REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*
3/1/2012

2. REPORT TYPE
FINAL REPORT

3. DATES COVERED *(From - To)*
February 1, 2009 – November 30, 2011

4. TITLE AND SUBTITLE
Anomaly-Based Intrusion Detection Systems Utilizing System Call Data

5a. CONTRACT NUMBER
FA9550-09-1-0067

5b. GRANT NUMBER
49527

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Skormin, Victor A.

5d. PROJECT NUMBER
1077358

5e. TASK NUMBER
1

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Binghamton University
(State University of New York)

Watson School of Engineering
Electrical & Computer Engineering
4400 Vestal Pkwy East
Binghamton, NY 13902

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research

Suite 325, Room 3112
875 N. Randolph Street
Arlington, VA 22203-1768

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Available to general public

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This research aims at the enhancement of computer defenses by making them invulnerable to new, mutating and obfuscated malware. It offers a semantic approach to system behavior analysis, centered on the concept of functionality. Functionality is the highest level of the behavior semantics, it is defined by the specific goal of computer operations, not by its software realization. This allows for identifying some classes of malware achieving the same specific malicious operations. Colored Petri nets are proposed as a basis for behavioral signatures representing particular functionalities, both legitimate and malicious. Special techniques are proposed to address three interrelated aspects: signature expressiveness, behavioral obfuscation and run-time signature matching efficiency. A signature based approach for detecting malicious functionalities in the system call domain is developed. It has been implemented in a prototype software and tested. It is superior to existing behavior based techniques in addressing behavioral obfuscations and multiple functionality realizations. The experiments results indicate low rate of false positives and negatives, and low execution overhead. Such results suggest that detecting malicious functionality presents a sufficiently dependable and efficient method for distinguishing malware from benign software.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					Dr. Victor Skormin
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU	28	**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U			607-777-4013

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

List of figures ... iv
List of tables ... v
1 Introduction ... 1
2 Modern Security Challenges to Computer Systems ... 4
2.1 Description and Classification of Modern Malware ... 4
2.1.1 Viruses ... 4
2.1.2 Worms ... 4
2.1.3 Trojans ... 5
2.2 State of the Art of Anti-Virus Technology: Limitations and Disadvantages ... 5
3 Taxonomy of Malicious Functionalities ... 9
3.1 Definition of Software Functionality ... 9
3.2 Taxonomy of Malware Replication Functionalities ... 10
3.2.1 Binary Self-Replication ... 10
3.2.2 Network Based (Server-Side) Self-Replication ... 11
3.2.3 Client Side Self-Replication ... 12
3.3 Taxonomy of Malware Payload Functionalities ... 14
3.4 Functionalities of Next Generation Malware ... 16
3.4.1 Targeted Attack ... 16
3.4.2 Behavioral Metamorphism ... 17
3.5 Conclusions ... 17
4 Signature Based Behavioral Detection ... 18
4.1 State of the Art in Signature Based Behavior Detection ... 20
4.1.1 State-Transition and CPN Specifications ... 20
4.1.2 Declarative and Algebraic Specification Languages ... 21
4.1.3 System Call Domain Specifications ... 23
4.2 Functionality Definition and Specification ... 24
4.2.1 Formalization of the Specification ... 24
4.2.2 Specification Abstraction ... 28

4.3 Behavioral Obfuscation 29
4.3.1 Behavior Obfuscation Techniques.. 29
4.4 Specification Generalization, Anti-Obfuscation 30
4.5 Functionality Recognition.. 39
4.5.1 Justification of the Recognition Model with Respect to Expressive Power.... 39
4.5.2 Justification of the Recognition Model with Respect to Computational Complexity. 40
4.5.3 Dynamic Information Flow Tracing 45
4.6 IDS Implementation 47
4.6.1 AD Designer 47
4.6.2 Specification Generalizer and CPN Constructor 47
4.6.3 Functionality Recognizer.. 48
4.6.4 Taint Propagation Engine 48
4.7 Conclusions.. 49
5 Evaluation of the Developed IDS 50
5.1 Experimental Setup.. 50
5.2 Detection Results 51
5.2.1 False Positives 51
5.2.2 False Negatives (Detection Rate). 55
5.2.3 Case Study - Password Stealing 56
5.3 Performance Overhead Evaluation 58
5.3.1 Run-time Performance Analysis.. 58
5.3.2 Stress test 60
5.4 Conclusions.. 61
6 Summary and Future Work 62
References.. 65
Appendix A – AD formalization.. 68
Appendix B - Remote IPC Ads.. 71
Appendix C - Generalization functionalities AD.. 73

Appendix D - Functions utilized in generalization algorithms 75
List of Symbols, Abbreviations and Acronyms 76

LIST OF FIGURES

Figure 1. Number of Binary Signatures of Malware (Source: Kaspersky Lab) 6

Figure 2. Detection Rate of Heuristic Analysis (May, 2010) 7

Figure 3. Utilization of Malicious Functionalities (Source: Trend Micro Inc.)..................... 7

Figure 4. Functionality Implementation at the System Level.. 9

Figure 5. Architecture of Proposed IDS 18

Figure 6. Activity Diagram of the “Remote Shell” Functionality 26

Figure 7. File Upload AD 36

Figure 8. Generalized File Upload AD 37

Figure 9. Generalized AD for Remote Shell.. 38

Figure 10. High Level (Subsystem Level) CPN for the “Remote Shell” Functionality...... 43

Figure 11. Low Level (system call level) CPN for the “Remote Shell” Functionality 44

Figure 12. High Level CP-subnet for the “Password Stealer” Functionality. 57

Figure 13. Handle Duplication Test.. 61

Figure B-1. Remote IPC- Create Operation... 71

Figure B-2. Remote IPC – Receive Operation... 71

Figure B-3. Remote IPC –Wait Operation.. 72

Figure B-4. Remote IPC – Send Operation 72

Figure C-1. Handle Duplication Functionality 73

Figure C-2. Process Generation Functionality... 73

Figure C-3. Code Injection Functionality 74

LIST OF TABLES

Table 1. Payload Collection 15

Table 2. "Remote Shell" Realization 25

Table 3. Functional Objects for Data Transfer 28

Table 4. Generalization Functionalities 31

Table 5. Source and Sink Operations 34

Table 6. Functionalities Detection Rate and False Positive Rate 52

Table 7. Place Reachability Statistics for Low Level CPN 54

Table 8. Place Reachability Statistics for High Level CPN 55

Table 9. Password Stealing Functionality 56

Table 10. Place Reachability of CPN for "Password Stealer" 57

Table 11. Execution Overhead due to IDS 59

1 INTRODUCTION

Our ever-growing dependence on computer networks is accompanied by ever-growing concerns about the network's vulnerability to information attacks and the dependability of the existing network security systems. Major threats primarily stem from increasingly sophisticated self-replicating malicious software. At present, more than 700 million computers are connected to the Internet and their numbers are growing rapidly [1]. Every year, thousands new public vulnerabilities in security systems are revealed, and at any given moment of time millions of vulnerable computers are connected via the Internet.

Sophisticated adversaries detect and utilize vulnerable computers to carry out various attacks. Many attacks are performed in a completely automated fashion and spread throughout the Internet at the speed of light without regard to geographical and national borders. Technologies utilized by malicious software are becoming more and more complex. In some cases attacks are completely concealed and cannot be revealed without a thorough analysis delaying both detection and mitigation efforts. In many instances attackers intend to compromise computer network security systems, rendering them ineffective. Moreover, poly- and meta-morphism are commonly utilized by attackers to minimize the efficiency of traditional anti-virus software tools that are dependent on gigantic, continuously updated databases. Fortunately, Intrusion Detection Systems (IDSs) utilizing behavioral signatures to match malware activity rather than its binary structure are immune to this binary morphism.

