existing approaches implemented for preventing malicious code attacks with the combination of user-mode and kernel-mode information, but did not apply kernel level process authentication mechanism. A few existing approaches obtained low performance overhead using either real-time dataset or own dataset except the approach proposed by Sun et al. with 36.7 % of overhead. This is because the proposed method by Sun et al. has only tested legitimate API functions which involved few important additional function calls. However, the proposed PAM mechanism outperforms than other existing approaches listed in the literature with 2% runtime overhead and has the potential to be customized or used in real-time. The PAM mechanism outperforms than the existing mechanisms in all aspects and has the potential to be combined real-time malware defense to provide stronger security for preventing different kind of malicious code attacks.

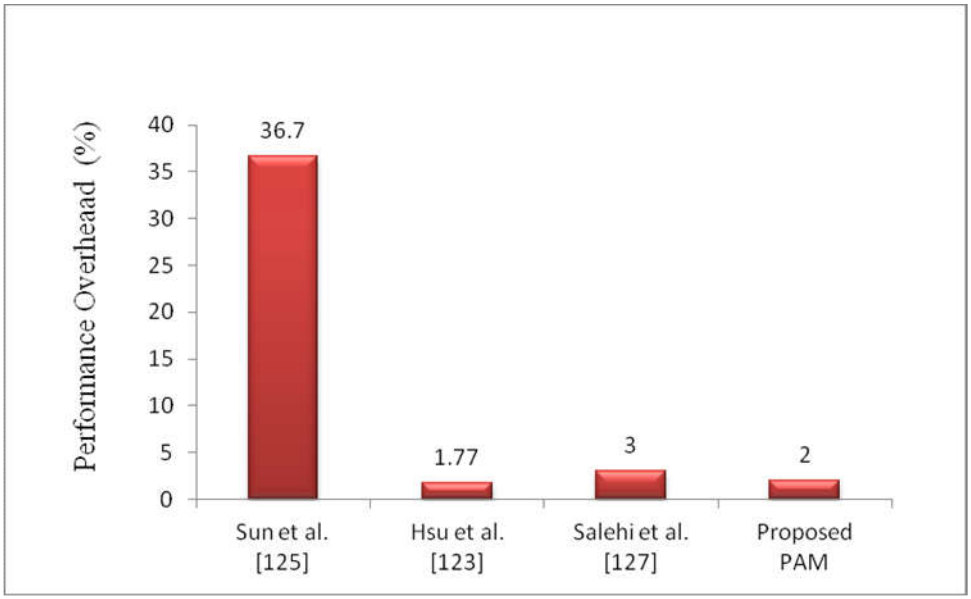

Figure 6.9 Overall runtime Overhead comparison of PAM
(Computed using additional time taken)

The performance overhead, false positives and sample set for test analysis of existing techniques for the prevention of malicious code attacks that target kernel level hooking attacks are compared and tabulated is shown in the Table 6.8.

Table 6.8 Comparison of existing malware detection and prevention Approaches with proposed PAM for Windows

Sl.No.	Existing Solution	Kernel Recompilation	Features						
			Protection of User-mode	Resist hidden process attacks	Resist Malicious code attacks	Supply of Incorrect ID	Runtime Overhead	Security Level	Detection Level
1	Rajagopalan et al. [19]	Policy based Detection	NO	Neglected	YES	Policy Based	7.92%	Minimal	Kernel-Level(Linux)
2	Nguyen et al. [126]	NO	NO	NO	YES	System Crash	9%	LOW	Kernel-Level
3	Yin et al. [114]	NO	Neglected	Neglected	NO	NO	0 %	Minimal	Kernel-Level
4	Sun et al. [125]	NO	Neglected	Neglected	NO	NO	36.70%	Minimal	Kernel-level
5	Hsu et al. [123]	NO	Neglected	NO	YES	NO	LOW	Minimal	Kernel-Level
6	Salehi et al. [127]	NO	NO	NO	YES	NO	(FP) 3%	Minimal	OS-Level
7	Proposed PAM	NO	YES	YES	YES	NO	2%	HIGH	User & Kernel-Level

6.8 Summary

Kernel level authentication has been effectively applied to identify and prevent malicious executable before the cause damage to the end-system. The PAM, a kernel level process authentication mechanism has been developed and implemented with an objective to detect and prevent unauthorized access through processes of a malicious application to a greater extent. PAM is a security enhancement mechanism that incorporates an algorithm for discovering hidden processes and services has been developed and implemented to authenticate all suspicious system calls made by the processes during runtime before getting services form the kernel of the OS.

To evaluate the performance overhead and suitability of the PAM, several experiments have been conducted on Windows to study the effectiveness of the PAM in terms of detection rate and performance overhead. According to simulation experimental results PAM outperforms the existing approaches proposed by Sun et al., Hau et al., and Salehi et al. with negligible performance overhead (2%) and 100 % detection rate. The PAM mechanism gives considerable security improvement over the approached proposed by Sun et al., Hau et al., and surpasses the security mechanism proposed by Salehi et al.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

This chapter concludes the dissertation with a brief discussion on the merits of the developed algorithms and techniques developed for malicious code detection and prevention. It also reveals a few open problems in the focused area of research.

7.1 Conclusions

The most ominous form of cyber attack is imperceptible. Such attacks may become a targeted attack that mainly utilizes some sophisticated stealthy techniques, such as defeating the OS, evading AV software, etc., Poor system configurations and security policies may act as entry-point and permit a remote attacker to easily bypass the predefined system security policies and execution of different attacks against vulnerable workstations and servers. More advanced attacks on a large private network is only possible with the existence of susceptible intermediate workstations in the network. Remote attacker makes network malware analysis more difficult by encrypted packets and end-to-end encryption. Therefore, another suitable place to detect and prevent malicious code attacks is the end-system.