The research presented in this report is aimed at the development of semantic approaches to behavior analysis in a scalable dependable IDS system. It resulted in a signature based IDS approach that was implemented, tested and characterized.

Section 2 discusses the modern malware and associated trends, malware classification, and a review of the limitations of conventional anti-virus technologies. First, various types of modern malware, statistics and emerging trends in malware development are presented. Secondly, this chapter reviews modern anti-virus products and discusses limitations of technologies currently used to detect malware such as binary signatures, heuristic analysis, and behavioral detection.

Section 3 presents taxonomy of the typical basic malware functionalities that could be attributed to the "essence" of malicious activity. In particular, self-replication mechanisms as well as malicious payloads are analyzed. Three types of the self-replication mechanism are discussed, including: binary self-replication, server-side replication, and client-side replication. Moreover, malicious payloads are classified and analyzed, including: persistence mechanisms, delivery and communication mechanisms, data acquire mechanisms, offensive payloads and others.

Among the known malicious activities, self-replication is an example of a highly discriminative and indicative malicious functionality. Obviously, there is no reason for legitimate software to self-replicate, since it can be distributed by legitimate means (e.g. downloads and install, trial etc.). Hence, computer security researchers are very interested in self-replication phenomenology.

It is important to be able to reliably detect self-replication as a specific functionality. However, before detecting it, it is reasonable that we should model self-propagation in order to investigate and estimate the possible impact of this functionality on network resources. Such a model would allow for selecting the most adequate means for both attack detection and mitigation.

Section 4 presents the developed signature based IDS approach. This approach detects malicious functionalities in the system call domain using generic and highly semantic signatures. Such an approach is superior to existing behavior based techniques in addressing behavioral obfuscations and multiple functionality realizations. Basically, the proposed IDS detects intrusions at the highest semantic level – the functional level, which is semantically higher than a simple behavior due to the fact that behavior is merely a manifestation of one of the realizations of functionality that could be obfuscated. While signature-based, behavior-based IDSs (BBIDS) have obvious advantages, however they could suffer from three interrelated problems: poor signature expressiveness, behavioral obfuscation and run-time signature matching inefficiency. The research presented in this chapter describes the development of novel, system call domain IDS that addresses both existing and future challenges of BBIDS. In particularly, we propose an approach to specify the functionalities of interest, specifically malicious ones, by using activity diagrams (AD) in terms of generic behavioral constructs. The resultant AD would incorporate multiple realizations of the specified functionality hence increasing the semantics and expressiveness of the signature. An AD signature would be automatically generalized to address existing and potential behavioral obfuscation techniques. Finally, a procedure is presented that is capable of automatic conversion of activity diagrams into colored Petri nets (CPN) defined in the system call domain, to be used by IDs for run-time recognition of the specified functionalities.

Section 5 presents a comprehensive experimental evaluation of the proposed approaches that were implemented in a prototype IDS. An experimental evaluation of the described technology was conducted on the virtual network testbed at Binghamton University [2], [3]. The testbed was configured as a virtual network comprised of dozens of victim hosts represented by virtual machines with vulnerable versions of Windows1 OS and our prototype IDS. The IDS was evaluated on hundreds of legitimate programs and dozens of malware that had various types of replication engines and payloads. The experimental results indicated extremely low false negatives and false positives. Finally, we performed a series of experiments to determine the run-time overhead induced by the IDS. This overhead was estimated based on established benchmarks. The results indicated that the IDS incurred less than 4% of the CPU utilization overhead.

The results of this research were presented at the following conferences (proceedings) and journals:

1. Military Communications Conference (three papers), MILCOM 2011, Baltimore, MD, USA 2011
2. European Symposium on Research in Computer Security (ESORICS), Athens, Greece 2010.
3. Military Communications Conference, MILCOM 2010, San Jose, CA, USA 2010.
4. International Conference on Security and Management, SAM'10, Las Vegas, NV, July 2010
5. Conference on Future Challenges in Network Security (FCNS), Prague, Check Republic, June 2010
6. International Conference on Security and Management, SAM'09, Las Vegas, NV, July 2009
7. 27th IEEE International Performance Computing and Communications Conference, (IPCCC), Austin, TX, Dec. 2008
8. Problems of Information Security. Computer Systems (Journal), issue 3, Moscow, Russia, 2008 (in Russian)
9. Journal of Information Assurance and Security, vol. 2, issue 2, pp. 107-116, 2007.

[1] Windows is registered trademarks of Microsoft Corporation in the United States and other countries

10. Third International Symposium on Information Assurance and Security (IAS’07), Manchester, England, August 2007
11. International Conference: “Mathematical Methods, Models and Architectures for Computer Networks Security”, Sept. 20, 2007, St. Petersburg, Russia.

2 MODERN SECURITY CHALLENGES TO COMPUTER SYSTEMS

Malicious software presents the most prominent threat to modern computers systems. The first example of malware was a computer virus developed almost thirty years ago for Apple computers [4]. At that time, most of the early computer viruses were limited to self-replication and had no specific payload. Since then malware evolved from academic proof-of concepts in middle 1980s to script-kiddies attacks in late 1990s, and finally to underground market and targeted attacks since 2000 (e.g. Zeus botnet, StuxNet worm). As a result, today malicious software is used to achieve a wide range of adversary goals. This includes everything from remote access to local sensitive data to full control of segments of information infrastructure such as servers, network appliances and even industrial controllers.

This chapter presets general discussion of the modern malware and their associated trends; malware classification; and review of the limitations of conventional anti-virus technologies.

2.1 Description and Classification of Modern Malware

Malicious software (malware) is software that is designed to secretly access a computer system without the owner's informed consent [5]. Malware is a general term meaning a variety of forms of hostile, intrusive, or annoying software or program code.

This section describes various types of modern malware, the associated statistics and some emerging trends in malware development. Malware is classified according to well-known notations and terms formalized by the computer security community (e.g. anti-virus vendors) [6], [7]. Note that this section focuses on malware types only. Further discussion on malicious functionalities and behavioral taxonomy is given in the next chapter.

Today, the cyber-security community distinguishes between the various types of malware depending on their main purpose or intentions [8], some of these are defined and described below.

2.1.1 Viruses

The first use of the term “computer virus” is attributed to Fred Cohen in 1983. Fred Cohen originally defined a computer virus as a program that can "infect" other programs by modifying them to include a possibly evolved copy of itself. Traditionally, viruses only infected the local host executable files during propagation. However, today viruses can self-replicate by infecting many file types such as MS Word files (macro virus), Adobe PDF files (script virus), script files etc. Also virus activity is no longer limited to a local host. A virus can also infect files located on network shares, cloud drives and removable media.

2.1.2 Worms

A worm is a form of self-propagating malware that spreads inter-host over the local/wide area network and also via the Internet. The main difference between a worm and a virus is that a worm transfers its binary image to the victim machine without injecting its code into any host files. While a virus, by definition, proliferates by injecting itself into a host file on the victim machine.

Depending on the propagation vector, worms could be classified into the following categories [7]:

- E-mail worms. This worm sends itself or a link to its image as an e-mail attachment.
- P2P worms. This worm proliferates via public P2P file sharing networks.
- Network worms. This worm propagates to hosts by exploiting vulnerabilities in publically exposed services. Since such a worm does not require user action, it could spread very fast causing large scale epidemics.
- Instant messaging and IRC worms. This worm sends a link to its image via instant messengers or IRC, e.g. ICQ or Skype.

After getting deployed to the remote host, worms often establish various backdoors, disable security tools, and install bot agents. Once compromised, these systems become part of what is known as a "zombie" network.

2.1.3 Trojans

A Trojan horse is malware pretending to be benign or useful software. When activated, Trojans perform unauthorized actions such as collecting, modifying, and forging data. Unlike viruses and worms, Trojans by themselves do not self-replicate. Hence, Trojans are delivered by some type of self-replicating malware as a payload or by so-called downloaders. Also, Trojans can be delivered via a social engineering attack by convincing victims to download and execute their code.

Depending on the actions performed, Trojans fall into the following classes [8]: downloaders, spyware, keyloggers, backdoors, dialers, ransom, proxy, clickers, and adware.

Downloaders are small programs that download and install other malware such as adware and spyware. Some downloaders also configure the OS to run the downloaded malware during system startup.

Spyware is a type of malware that steals sensitive data such as a user's credentials, bank account information, web service passwords etc. Such data is transmitted to adversaries via various channels including FTP, e-mail and even covert channels (e.g. DNS reverse tunnel).

Keyloggers monitor and record various system events such as mouse clicks and pressed keys. Usually, keyloggers are used by spyware to gather a user's credentials.

A backdoors is a specific type of malware that attempts to grant unauthorized external remote access to a host. It is usually achieved by rather simple techniques such as remote shell, but there are some sophisticated backdoors that may themselves have system functionality that enables remote system control. Backdoors are usually used to steal personal information including login details, e-mail addresses etc. Moreover, covert backdoors are frequently used to control botnets made up of compromised zombie machines.

Clickers, Adware, Dialers and Ransom represent malware for profit that forces the user to pay money to adversaries or buy pay-per-use services without a user's consent.

2.2 State of the Art of Anti-Virus Technology: Limitations and Disadvantages

In 1987, the German hacker Bernd Fix presented the first anti-virus software that was able to detect and neutralize the Vienna Virus [4]. Since then, the market for anti-virus products has received much attention, and has led to the development of more than sixty anti-virus products. In spite of such development, for the most part, anti-virus products still rely on old, binary signature based technology to detect malware [9]. This technology detects malicious software by matching file images to binary patterns of all known malware that are stored in gigantic, continuously updated databases.

In an attempt to avoid signature-based detection most sophisticated attackers currently utilize polymorphic and metamorphic worms that are able to continuously mutate, i.e. change their binary composition without changing their malicious payload. As a result, the number of binary signatures of malicious codes that need to be stored and utilized for successful attack detection continues to grow exponentially, see Figure 1 below. This trend indicates that in the near future virus detection will consume a significant percentage of a computer's finite resources. At the same time, reliance on the known binary signatures implies that in principle it cannot detect new, previously unknown malware.

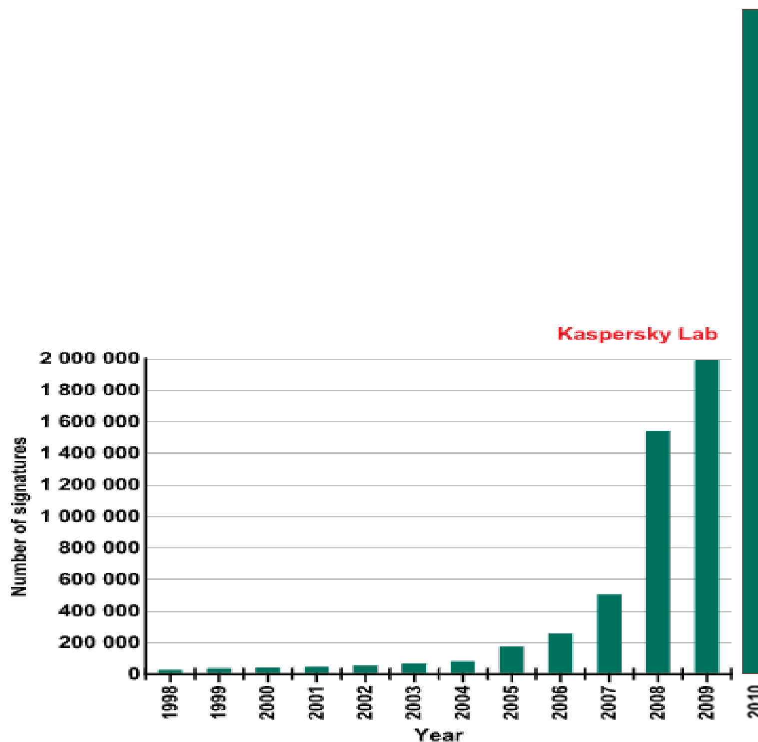

Figure 1. Number of Binary Signatures of Malware (Source: Kaspersky Lab)

To deal with unknown malware, advanced anti-viruses use heuristic analysis that utilizes generic signatures to match classes of malware. However, such technology generates false positives and has rather limited efficiency. Figure 2 shows that the detection rate of heuristic analysis is at most 65% and is only 40% on average [10].

On the other hand, there is an alternative to binary signature-based intrusion detection. Generally, malicious software and infected legitimate software demonstrate specific behavior patterns that are atypical for legitimate software. Some advanced anti-virus products such as Kaspersky, Symantec and ThreatFire perform dynamic analysis of program activity to detect malicious behavior. However, such products usually take into account only individual OS object operations and are also limited to a single process. In other words, such anti-virus products do not correlate the behavior of multiple processes and objects to recognize a complex malicious pattern. As a result, such systems are only able to detect primitive and obvious misuse such as access to a system file, starting a particular process or loading a device driver. While such a behavior could be exhibited by many types of malware, this same behavior is also exhibited by legitimate software (e.g. system tools).

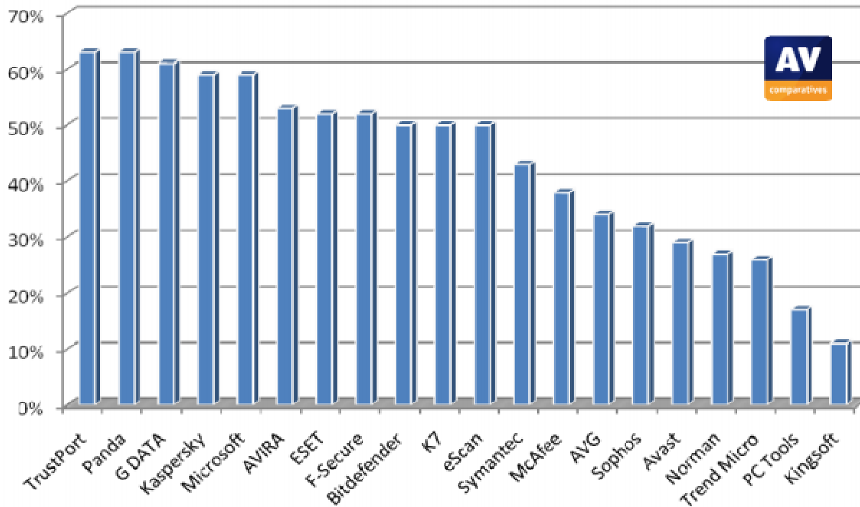

Figure 2. Detection Rate of Heuristic Analysis (May, 2010)

In our observation, behavior by itself is not malicious; however the goals or functionalities of the malware are malicious. It is very important to point out, that the total number of malicious functionalities is fairly stable, and the only thing that changes is how often particular functionalities are utilized (see Figure 3). Inventing a new malicious functionality requires significant effort and is typically beyond the means of most attackers. Consequently, when malicious code mutates, it implements the same malicious functionality in spite of the variations in its binary code. Moreover, developers of new computer worms are destined to utilize the same malicious functionalities again and again (for example self-replication engines).