A stealthy attacking technique works silently, evading the footprints of an attacker's events. By masking evidence of the operations, the criminal had enough time to perform any kind illegal activities. In order to stay ahead of the traditional security measures and tools, many cyber criminals apply ingenuity in stealth plans. Because Windows is the one of the most popular and widely used operating system in the information communication environment for personal computers, it becomes most attract target for malware writers. It is possible for a remote attacker to control a large network even by compromising a server or workstation. Tampering the kernel of the operating system would question the trustworthiness of the underlying computing environment. Therefore, Kernel integrity is more important to ensure a secure computing environment. For years, security experts have urged the requirement of extra layer of security measures across workstations and networks.

Though many approaches have been suggested to detect and analyze malicious code attacks, still they suffer the adaptation of ineffective approaches. In addition, literature survey reveals that system call monitoring using policy based technique are not sufficient to ensure trustworthiness of the computing environment by detecting and preventing malicious code attacks. Alternatively, the process authorization mechanisms can also an alternate mechanism but not sufficient to handle stealthy malicious code attacks. In this research, graph-based malware detection approach, detection of malicious-code attacks at User-mode, hidden processes and services detection algorithm, and kernel level process authentication mechanisms have been proposed. The developed approaches and algorithms were evaluated by conducting simulation experiments using different datasets.

(i) A graph based static approach namely, GraMD has been developed for detecting malware attacks. The graph-based approach classified malware attacks based on monitoring and capturing the execution of system calls while interacting with the kernel of the operating system. Two novel graph-based algorithms namely, ACA and GMA have also been developed and incorporated into GraMD for API call graph generation and comparing two graphs respectively. The proposed GraMD approach each system call as a call graph using ACA algorithm. Then, the generated call graph is compared against approach graphs using GMA algorithm by determining the similarity value through means of graph isomorphism.

The experimental results show that GraMD outperforms the existing approached proposed by Park et al., Zhao et al., and Elhadi et al. with 97.68-100 percent detection rate and 3.40-6 percent false positive rate. It is also verified mathematically that the GraMD utilizes only minimum number of API function calls using game theoretic approach and thus GraMD takes less time with reduced space requirement compare to existing graph based malware detection approaches. However, with an increasing amount of malware adopting rootkit techniques to evade AV, further research into defenses against malicious code attacks is absolutely essential.

(ii) A dynamic based user-mode malicious code detection and prevention approach namely, UMDetect has been proposed. The proposed UMDetect traced and prevent user-mode malware with native API hook functionality in Windows.

UMdetect make use a novel DLL classification algorithm namely, DCA algorithm which has been proposed and implemented to determine whether the exported / imported DLL file is malicious or legitimate. Experimental results indicate that the UMDetect outperforms than existing anti-malware detection tools such as BlackLight, IceSword, VICE, and R3 Hook Scanner with 100 percent detection rate and negligible performance overhead. In addition, the overhead of UMDetect (1300 CPU cycles for completion) is also compared with the methods proposed by Deng et al., Abonghadareh et al., and Yoghi et al. Because, the DCA used only countable API function for completion, the runtime complexity of DCA is very negligible. In order to optimize the proposed UMDetect approach, a new algorithm for detecting hidden entries of a malicious application has been proposed which is explained next.

(iii) The problem of discovering hidden footprints of a malicious executable in Windows has also been explored in this thesis work. Because, advanced malware authors have taking the advantage of rootkit technique to evade the footprints of their malicious code, detecting them is a challenging and can also be used for optimizing performance overhead of a malware detection approach. A cross-view based hidden processes and services detection algorithm namely, CoPDA has been proposed. At runtime, the CoPDA algorithm discovers all hidden entries in Windows and validates whether they are legitimate or suspicious.

Experimental results indicate that the CoPDA algorithm outperforms than existing anti-rootkit detection tools such as Hellioslite, GMER, HiddenFinder, IceSword, BlackLight, and Rootkit Unhooker with 100 percent accuracy rate and 1.82 percent of false positives. In addition, the performance overhead of CoPDA algorithm has also been compared against existing techniques cross-view based hidden process detection approaches proposed by Desheng et al., Xie et al., and Richer et al. The CoPDA algorithm caused a tiny overhead of 0.6 percent (10.762 seconds taken for completion).

(iv) This thesis finally proposed a kernel level process authentication mechanism namely, PAM to validate the originality of all suspicious processes of a malicious executable during runtime.

PAM enhanced the security strength of the computing environment by the combination of user-mode information through CoPDA algorithm for discovering suspicious processes and services of a malicious executable and kernel mode information by authenticating all identified suspicious processes and services during run time. The proposed mechanism is an extension of process authentication mechanism by incorporating strong security check. The impact of how process authentication mechanism can effectively isolate and disallow malicious processes from getting system services and preventing system resources against malicious code attacks are also discussed. PAM ensure only authenticated service request being serviced by the kernel, it blocks all malicious processes and thwart subsequent attacks.

The effectiveness of PAM is evaluated using different datasets, benchmarks, and test-beds. Experimental results show that PAM surpasses existing anti-rootkit detection tools such as GMER, IceSword, and Redline with zero percent false positives. The low overhead of 1 percent when evaluating PAM on a single computer (with smaller number of malware and legitimate samples), is because of the testbed with single computer and the variability present in malware samples. But the overhead increased to 2 percent when a client-server scenario with various malware samples is used. The proposed PAM outperformance than the existing approaches proposed by Sun et al., Hsu et al., and Salehi et al. with the generic overhead of 1-2 percent and does not significantly affect the overall system performance. Mathematical verification can also be done to verify the same. The authentication mechanism of PAM is portable and can be integrated with other static or dynamic behavior based system call monitoring tools with customization.