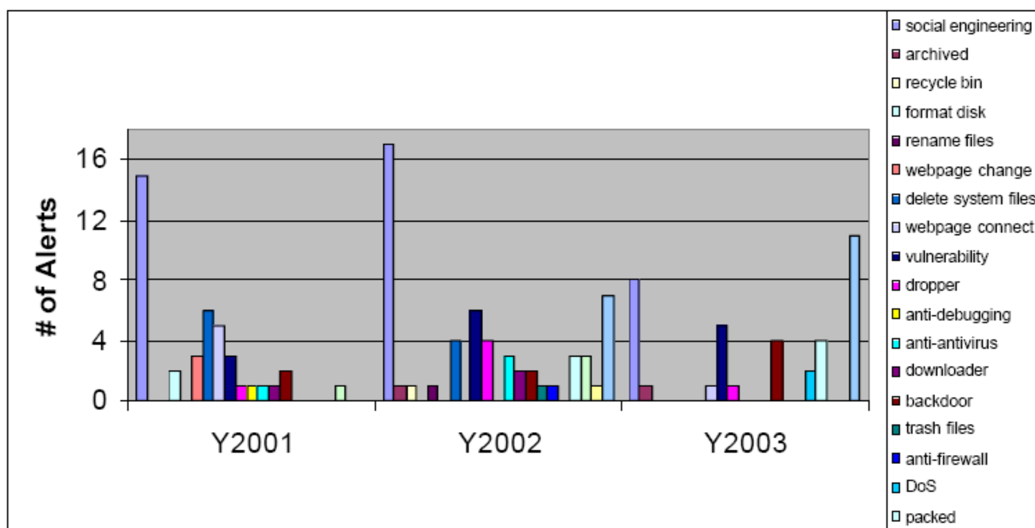

Figure 3. Utilization of Malicious Functionalities (Source: Trend Micro Inc.)

Taking into account the above considerations, it is critical to detect complex functionalities, rather than merely simple behavior. Such functionalities may involve interrelated sessions of object operations. For instance, we propose to detect the following sophisticated and highly semantic functionalities: password theft, multipartite self-replication or self-concealing using third party rootkits. The complexity and high level semantics of these functionalities

allows for confident discrimination of maliciousness in the program activity with minimum false positives.

In order to detect malicious functionalities, first we must define, classify and analyze existing and potential malicious functionalities. The next chapter is dedicated to the taxonomy of malicious functionalities which covers most known and conceivable malicious functionalities that each has a very distinctive footprint.

3 TAXONOMY OF MALICIOUS FUNCTIONALITIES

3.1 Definition of Software Functionality

Let us describe the software behavior and functionality execution in Microsoft Windows™ operating system (OS). This OS provides system recourses and services to processes through executive objects that are maintained in the Kernel (Figure 4). In order to access a particular resource or service a process creates a corresponding object such as a file, process, thread, memory section, etc [11]. Every object has its own set of operations which are exported to user mode processes through system services (system calls)[2]. In the user mode, such system calls can be invoked directly or more conveniently through subsystem API functions.

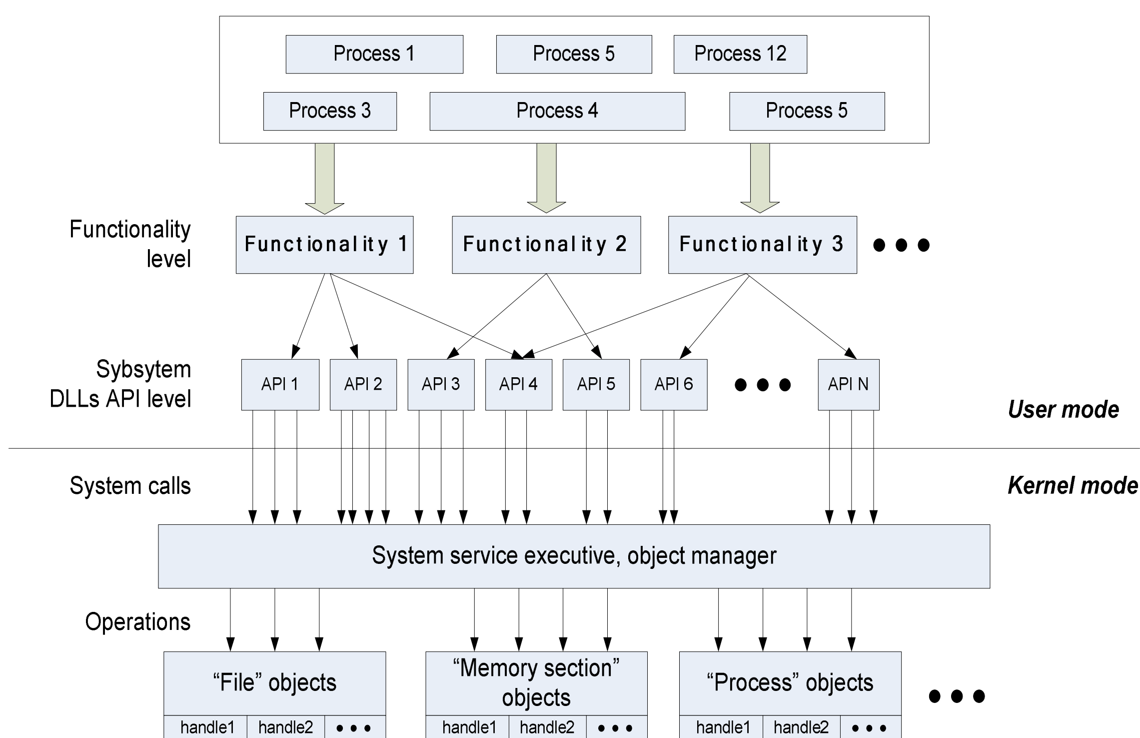

Figure 4. Functionality Implementation at the System Level

Processes invoke API functions or system calls in order to perform object operations (manipulations)[3] that complete some semantically distinct action(s), such as writing data to a file or sending data to a specified IP address. Consequently, we define individual functionality as a combination of those actions that achieve a certain high-level objective.

[2] In Unix based systems such services are called system calls, while in Windows they are called executive system services. Hereafter, we stick to system call term.

[3] Here, we use terms "Operation" and "Manipulation" interchangeably, because both of the terms are used extensively in the literature.

It is important to understand the difference between a functionality and behavior. The behavior of a process is what the process does at a given stage, while the functionality determines the semantic goals of the process. In other words, behavior simply manifests the realization of functionality. As a result the major limitation of the existing behavior-based specifications is that they can fail when dealing with multiple realizations of the same functionality. This motivated us to develop a novel specification free from this limitation.

3.2 Taxonomy of Malware Replication Functionalities

Today, major threats primarily stem from self-replicating malicious software such as worms and file viruses. Self-replicating malware has two basic components, a payload and a replication mechanism. The payload functionalities normally perform some malicious activity on the victim host, while the replication mechanism ensures its most essential feature, the self-replication. While any set of functionalities can constitute a payload, most attackers inherently rely on existing self-replication engines.