The essence of this research work lies in better detection and prevention of unauthorized process of a malicious executable through authenticating its originality during runtime which would significantly contribute to the better enhancement of this research work.

7.2 Future Research Directions

In this research, an attempt is made to provide solutions for detecting and preventing malicious code attacks by developing and implementing a kernel level process authentication mechanism. The various algorithms and techniques developed in this thesis can be further extended in the following sections.

(i) The algorithms developed for graph-based malware detection approach applied modified graph-edit distance isomorphism algorithm for comparing two graphs. Because, graph matching algorithms plays a vital role in malware detection process, GMA algorithm can be extended to optimize its complexity further to some extent.

(ii) The algorithm developed for finding hidden suspicious entries can be extended to use authentication mechanism to overcome the limitations of misusing undocumented specifically, kernel level API functions.

(iii) As cyber criminals will make sure that their malicious software creations work equally on Windows XP to Android OS, porting PAM to Android OS for mobile devices to provide strong authentication to applications is very essential. This will be considered as future work.

REFERENCES

[1] MacAfee Lab 2016 threat Report. Available at http://www.mcafee.com/in/resources/reports/rp-quarterly-threats-mar-2016.pdf

[2] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray, ”A semantics based approach to malware detection”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 5, Article 25, 2008.

[3] Yanfang, “IMDS: Intelligent malware detection system”, *Proceeding of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1043-1047, 2007.

[4] Jeremy Z. Kolter and Marcus A. Maloof, “Learning to detect malicious executables in the wild”, *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 470–478, 2004.

[5] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand, “Practical taint-based protection using demand emulation”, *Proceedings of the EuroSys conference*, April 2006.

[6] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis”, *Proceedings of the 14th ACM conference on Computer and communications security (CCS ’07)*, pp. 116-127, October 2007.

[7] M. Bernaschi, E. Gabrielli, and L. Mancini, ”Operating System Enhancements to prevent the Misuse of System Calls”, *Proceedings of the ACM Conference on Computer and Communication Security (CCS’00)*, pp. 174-183, 2000.

[8] K.Jain and R.Sekar,"User-level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement" *Proceedings of the Network and Distributed Systems Security Symposium*, pp. 19-34, February 2000.

[9] T.Garfinkel, "Traps and pitfalls: Practical Problems in System Call Interposition Based Security Tools", *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[10] C.Kruegel, D.Mutz, F.Valeur, and G.Vigna, "On the Detection of Autonomous System Call Arguments", *Proceedings of the Eighth European Symposium Research in Computer Security (ESORICS '03)*, pp. 326-343, 2003.

[11] N.Provos, "Implementing Host Security with System Call Policies", *Proceedings of the 12th USENIX Security Symposium*, pp. 257-272, August 2003.

[12] R.Sekar, V.Venkatakrishnan, S.Basu, S.Bhatkar, and D.Duvarney, "Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications", *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, pp. 15-28, 2003.

[13] T.Garfinkel, B.Pfaff, and M.Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition", *Proceedings of the Network and Distributed Systems Security Symposium*, February 2004.

[14] P.Loscocco and S.Smalley, "Integrating Flexible Support for Security policies into the Linux Operating System", *Proceedings of the USENIX Annual Technical Conference*, pp. 29-42, 2011.

[15] grsecurity. [Online] Avaiable : http://www.grsecurity.net/

[16] Saeed Parsa and Somaye Arabi Naree, "A New Semantic Kernel Function for Online Anomaly Detection of Software", *ETRI Journal*, vol. 34, no. 2, pp. 288-291, April 2012.

[17] Nguyen Anh Quynh and Yoshiyasu Takefuji, ”Towards a tamper-resistant kernel rootkit detector”, *Proceedings of the ACM symposium on Applied computing (SAC ’07)*, pp. 276–283, March 2007.

[18] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, “Lares: An architecture for secure active monitoring using virtualization”, *Proceedings of the IEEE symposium on Security and Privacy (SP ’08)*, pp. 233–247, May 2008.

[19] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting, “System Call Monitoring Using Authenticated System Calls”, *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 216–229, July-September 2006.

[20] Battistoni, R., E. Gabrielli and L.V. Mancini, “A host intrusion prevention system for Windows operating systems”, *Proceeding of 9th European Symposium on Research in Computer Security (ESORICS ’04)*, pp. 352-368, 2004.

[21] Flavio Lombardi and Roberto Di Pietro., ”KVMSec: A security extension for linux kernel virtual machines”, *Proceedings of the ACM symposium on Applied Computing*, pp. 2029-2034, 2009.

[22] Nguyen Anh Quynh and Yoshiyasu Takefuji., ”Towards a tamper-resistant kernel rootkit detector”, *Proceedings of the ACM symposium on Applied computing*, pp. 276-283, March 2007.

[23] Koichi Onoue, Yoshihiro Oyama, and Akinori Yonezawa., ”Control of system calls from outside of virtual machines”, *Proceedings of the ACM Symposium on Applied Computing*, pp. 2116–2121, 2008.

[24] Trusted Computing Group, *TCG Specification Architecture Overview*, 2004.

[25] Symantec Advanced Threat Research. Technical Report, Security implications of Microsoft Windows Vista, [Online] Available: www.symantec.com/avcenter/reference/Security_Implications_ of_Windows_Vista.pdf, February 2007.

[26] J. Rutkowska. *Subverting vista kernel for fun and profit*, August 2006.

[27] Skywing. Bypassing patch guard on windows x64. Technical Report. Available: http://www.uninformed.org/?v=3&a=3&t=sumry, December 2005.

[28] Fabrice Bellard. Qemu, ”A Fast and Portable Dynamic Translator”, *Proceedings of the 2005 USENIX Annual Technical Conference (ATEC ’05)*, pp. 41-41, 2005.

[29] Keith Adams and Ole Agesen, “A comparison of software and hardware techniques for x86 virtualization”, *Proceedings of the 12th International conference on Architectural support for programming languages and operating systems*, pp. 2-13, November 2006.

[30] K.Xu, H.Xiong, D.Stenfan, C.Wu, and D.Yao, ”Data-Provenance verification for secure hosts”, *IEEE Transaction on Dependable and Secure Computing*, vol. 9. no. 2, pp. 173-183, March-April 2012.

[31] W.Dai, T.P.Parker, H.Jin, and S,Xu, ”Enhancing data trustworthiness via assured digital signing”, *IEEE Transaction on Dependable and Secure Computing*, vol. 9, no. 6, pp. 838-851, 2012.

[32] Morris Worm Shut down ten percent of the Internet. [Online] Available: http://www.atlasobscura.com/articles/

[33] A.K. Sood, R. Bansal, and R.J. Enbody, ”Cybercrime: Dissecting the state of underground enterprise”, *IEEE Internet Computing*, vol. 17, no. 1, pp. 60–68, 2013.

[34] Brett Stone-Gross, Ryan Abman, Richard A Kemmerer, Christopher Kruegel, Douglas G Steigerwald, and Giovanni Vigna, ”The Underground Economy of Fake Antivirus Software”, *Proceedings of the Workshop on Economics of Information Security and Privacy III*, pp. 55–78, July 2012.

[35] MacAfee labs threats report, *Technical report*, Intel security, 2015.

[36] G. Hoglund and J. Butler, *Rootkits: Subverting the windows kernel*, Addison Wesley, 2006.

[37] Gregoire Jacob, Herve Debar, and Eric Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy", *Journal in Computer Virology*, vol. 4, pp. 251–266, 2008.

[38] Peter Szor, *The Art of Computer Virus Research and Defense*, Addison Wesley Professional, 2005.

[39] A. Patcha and J. M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, no. 12, pp. 3448–3470, 2007.

[40] Kuo-Chen Lee, Jason Chang, Ming-Syan Chen, "PAID: packet analysis for anomaly intrusion detection", *Proceedings of the 12th Pacific-Asia conference on Advances in knowledge discovery and data mining (PAKDD'08)*, pp. 626-633, 2008.

[41] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, September 2009.

[42] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An Overview of IP Flow-Based Intrusion Detection", *IEEE Communication Surveys and Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[43] Jonathan J. Davis, Andrew J. Clark, "Data preprocessing for anomaly based network intrusion detection: A review", *Computers and Security*, vol. 30, no. 6-7, pp. 353-375, September 2011.

[44] Yingbing Yu, “A survey of anomaly intrusion detection techniques”, *Journal of Computer Science in Colleges*, vol. 28, no. 1, pp. 9-17, October 2012.

[45] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-ChihLin, “Intrusion detection system: A comprehensive review*”, Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, January 2013.

[46] Bhuyan, M.H., Bhattacharyya, D.K.,Kalita, J.K., ”Network Anomaly Detection: Methods, Systems and Tools”, *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 303-336, 2014.

[47] Hanghang Tong, Chongrong Li, Jingrui He, Jiajian Chen, Quang-Anh Tran, HaixinDuan, Xing Li, ”Anomaly Internet Network Traffic Detection by Kernel Principle Component Classifier”, *Proceedings of the 2nd International Symposium on Neural Networks*, pp. 476-481, 2005.

[48] F.S.Wattenberg, J.I.A.Perz, P.C.Higuera, M.M.Fernandez, and I.A.Dimitriadis, “Anomaly Detection in Network Traffic Based on Statistical Inference and alpha-Stable Modeling”, *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 494-509, August 2011.

[49] Yuh-Jye Lee, Yi-Ren Yeh, and Yu-Chiang Frank Wang, “Anomaly Detection via Online Oversampling Principal Component Analysis”, *IEEE Transactions on Knowledge and Data Engineering,* vol. 25, no. 7, pp. 1460 – 1470, July 2013.

[50] S.Jianga, X.Songb, H.Wangc, Jian-Jun Hand, Qing-Hua Li, ”A clustering-based method for unsupervised intrusion detections”, *Pattern Recognition Letters*, vol. 27, no. 7, pp. 802-810, May 2006.

[51] Z. Zhuang, Y. Li, and Z. Chen, “Enhancing Intrusion Detection System with proximity information”, *International Journal of Security and Networks*, vol. 5, no. 4, pp. 207–219, December 2010.

[52] Jabez.J, B.Muthukumar, “Intrusion Detection System (IDS): Anomaly Detection using outlier Detection Approach”, *Procedia Computer Science*, vol. 48, pp. 338-346, 2015.

[53] G. Liu, Z. Yi, and S. Yang, “A hierarchical intrusion detection model based on the PCA neural networks,” *Neuro computing*, vol. 7, no. 7-9, pp. 1561–1568, 2007.