Self-replicating malware can be divided into three major classes: network worms, e-mail worms (trojans) and file viruses [12], [13]. While the replication mechanism is usually determined by the type of malware, some malware instances utilize multiple replication mechanisms; however, these malware does not introduce a conceptually new type of self-replication.

Based on the attack vector, one can distinguish between the following replication mechanisms:

- Binary self-replication (used by file viruses)
- Network based (server-side) self-replication (used by network worms)
- Client side self-replication (used by trojans and e-mail worms)
- These replication mechanisms are described in details in the following sections.

3.2.1 Binary Self-Replication

File virus replication mechanisms were analyzed in [3]. There are basic types of virus replication techniques:

- Overwriting existing files (Overwriting viruses)
- Creating new look-alike files (Companion viruses)
- Attaching to existing files (Parasitic viruses)
- Injecting itself to empty spaces in PE (Cavity viruses)

An Overwriting virus replaces existing executable or its code section with the body of the virus. A Companion virus renames (or moves) an existing executable and replaces the original with a copy of itself. The virus then runs the victimized binary after executing its own body. A Parasitic virus infects an existing executable by injecting its code into the executable body and replacing code entry points. Cavity virus can inject itself to the PE header or unoccupied portions of PE segments. Such a virus does not change the overall length of the host file.

The authors in [3] proposed tracking semantically primitive functional blocks such as file search, memory mapping, file copy etc., and to then attribute their combinations to a so-called Gene of Self Replication (GSR). However, a GSR is only defined for file viruses and it does not allow for tracing alternative realizations. Moreover, due to the token dynamics, the CP-net [14] recognition mechanism is more efficient than the one proposed in [3], i.e. state machines.

3.2.2 Network Based (Server-Side) Self-Replication

Servers connected to the Internet usually expose services that clients can interact with. Moreover, most hosts in a local network provide several publically accessible services that potentially could be compromised. This includes services such as RPC DKOM, Remote Desktop etc. After the appearance of the first high-profile network worm in 2001 (Code-Red) that exploited a vulnerability in a server service (MS IIS) server-side vulnerabilities became the most convenient attack vectors for self-replicating malware such as network worms including Sasser, Welchia, Blaster etc. [15].

To estimate tendency of network worm generation and propagation engine utilization in modern worms, we analyzed 25 network worm families including: Sasser, Welchia, Blaster, Slammer and Mytob. The propagation engine type for each worm was determined based on the anti-virus databases as well as by reverse engineering and analyzing particular strains of the worms. It was observed that more than 60% of the worms use the "bind shell" engine. "Reverse shell" and "executable download and execute propagation" engines were shared by 30% of the worms. Less than 10% of the worms utilize other types of the engines such as thread injection, remote command execution, and others.

A typical "Bind shell" engine opens a network socket (port) and listens to the socket until the intruder is connected to the port. Then the connection is accepted and the shell code starts a command interpreter, for instance "cmd.exe", such that inputs and outputs are tied to the socket. These actions cause the "cmd.exe" process to listen for commands and execute them. The previous three steps are performed by the exploited process as shell code. The final steps thencomplete the propagation. It could be seen that the attacking host simply passes commands to make the victim host download worm executable code and run it.

The "reverse shell" engine is very similar to the bind shell. However, in order to avoid a firewall, the shell code automatically connects to the intruder, and instead of waiting for a connection from the intruder. From this point on, the remaining steps are identical.

The "executable download and execute" engine perform the entire propagation in the shell code without post-activity as is performed in the first two engines. In this case, it simply creates a socket, establishes a connection to the intruder and retrieves a copy of the worm through the established channel. The shell code usually uses high level protocols such as http to download the worm, but sometimes it downloads a worm directly through the channel simply using transmission control protocol (TCP).

The above considerations indicate that worms from the same family tend to share the same propagation engine(s). The number and type of propagation engines is limited. In the first stage of the propagation session, the worm shell code is executed by the compromised process. To achieve the propagation effect, the shell code has to utilize system resources via various API functions. As a result, each type of shell code has its own system call execution pattern.

A new trend in shell code development can be attributed to so-called client side (one way) shell codes that provide access to the victim's machine with a minimized system footprint. Such shell codes include: find socket, reuse socket, DNS reverse tunnel, and HTTP reverse tunnel [16].

Find and reuse socket shell codes do not create a new socket, they simply utilize the socket of the existing connection through which the victim process was exploited. These shell codes are implemented similarly to the standard Bind-Shell with the difference being that the socket is not created, but instead the code enumerates handle values and checks the remote port through the "GetPeerName" API. At the system call level the GetPeerName API is recognized as a command

to the AFD.sys driver with specific control code executed via the NTDeviceIOControlFile system call.

The DNS reverse tunnel shell code implements a hidden channel through which the worm image is transmitted to the victim machine. In this case no dedicated connection is established, and no file or object download is performed (e.g. via HTTP). The only activity the victim host exposes is making DNS requests. These requests are similar to those that are frequently performed by any host connected to the Internet. After the shell code makes a DNS request to a malicious DNS server, the server sends a fake DNS response containing a small piece of worm's binary code to the shell code. Then the shell code receives the worm image piece by piece, decodes it, reconstructs the code and eventually executes it.

3.2.3 Client Side Self-Replication

Historically, the majority of network worms propagate through server-side vulnerabilities, but over the recent years it has become increasingly difficult to use server-side attack vectors for the following reasons:

- A move towards secure/hardened implementations of system and production software (less vulnerabilities on servers)
- More frequent patch cycles for publicly accessible services
- Prevention of remote code execution in critical services using techniques such as DEP4 and ALSR5 [17]. Avoiding these techniques (e.g. return-oriented programming, heap spray) significantly increases exploit and shell code development time [18]
- Utilization of network-based intrusion detection systems.
- Network fragmentation that builds a defense in depth (firewalls, external vs. internal network vs. DMZ)

On the other hand, client-side attacks are quite a different story. These attacks exploit vulnerabilities in client applications, such as web browsers and office application suites, that process malicious data from servers. Today, client-side vulnerabilities are the most popular entry points for attacks. This is supported by the fact that, the vast majority of vulnerabilities (more than 93%) that have been exploited in recent years have been on the client-side [19]. These vulnerabilities are located in various applications, such as web browsers, file readers, office suites and ActiveX components. As a result, in recent years majority of the malware has been propagating via the following web attack vectors [20]:

- Browser vulnerabilities
- Adobe Flash vulnerabilities
- ActiveX vulnerabilities
- Adobe Acrobat Reader vulnerabilities
- Apple QuickTime vulnerabilities

---

[4] DEP – Data Execution Prevention is Microsoft implantation of "Write xor Execute" feature that does not allow for executing injected code (e.g. potentially stops buffer overflow exploit).

[5] ASLR – Address Space Layout Randomization. Process loader technique (feature) which arbitrary arranges the positions of key data areas, usually including the base of the executable modules (dll entry points) , heap, and stack, in a process's address space. ASLR prevents proper execution of an injected shell code. This technique was utilized by Microsoft for system processes containing critical services. User applications (e.g. MS Internet browser) did not use such technique due to compatibility reasons.

- 3rd-party plugins, widgets/gadgets, banner ads
- RealPlayer vulnerabilities

Since client-side attacks affect individual workstations that are located inside the perimeter protected by network security appliances such as firewalls and NIDS, such an attack could be viewed as an insider threat. Some attacks, such as drive-by install, represent normal behavior and are initiated by an unwary user. After obtaining control over the exploited client application, the attacking malware (shell-code) has enough system privileges to propagate itself.