[54] X.Song, M.Wu, C.Jermaine, and S.Ranka, ”Conditional Anomaly Detection”, *IEEE Transaction on Knowledge and Data Engineering*, vol. 19, pp. 631-645, May 2007.

[55] A.O.Adetunmbi, S.O, Falaki, O.S.Adewale, and B.K.Alese, “Network Intrusion Detection based on Rough set and k-Nearest Neighbour”, *International Journal of Computing and ICT Research*, vol. 2, no. 1, pp. 60-66, 2008.

[56] M.Y.Su, G.J.Yu, and C.Y.Lin, “A real time network intrusion detection system for large scale attacks based on an incremental mining approach”, *Computers & Security*, vol. 28, no. 5, pp. 301-309, 2009.

[57] A. Tajbakhsh, M. Rahmati, and A. Mirzaei, “Intrusion detection using fuzzy association rules”, Applied Soft Computing, vol. 9, no. 2, pp. 462–469, March 2009.

[58] F.Geramitaz, A.S.Memaripour, and M.Abbaspour, “Adaptive Anomaly Based Intrusion Detection System Using Fuzzy Controller”, *International Journal of Network Security*, vol. 14, no. 6, pp. 352-361, 2012.

[59] M. S. A. Khan, “Rule based Network Intrusion Detection using Genetic Algorithm”, *International Journal of Computer Applications*, vol. 18, no. 8, pp. 26–29, March 2011.

[60] A. Visconti and H. Tahayori, “Artificial immune system based on interval type-2 fuzzy set paradigm”, *Applied Soft Computing*, vol. 11, no. 6, pp. 4055–4063, September 2011.

[61] N. G. Duffield, P. Haffner, B. Krishnamurthy, and H. Ringberg, "Rule-Based Anomaly Detection on IP Flows", *Proceedings of the 28th IEEE International Conference on Computer Communications*, pp. 424–432, 2009.

[62] T. Abbes, A. Bouhoula, and M. Rusinowitch, "Efficient decision tree for protocol analysis in intrusion detection", *International Journal of Security and Networks*, vol. 5, no. 4, pp. 220–235, December 2010.

[63] Z. Muda, W. Yassin, M. N. Sulaiman, and N. I. Udzir, "A K-meansand naive bayes learning approach for better intrusion detection", *Information Technology Journal*, vol. 10, no. 3, pp. 648–655, 2011.

[64] Francesco Palmieri, Ugo Fiore, and AnielloCastiglione, "A distributed approach to network anomaly detection based on independent component analysis", *Concurrency and Computation: Practice and Experience*, vol. 26, no. 5, pp. 1113–1129, April 2014.

[65] X. Tong, Z. Wang, and H. Yu, "A research using hybrid RBF/Elmanneural networks for intrusion detection system secure model", *Computer Physics Communications*, vol. 180, no. 10, pp. 1795–1801, 2009.

[66] G. Folino, C. Pizzuti, and G. Spezzano, "An ensemble-based evolutionary framework for coping with distributed intrusion detection", *Genetic Programming and Evolvable Machines*, vol. 11, no. 2, pp. 131–146, June 2010.

[67] D. Ariu, R. Tronci, and G. Giacinto, "HMMPayl: An intrusion detectionsystem based on Hidden Markov Models", *Computers & Security*, vol. 30, no. 4, pp. 221–241, 2011.

[68] S.Benferhat, A.Boudjelida, K.Tabia, H.Drias, "An intrusion detection and alert correlation approach basedon revising probabilistic classifiers using expert knowledge", *Applied Intelligence*, vol. 38, no. 4, pp. 520-540, June 2013.

[69] Hsien-De Huang, Chang-Shing Lee, Mei-Hui Wang, Hung-Yu Kao, "IT2FS-based ontology with soft-computing mechanism for malware behavior analysis", *Soft Computing*, vol. 18, no. 2, pp. 267-284 , February 2014.

[70] G.Bonfante, M.Kaczmarek and J.Y.Marion, "Control Flow Graphs as Malware Signatures", *Proceedings of the Inter Regional workshop on Rigorous System Development and Analysis*, October 2007.

[71] M. Shafiq, Syed Khayam, and Muddassar Farooq, "Embedded malware detection using markov n-grams", *Lecture Notes in Computer Science*, vol. 5137, pp. 88–107, 2008.

[72] Silvio Cesare, Yang Xiang, and Wanlei Zhou, "Malwise: An effective and efficient classification system for packed and polymorphic malware", *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1193–1206, 2013.

[73] Younghee Park and Douglas S. Reeves, Mark Stamp, "Deriving common malware behavior through graph clustering", *Computers & Security*, vol. 39, pp. 419–430, 2013.

[74] Raymond Canzanese, Moshe Kam, and Spiros Mancoridis, "Multi-channel change-point malware detection", *Proceedings of the 7th International Conference on Software Security and Reliability (SERE)*, 2013.

[75] Edward Stehle, Kevin Lynch, Maxim Shevertalov, Chris Rorres, and Spiros Mancoridis, "On the use of computational geometry to detect software faults at runtime", *Proceedings of the International conference on autonomic computing (ICAC)*, pp. 109–118, 2010.

[76] Wang, M., C. Zhang and J. Yu, "Native API based windows anomaly intrusion detection method using SVM", *Proceeding of IEEE International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing (SUTC '06)*, vol. 1, pp. 514-519, 2006.

[77] Leian Liu, Zuanxing Yin, Yuli Shen, Haitao Lin, Hongjiang Wang", Research and Design of Rootkit Detection Method", *Physics Procedia*, vo. 33, pp. 852-857, 2012.

[78] Yi, X., H. Da-Rong and S., "Analysis of windows rootkits stealth and detection technologies", *Proceedings of the Second International Conference on Applied Robotics for power Industry*, 2010.

[79] White, A., B. Schatz and E. Foo, "Surveying the user space through user allocations", *Digital Investigation*, vol. 9, pp. S3-S12, August 2012.

[80] Hejazi, S.M., C. Talhi and M. Debbai, "Extraction of forensically sensitive information from windows physical memory", *Digital Investigation*, vol. 6, pp. S121-S131, September 2009.

[81] Deng, Z., D. Xu, X. Zhang and X. Jiang, "IntroLib: Efficient and transparent library calls introspection for malware forensics", Digital Investigation, vol. 9, pp. S13-S23, August 2012.

[82] Rabek, J.C. and R.I. Khazan, "Detection of Injected, dynamically and obfuscated malicious code", *Proceedings of the 2003 ACM workshop on Rapid malcode (WORM '03)*, pp. 76-82, 2003.

[83] Wagner, D. and P. Soto, "Mimicry attacks on host-based intrusion detection systems", *Proceeding of 9th ACM Conference on Computer and Communication Security*, pp. 255-264, 2002.

[84] Mansoori, M., O. Zakaria and A. Gani, "Improving exposure of intrusion deception system through implementation of hybrid honeypot", *International Arab Journal of Information Technology*, vol. 9, no. 5, pp. 436-444, 2012.

[85] J.Lee, K.Jeong, H.Lee, "Detecting metamorphic malwares using code graphs", *Proceedings of the 2010 ACM symposium on Applied Computing* , pp. 1970-1977, 2010.

[86] J. Kinable,O.Kostakis, "Malware classification based on call graph clustering", *Journal in Computer Virology*, vol. 7, no. 4, pp. 233-245, 2011.

[87] Swiler L.P, Phillips.C. Ellis.D.,Chakerian.S, "Computer-attack graph generation tool", *Proceedings of the DARPA Information Survivability Conference & amp (DISCEX '01)*, pp. 307-321, 2001.

[88] Pro Interactive DisAssembler, [Online] Available: http://www.hex-rays.com/.

[89] Monitor A. Spy and display API calls made by Win32 applications, [Online] Available: http://www.apimonitor.com.

.

[90] Michael R. Garey, David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*, WH Freeman & Co., 1979.

[91] H.Guo, J.Pang,Y.Zhang,F.Yue,R.Zhao, "HERO: A novel malware detection framework based on binary translation", *Proceedings of the IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*, pp. 411-415, 2010.

[92] J.Li, Z.Wang, Bletsch.T, Srinivasan.D, Grace.M, X.Jiang, "Comprehensive and Efficient Protection of Kernel Control Data", *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 4, pp. 1404-1417, December 2011.

[93] S. Zander, T. Nguyen and G. Armitage, "P2P Traffic Identification Based on the Signature of Key Packets", *Proceedings of the 14th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD '09)*, 2009.

[94] Fatemeh Karbalaie, Ashkan Sami and Mansour Ahmadi, "Semantic Malware Detection by Deploying Graph Mining", *International Journal of Computer Science Issues*, vol. 9, no. 3, pp. 373-379, January 2012.

[95] Riesen, K. and H. Bunke, ”Approximate graph edit distance computation by means of bipartite graph matching”, *Image Vision Compute*, vol. 27, pp. 950-959, 2009.

[96] Faraz Ahmed, Haider Hameed, M. Zubair Shafiq, Muddassar Farooq, “Using Spatio-Temporal Information in API Calls with Machine Learning Algorithms for Malware Detection”, *Proceedings of the 2nd ACM workshop on Security and artificial intelligence (AISec '09)*, pp. 55-62, November 2009.

[97] Younghee Park, Douglas Reeves, Vikram Mulukutla, Balaji Sundaravel, “Fast Malware Classification by Automated Behavioral Graph Matching”, *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '10)*, Article 45, 2010.

[98] Bai, L., J. Pang, Y. Zhang, W. Fu and J. Zhu “Detecting malicious behavior using critical API calling graph matching”, *Proceedings of the 1st International Conference on Information Science and Engineering*, pp. 1716-1719, 2009.

[99] Padmini Jaikumar and Avinash C.Kak, “A graph-theoretic framework for isolating botnets in a network”, *Security and Commination Networks*, vol. 8, no. 16, pp. 2605-2623, November 2015.

[100] Ammar Ahmed E. Elhadi, b,Mohd Aizaini Maarofa, Bazara I.A. Barryc, Hentabli Hamzaa, “Enhancing the detection of metamorphic malware using call graphs”, *Computers & Security*, vol. 46, pp. 62-80, October 2014.

[101] Zhao Z, Wang J, Wang C., “An unknown malware detection scheme based on the features of graph”, *Security and Communication Networks*, vol. 6, no. 2, February 2013.

[102] Eric Uday Kumar, ”User-mode memory scanning on 32-bit & 64-bit Windows”, *Journal in Computer Virology*, vol.6, pp. 123-142, May 2010.

[103] Yan Wen, Jinjing Zhao, Huaimin Wang, ” Implicit Detection of Hidden Processes with a Local-Booted Virtual Machine”, *International Journal of Security and Its Applications*, vol. 2, no. 4, pp. 39-47, 2008.

[104] J.Nick L. Petroni, T.Fraser, J.Molina, W.A.Arbaugh, ”Capilot-a Coprocessor-based Kernel Runtime Integrity Monitor”, *Proceedings of the 13th conference on USENIX Security Symposium (SSYM ’04)*, vol. 13, pp. 13-13, 2004.