A client-side exploitation can be different from the classical server-side exploitation. This does not necessarily mean that remote code execution (e.g. buffer overflow), will occur, for instance a JavaScript could "legitimately" use an ActiveX component to overwrite a file that is scheduled to be executed by the OS itself. However, the functionality of the shell code chould be identical – namely, the delivery of malware.

Due to the isolation of the workstation from the external network (i.e. firewalls, NIPS), client-side shell code is destined to use normal connections, protocols and services that are allowed by the network/host security policies. For instance, in the case of a web-browser attack vector, the shell code can use the same http session, that is used for normal data exchange, to transfer the malware binary. The client-side delivery mechanisms could be roughly classified as follows:

- Drive-by install (social engineering attack)
- Drive-by write
- Drive-by download

Drive-by install is an example of a social-engineering attack requiring cooperation from an unwary user [21]. Usually, a drive-by install is used for delivering special types of malware such as spyware and adware. This malware can be distributed using various scenarios including: convincing a user to download malware (e.g. fake flash player) from the malicious web-site (e.g. continual prompting); sending an email with malware attached; as a bundled and chained install with some other third party software; and finally via peer-to-peer installation.

Generally, a drive-by install delivery is performed through a self-mailing engine that is used by e-mail worms that may have Trojans in their payloads. In this case, a typical self-mailing engine constitutes a particular functionality that performs at least two essential tasks:

- Load the malware image into memory
- Send the image as an attachment with the e-mail to the victim address

While image reading (loading) is a trivial task performed by a standard subsystem API, sending an e-mail requires using a mail protocol such as: SMTP, ESMTP or SMPT-AUTH. The e-mail consists of a header and a body that may contain an attachment formatted according to the MIME standard. The message is sent to a mail relay server (or MTA) through SMTP. While message formatting does not involve any system calls (only simple memory manipulations), message sending utilizes network resources exported through an API (ws2_32.dll). While the SMTP functionality performed by any Mail User Agent (e-mail client) or MTA (mail exchange server) is absolutely legitimate, such software is not supposed to send its image as an attachment with the message that is being sent.

The drive-by write delivery mechanism may use special vulnerabilities that allow for saving a file (e.g. picture). For instance, a shell-code may directly overwrite or infect a system file using JavaScript and vulnerable ActiveX exports. Another technique would be using

ActiveX components to modify the windows registry and execute malware. Clearly, such a technique requires having a malware binary that itself could be partially contained in the exploit data and then later be reconstructed by the shell code.

Drive-by download propagation implies transferring the malware image using various available mechanisms. The shell code can use ActiveX (JavaScript) to download malware, or retrieve an image via an http channel using a high level API such as *InternetOpenURL* and *InternetReadFile* (wininet.dll). However, to physically transmit the file, all such high level functions eventually utilize standard API and system calls such as: socket, connect, send, recv, (ws2_32.dll).

Another technique for downloading the malware, under the conditions of strict connectivity policy, is to utilize a so-called covert channel. A covert channel implies transferring data using system mechanisms that are not blocked or checked by existing network security appliances (e.g. firewalls, NIDS, web proxies) [16]. Such covert channels could be arranged by using already existing connections, for instance, finding and reusing an opened socket (i.e. socket reuse shellcode). Another way could be to use a DNS service to establish a so-called DNS reverse tunnel that was described previously. Analogously, one could use an ICMP reverse tunnel that works by sending an echo request packet to a remote malicious host. The remote host replies with another ICMP packet that contains a small part of malware binary code. This way a client can receive the entire malware image, piece by piece.

3.3 Taxonomy of Malware Payload Functionalities

Recent attacks on information systems demonstrate a steady increase in the quality and sophistication of newly deployed malware. To be effective, host defense technologies such as an intrusion defense/detection systems (IDSs) must cope with continuing advances in intrusive technologies. The net result is an escalating "arms race" between malware and the IDSs. Generally, malicious technology develops along two major directions: delivery mechanisms (new attack vectors, propagation hiding etc.) and persistence mechanisms, such as self-concealment or self-protection. While the delivery mechanism is a critical part of malware, self-propagation based delivery can successfully be detected and prevented as described in papers [3], [2]. A substantial part of today's malware does not self-propagate, instead it is delivered by downloading and running camouflaged Trojans. These are typically initiated by the legitimate behavior of an unwary user. Hence, detection of payload functionalities including the persistence mechanism becomes extremely important.

After getting deployed on the victim host, malware runs its payload functionalities. Usually its payload corresponds to the malware type, e.g. downloader, dropper or backdoor. In our view, payload functionalities could be distinguished by their purpose according to the following categories:

- Persistence mechanism (self-concealment, self-protection, etc.)
- Delivery and communication mechanism (commands, software download, etc.)
- Data collection / acquisition mechanism (key-logger, network sniffer, etc.)
- Offensive payload
- Other

Malware persistence mechanisms have received significant attention since they allow for concealing and protecting deployed malware. Such technology represents a major challenge for

the IDS since it impedes the intrusion response. Table 1 presents the most popular persistence mechanisms including User level [22] and Kernel level techniques [23].

The delivery and communication mechanism is usually used by network bot agents to relay and execute control commands to a victim host. They are also used to fetch new updates for deployed malware. Some standard mechanisms are presented in table 1.

The data acquisition mechanisms are typically used by spyware to mine the user's sensitive information (e.g. user credentials) or user/host behavioral statistics (web surfing). This information is then transferred to the adversary via the communication mechanism.

The offensive payload is the most malicious payload, andcan cause significant damage such as denial of network/server services, subversion of anti-malware services and windows blocker (back-mail malware). Table 1 summarizes the most infamous offensive payloads.

Table 1. Payload Collection

Payload category	Functionality	Description
Persistence mechanism	Mimicry technique	Camouflage malware image: • renaming its image • appending its image to victim legitimate image • external restructure of the image
	Injection based	Inject malicious code into victim process: • Thread injection • Dll injection • Event spoofing
	Multipartite approach	Multipartite approach: • Watchdog agents • Inter-process distribution of the activity
	Root persistence	Kernel level techniques: • System call spoofing (SSDT substitution, malware resources hiding) • Non direct object manipulation (reparse point, shared memory) • DKOM (process hiding, driver concealment)
	OS service subversion	Makes changes in the system to render security and administrative services useless: • Resets system restore points • Disable various security services (security, update, error reporting, firewalls and anti-viruses) • Spoof DNS service to block security content
	Registry setup	Makes modifications in registry to arrange autorun of the malware image or compromise host security
Delivery and communication mechanism	Dropper	Creates small binary which contains selected payloads and is not able to proliferate.
	Backdoor	Provides remote unauthorized access to the host. This makes the host a "zombie" machine. To avoid firewalls, such mechanism may use several channels such as direct TCP, covered ICMP, covered DNS etc.
	System call proxy	Executes system calls with attributes received from C&C center
	Downloader	Periodically downloads update malware
	Spam engine	Forms and massively sends e-mails potentially to contacts from local address book.
Data acquire mechanism	Credential stealer	Hidden module which monitors user's activity to steal user credentials
	Key-logger	Intercepts and logs key strokes
	Traffic sniffer	Records packets, acquires network statistics
	Web sniffer	Records web surfing statistics
Offensive	DoS	Performs Denial of Service (including OS lock)

payload	Auto-dialer	Dials expensive phone calls causing financial loss for the victimized user
	Windows blocker	Denies access to OS and data until a user pays a ransom
Other	Adware	Added harmless (commercial) functionality in third party environment, e.g. toolboxes in IE.
	Rouge	Fake anti-virus activity

3.4 Functionalities of Next Generation Malware

3.4.1 Targeted Attack

A new trend in malware development is highly professional and targeted attacks that achieve very specific goals. One such example is the StuxNet worm that was the first publically known computer worm that targeted a specific industrial systems/processes. The ultimate goal of the worm is to alter/reprogram the control processes of a particular industrial plant. Exactly which one was targeted still remains unknown, however a majority of the attacks took place in Iran [24].

Due to the target specific nature of the worm, it employs several functionalities to conceal its activity and to help it keep a low activity profile. Such functionalities include:

Controlled self-propagation that is achieved by limiting the number of generations of the worm – this prevents burst-like epidemics, helping it keep a low profile.

The use of multiple attack vectors to self-propagate via USB removable drives and local network. The range of 0-day vulnerabilities that the StuxNet exploits allowed it to proliferate over various Windows™ OSs including Windows 2000, XP, Vista and Windows 7 [25].

The use of rootkits to hide the worm binaries. To remain stealthy the rootkits are digitally signed with valid, but stolen certificates.

The use of centralized command and control (C&C) server and a decentralized peer-to-peer (P2P) mechanism to allow infected machines to request updates.

The targeting nature of the worm is evident due to the fact that the worm uses various techniques to precisely identify the system to be attacked. The target consists of specific Siemens hardware and software, such as certain Programmable Logic Controllers (PLCs) (6ES7-315-2 and 6ES7-417), and the WinCC SCADA (Step7 Sematic Manger component)[6].

Another functionality is the modification of the WinCC SCADA communication library (s7otbxdx.dll). This allows a targeted PLC to be infected by injecting worm's blocks of STL code and also hides the injected code from human operators.

The injected malicious STL code basically changes the PLC behavior. In particularly, the malicious code checks for incoming data to PLC (could be sensor data) and may modify and post-process the data [24]. Theoretically, this may alter feedback response of the control system itself that may drive the plant to unstable phase and eventually physical damage.

It is interesting to note that a particular block of code - block DB8061 is automatically generated during the infection phase and its contents depends on the data that is in targeted PLC [24]. Analysis revealed that this particular block of data is not present in the StuxNet and is

[6] PLC is a stand-alone industrial class digital computer that controls equipment of a plant to maintain industrial processes along with desired specifics. A Sematic PLC runs assembly like code that controls the plant. SCADA system is PC software that allows a human operator to monitor, remotely control plant's equipment as well as to update the code on PLC.

generated when the PLC with a specific code is found. This way the malware authors are able to hide certain behavioral details of the malicious PLC code.

In summary, a targeted attack such as the StuxNet worm not only has a Windows rootkit functionality but also PLC rootkit functionality that is concealed from the human operator. This example clearly indicates that there has been a fundamental shift in the security perimeter such that it is no longer limited to personal computers and has clearly been expanded to include industrial computing facilities.

3.4.2 Behavioral Metamorphism

Today binary morphism is a common feature of modern malware. Given the extensive development of behavioral based IDS, it is expected that adversaries will employ behavioral metamorphism as the next step in offensive information warfare.

Currently, behavioral metamorphism has not received much attention, and as such has not been studied or defined in the literature, however, an approach for behavioral obfuscation is given in [14]. This approach implies the use of several techniques to alter the realization of the given functionality so that it would have a different footprint in the system call domain. These techniques obfuscate behavior by using single process or multiple processes, i.e. a multipartite approach.

From the perspective of the system call domain, this behavioral metamorphism could be viewed as dynamic functionality obfuscation. The difference here would be that the obfuscation is performed at moment of functionality implementation (compilation), and that the metamorphism is performed dynamically during the execution stage.

As with intra-process obfuscation, behavioral metamorphism could be achieved by randomly switching between alternative system mechanisms (e.g. network pipes, sockets, file mapping etc.) to perform elementary operations such as data transfer or file data access.

Similar to multipartite obfuscation [14], another approach for behavioral metamorphism would be dynamic scattering of malicious functionalities among different benign processes so that none of the processes would have a consistent system call pattern.

3.5 Conclusions

This chapter described the basic functionalities of typical malware that could be attributed to the essence of malicious activity. In particularly, we reviewed typical self-replication mechanisms as well as several malicious payloads. Three types of the self-replication mechanism were discussed including binary self-replication, server-side replication and client-side replication. Additionally, malicious payloads were classified and analyzed including persistence mechanisms, delivery and communication mechanisms, data acquisition mechanisms, offensive payloads and others.

Understanding malicious functionality is critical for the successful detection of malware activity. The remaining chapters describe novel technologies for malware detection based on identification of malicious functionalities as anomalies and generic behavioral signatures.

4 SIGNATURE BASED BEHAVIORAL DETECTION

Computer networks, being a critical component of the national infrastructure, are continuously subjected to information attacks. The most devastating attacks are perpetrated by the deployment of self-replicating malicious software that propagates through different media and uses multiple attack vectors. Recent information attacks demonstrate a steady increase in the quality and sophistication of newly deployed malware. To avoid signature-based detection by most commercial IDSs, modern malicious software employ polymorphism and sometimes even metamorphism. Fortunately, IDSs that effectively utilize behavioral signatures to match malware activity, rather than its binary structure are immune to binary morphism.

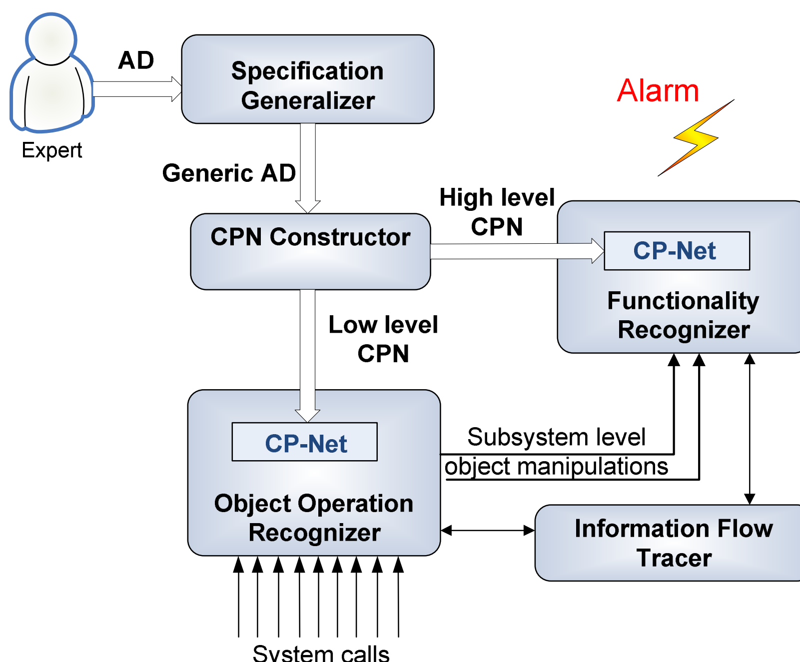

Figure 5. Architecture of Proposed IDS

While behavior-based IDSs (BBIDSs) have obvious advantages, they can suffer from three interrelated problems: poor signature expressiveness, behavioral obfuscation and run-time signature matching inefficiency. Signature expressiveness determines the success of an IDS in detecting new realizations of the same malware. Since most malware incidents are derivatives of some original malware, a successful signature must capture invariant generic features of the entire malware family. At the same time, the signature should be expressive enough to reflect most of the possible malware realizations. Behavioral obfuscation is an emerging threat that, given the extensive development of BBIDSs, is expected to become a necessary and trivial feature of future information attacks [26].