[105] S.Forrest,”A sense of self for Unix processes”, *Proceedings of the IEEE symposium on Computer Security and Privacy*, pp. 1-20, 2012.

[106] Andrei Lutas Adrian Colesa, Sandor Lukacs Dan Lutas, “U-HIPE: hypervisor-based protection of user-mode processes in Windows”, *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 1, pp. 23-36, 2016.

[107] Shabnam Aboughadareh, Christoph Csallner, Mehdi Azarmi, “Mixed-Mode Malware and Its Analysis”, *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW-4)*, pp. 1-12, 2014.

[108] Kota Yoshizaki, Toshihiro Yamauchi, “Malware Detection Method Focusing on Anti-Debugging Functions”, *Proceedings of the Second International Symposium on Computing and Networking (CANDAR)*, 2014.

[109] Karla Saur, Julian B.Grizzard, ”Locating x86 paging structures in memory images”, *Digital Investigation*, vol. 7, no. 1-2, pp. 28-37, 2010.

[110] Andreas Schuster, ”Searching for processes and threads in Microsoft Windows memory dumps”, *Digital Investigation*, vol. 3, pp. 10-16, 2006.

[111] Burdach Mariusz, *An introduction to Windows Memory Forensic*, 2005.

[112] Betz Chris. MemParser, [Online] Available: http://www.dfrws.org/2005/challenge/memparser.html

[113] Garner George M, Mora Robert-Jan. Kntlist, [Online] Available: http://www.dfrws.org/2005/challenge/kntlist.html

[114] H.Yin, Z.Liang, D.Song, “HookFinder: Identifying and Understanding Malware Hooking Behaviors”, *Proceedings of the Annual Network and Distributed System Security Symposium*, 2008.

[115] Z.Wang, X.Jiang, W.Cui, X.Wang, “Countering persistent Kernel Rootkits through Semantic Hook Discovery”, *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pp. 21-38, 2008.

[116] Y.M.Wang, D.Beck, B.Vo, R.Roussev, C.Verbowski, ”Detecting Stealth Software with Strider GhostBuster”, *Proceedings of the International conference on Dependable Systems and Networks*, pp. 368-377, 2005.

[117] Windows Sysinternals RootkitRevealer. [Online] Available: http://technet.microsoft.com/bb897445.aspx.

[118] Garfinkel, Tal, and Mendel Rosenblum, ”A Virtual Machine Introspection Based Architecture for Intrusion Detection”, *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[119] Stephen T. Jones, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, “VMM-based hidden process detection and identification using Lycosid”, *Proceedings of the fourth ACM SIGPLAN/SIGOPS International conference on Virtual execution environments*, pp. 91-100, 2008.

[120] Toby J. Richer, Grant Neale, Grant Osborne, “On the Effectiveness of Virtualization Assisted View Comparison for Rootkit Detection”, *Proceedings of the 13th Australian Information Security Conference (AISC 2015)*, pp. 35-44, 2015.

[121] Xiongwei Xie, Weichao Wang, “Rootkit Detection on Virtual Machines through Deep Information Extraction at Hypervisor-level”, *Proceedings of the 4th International Workshop on Security and Privacy in Cloud Computing*, pp. 498-503, 2013.

[122] Desheng Fu, Shu Zhou, Chenglong Cao, “A Windows Rootkit Detection Method Based on Cross-View”, *Proceedings of the 2010 International Conference on E-Product E-Service and E-Entertainment (ICEEE)*, pp. 1-3, 2010.

[123] Fu-Hau Hsu, Chang-Kuo Tso, Yi-Chun Yeh, Wei-Jen Wang, and Li-Han Chen “BrowserGuard: A Behavior-based Solution to Drive-by-Download Attacks*”, IEEE Journal on Selected Areas in Communications*, vol. 29, no. 7, pp. 1461–1468, August 2011.

[124] Arati Baliga, Vinod Ganapathy, and LiviuIftode, ”Detecting Kernel-Level Rootkits Using Data Structure Invariants”, *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no.5, pp. 670–684, September-October 2011.

[125] Hung-Min Sun, Hsun Wang, King-Hang Wang, Chien-Ming Chen, ”A Native APIs Protection Mechanism in the Kernel Mode against malicious Code”, *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 813–823, June 2011.

[126] Jiayuan Zhang, Shufen Liu, Jun Peng, Aijie Guan, “Techniques of user-mode detecting System Service Descriptor Table”, *Proceedings of the 13th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2009)*, 2009.

[127] Zahra Salehi, Ashkan Sami, Mahboobe Ghiasi, “Using feature generation from API calls for malware detection”, *Computer Fraud & Security*, vol. 2014, no. 9, pp. 9-18, 2014.

[128] Lynette Qu Nguyen, Tufan Demir, Jeff Rowe, Francis Hsu, Karl Levitt, “A framework for diversifying Windows native APIs to tolerate code injection attacks". *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pp. 392-394, 2007.

[129] Malware Samples [Online] Available: http://www.offensivecomputing.net/

[130] Malware Samples [Online] Available http://vx.netlux.org/

[131] BlackLight, [Online] Available: http://www.majorgeeks.com/mg/getmirror/icesword,2.html

[132] IceSword v 1.22. [Online] Available. https://icesword.jaleco.com

[133] VICE [Online] Available: http://www.downloadcollection.com/freeware/vice-rootkit.htm

[134] R3 Hook Scanner 1.6 [Online] Available http://www.pcadvisor.co.uk/download/security/ring3-api-hook-scanner-16-3328574/

[135] J.Butle, S.Sparks,Windows rootkits of 2005, [Online] Available: www.securityfocus.com/infocus/1854

[136] GMER V1.0.15.15087. [Online] Available: www.nonags.com/freeware-gmer-3786.html

[137] HeliosLite [Online] Available: http://www.xfocus.net/

[138] Rootkit Unhooker [Online] Available: http://www.download.com

[139] HiddenFinder [Online] Available: https: http://hiddenfinder.jaleco.com/

[140] M.Harchol Balter and A.B.Downey, “Exploiting process lifetime distribution for dynamic load balancing”, *ACM Transactions on Computer Systems*, vol.15, no. 3, pp. 253-285, 1997.

[141] Malware Collection, [Online] Available: http://www.xfocus.net/

[142] Rookit malware Samples, [Onlne] Available: http://www.rootkit.com/

[143] Malware Collection, [Online] Available: www.download.com

[144] Benign Samples Collection, [Online] Available: technet.microsoft.com

[145] Windows Driver Developmet toolkit, [Online] Available: http://www.microsoft.com/download/en/

[146] Redline, [Online] Available: www.mandiant.com/product/free-software/redline/

[147] PcMark8 Bechmark tool [Online] Available: http://www.passmark.com/

LIST OF PUBLICATIONS

Journals

[1] K. Muthumanickam and E. Ilavarasan, "PAM: Process Authentication Mechanism for Protecting System Services against Malicious Code Attacks", *ETRI Journal*. (Under Review)

[2] K. Muthumanickam and E. Ilavarasan, "CoPDA: Concealed Process and Service Discovery Algorithm to Reveal Rootkit Footprints", *Malaysian Journal of Computer Science*, vol. 28, no. 1, pp. 1-15, March 2015.

[3] K. Muthumanickam and E. Ilavarasan, "An Effective method for protecting native API hook attacks in User-mode", *Research Journal of Applied Sciences, Engineering and Technology*, vol. 9, no. 1, pp. 33-39, March 2015.

[4] K. Muthumanickam and E. Ilavarasan, "Optimization of Rootkit Revealing System Resources - A Game theoretic Approach", *Journal of King Saud University-Computer and Information Sciences*, vol. 27, no. 4, pp. 386-392, October 2015.

[5] K. Muthumanickam and E. Ilavarasan, "Demanding Requirement of Security for Wireless Mobile Devices-A Survey", *Research Journal of Applied Sciences, Engineering and Technology*, vol. 8, no. 24, pp. 2381-2387, December 2014.

[6] K. Muthumanickam and E. Ilavarasan, "Enhancing Malware Detection Accuracy through Graph Based Model", *British Journal of Mathematics & Computer Science*, vol. 4, no. 15, August 2014.

[7] K. Muthumanickam, E. Ilavarasan and Sanjeev Kumar Dwivedi, "A Dynamic Botnet Detection Model Based on Behavior Analysis", *International Journal on Recent Trends in Engineering & Technology*, vol. 10, no. 1, pp. 104-111, January 2014.

[8] K. Muthumanickam and E. Ilavarasan, ”Automatic Generation of P2P Botnet Network Attack Graph”, *Lecture Notes in Electrical Engineering* , vol. 150, pp. 367-374, 2013.

Internal Conference

[1] K. Muthumanickam and E. Ilavarasan, “Behavior based Authentication Mechanism to Prevent Malicious Code Attacks in Windows”, *Proceedings of the 2015 International Conference on Innovations in Information Embedded and Communication Systems (ICIIECS)*, March 2015.

[2] K. Muthumanickam and E. Ilavarasan, ”P2P Botnet detection: Combined host- and network-level analysis”, *Proceedings of the Third International Conference on Computing Communication & Networking Technologies (ICCCNT)*, pp. 1-5, July 2012.

[3] K. Muthumanickam and E. Ilavarasan, ”Automatic Generation of P2P Botnet Network Attack Graph”, *Proceedings of the Third International Conference on Trends in Information, Telecommunication and Computing*, vol. 150, pp. 367-373, September 2013.

[4] K. Muthumanickam and E. Ilavarasan, ”A Survey on host-based Botnet identification”, *Proceedings of the International Conference on Radar, Communication and Computing (ICRCC)*, pp. 166-170, December 2012.

National Conference

[1] K. Muthumanickam and E. Ilavarasan, “P2P Botnet Detection by Correlating Network Behaviors and Host Behaviors: A Study”, *Proceedings of the National Conference on Internet and Webservice Computing (NCIWSC-12)*, August 2012.

VITAE

K.MUTHUMANICAKM, the author of this thesis is part time research scholar in the Department of Computer Science and Engineering at Pondicherry Engineering College, Puducherry, India. He was born in May 1975 at India. He received her Bachelor's degree in Computer Science and Engineering from the University of Madras in the year 1997. He started his teaching life as a Lecturer in the Department of Computer Science and Engineering at Vinayaka Missions Kirupanada Variyar Engineering College, Tamilnadu, India. Later, he joined as a Lecturer in the Department of Computer Science and Engineering in Arulmigu Meenakshi Amman College of Engineering at Tamilnadu, India for more than three years. Presently he is working as Assistant Professor in the Department of Computer Science and Engineering at Arunai Engineering College, Tamilnadu, India. He completed his Master's degree in the same major in the year 2007 from the Anna University. Subsequently he promoted as an Assistant Professor in the same department for more than nine years. He has published more than twenty papers in the International Journals and Conferences. His area of specialization includes Computer Security, Information Security and Computer network.