The behavior of a program can be viewed as a manifestation of the functionalities implemented in the program. A particular functionality is malicious if it performs some specific activities intended for adversarial purposes. Discovering a malicious functionality in any software qualifies it as a malware. Hence, the detection of malicious functionalities becomes crucial and sufficient for confident malware detection.

We developed a novel system call domain IDS that addresses existing and future challenges to BBIDSs. In order to achieve higher signature expressiveness, we proposed to specify the functionalities of interest, specifically malicious ones, by creating activity diagrams (ADs) in terms of both standard system objects and abstract behavioral constructs named functional objects. The utilization of functional objects and operations provides the necessary level of generalization, yet it preserves discriminatory properties of the specification. As a result, such an AD would incorporate multiple realizations of the specified functionality, hence increasing the semantics and expressiveness of the signature.

We investigated possible approaches to behavioral obfuscation including inter-process (multipartite) techniques. To mitigate obfuscation, we proposed an automatic generalization of the AD specifications. We developed a set of generalization algorithms that automatically augment signatures, making them resilient to several behavioral obfuscation techniques, such as object relocation and multipartite activity.

Finally, we developed a procedure capable of automatic conversion of activity diagrams into the system call domain using Colored Petri nets (CPNs). These are intended for run-time recognition of the specified functionalities in the IDS. Our experiments showed that a CPN is highly dependable and efficient for recognizing specified functionalities in the flow of system calls with data.

The system architecture of the proposed IDS is shown in Figure 5. In the learning phase, an expert designs ADs representing known malicious functionalities. The Specification Generalizer module automatically augments the original ADs making them more generic and resilient to obfuscations. The CPN Constructor generates a low-level and a high-level CPN by processing the relevant ADs. The low-level CPN recognizes individual subsystem-level object operations in the system call domain and aggregates the system call information for processing at the higher level. The high-level CPN recognizes the specified functionalities in the object operations domain. While simulating the CPN, The Recognizer accesses the information flow tracer to feed data dependencies for particular transitions in the CPN.

As shown in Figure 5, at the detection phase the Object Operation Recognizer receives system calls and utilizes the low-level CPN to identify subsystem object manipulations. The Functionality Recognizer then utilizes the high-level CPN to assemble object operations into particular functionalities.

The contributions of this research are as follows.

1. Increasing signature expressiveness and simplifying the process of signature specification:

- Formal functionality specifications using ADs are defined at the abstract object level. Each specification allows for capturing multiple alternative realizations of a given functionality
- Separation of the specification domain (abstract OS objects) from the detection domain (system calls). The abstract specification domain allows an expert to concentrate on conceptual realizations of a functionality omitting certain implementation details. The detection domain allows for efficient functionality detection in the system call flow by executing the respective CPN that was obtained from the specification.
- Automation of the IDS signature generation process. It includes computer aided AD specification design, automatic AD generalization, AD visualization and finally automatic translation of the AD to a CPN that is used as a signature in the intrusion detector (Functionality Recognizer)

2. Mitigation of possible behavioral obfuscations:

- Analysis and classification of possible behavioral obfuscation techniques
- Automatic generalization of functionality specifications thus making them invulnerable to behavioral obfuscation

3. Achieving high efficiency of signature matching:

- Automatic translation of an AD specification into a CPN that recognizes the functionality in the system call domain
- Efficient CPN simulator for recognizing specified functionalities in the flow of system calls and flow of utilized data/information (available at http://apimon.codeplex.com as an open source project).
- Prototype of information flow tracing engine (implemented in IDA Debug)

To demonstrate our approach we implemented it in a prototype IDS and tested it by detecting several malicious functionalities that are employed by network worms and bots, including self-replication engines and various malicious payloads.

4.1 State of the Art in Signature Based Behavior Detection

Current commercial intrusion detection systems such as antivirus packages primarily use a signature based approach for efficiency reasons. However, recent information attacks demonstrate steady increase in the quality and sophistication of newly deployed malware. To avoid detection by most commercial antiviruses, modern malicious software is at least polymorphic and sometimes even metamorphic. This results in an exponential increase in the number of signatures that need to be maintained by modern anti-viruses software (close to 5 million as of today, Kaspersky). Apparently, this reality drives traditional antiviruses to a dead end. It is interesting, however, to note that the number of distinctive malicious functionalities (behaviors) has not changed. This is due to the fact that malware continues to use the same functionalities to achieve the same goals. Hence, it is reasonable to detect functionalities rather than binary patterns. Moreover, dynamic IDS utilizing behavioral signatures to match malware activity cannot detect binary morphism.

The success of a dynamic, behavioral based IDS is determined by two aspects: the expressiveness of the signature specification language and the efficiency of the recognition mechanism. Moreover, usability of an IDS depends on the clarity and degree of abstractness of the specification language. Below we survey the existing behavioral specification languages and discuss the advantages of our approach. Then we show how our system is different and better than similar, existing system call domain IDSs.

4.1.1 State-Transition and CPN Specifications

Kumar and Spafford developed the IDIOT system that utilizes a variant of a CPN, termed Colored Petri Automation (CPA), to detect UNIX system misuses [27]. Compared to a CPN, CPA lacks concurrency, local condition variables and arc generative expressions. While concurrency may be not critical for detection, condition variables and arc generative expressions are vital for structural simplicity of the CPN recognizing our AD specifications. Moreover, our ADtoCPN procedure represents AD variables in CPN arc expressions.

Helmer et al. utilized a hierarchical CPN and Software Fault Trees (SFT) as behavioral specification for a distributed IDS [28]. First, the CPN structure is obtained from SFT specifications. Next, an expert defines CPN semantics (arc and guard expressions) that are

responsible for processing monitored system/network events. The spirit of their approach is close to ours in that it implies a separation of the specification and execution domains. However, the authors in [28] did not address generalization and behavioral obfuscation issues, which are our main contributions.

Ho et al. proposed a Partial Order Sate Transition Analysis Technique (POSTAT) to specify both local and distributed attack scenarios that are matched by CPNs [29]. The authors also give insight for creating a normalcy specification through POSTAT.

Eckmann et al. proposed the STATL language for specifying misuse signatures in the domain of interest (host or network) [30]. In STATL transitions represent executed activity and states represent the status of particular system objects. While STATL allows for specifying quite generic activity, the specifications must also reflect execution semantics making the signature description cluttered and complex. In contrast, we do not specify execution semantics in our AD specification, since the AD is ultimately translated into the CPN that possesses the execution semantics.

The problem of most state transition (ST) techniques is that the signature serves as a specification and a recognition mechanism at the same time. Hence, ST signatures are specified in the corresponding execution domain, e.g. system calls, network low level activity, shell commands, etc. For instance, for host based detection, an expert has to specify how transitions should process system calls invoked by malicious activity. Consequently, the specification would be overloaded with implementation details making it hard to create and verify. In contrast, we specification domain from the execution/detection domain. We specify functionalities at a high level (object operations) and we detect specified functionality at a low level (system calls). As a result, our specifications are quite readable and succinct, reflecting only critical malicious activity at a high/abstract level, while omitting low level implementation details. In particular, in our system an expert does not have to bother about how a CPN processes monitored system calls to recognize the specified behavior, because a CPN's structure and semantics are automatically built from a rather generic AD specification that could be defined in terms of abstract functional objects.

In summary, our approach differs from the above methods in the following aspects:

First, the specifications in the above papers do not formalize behavior in the domain of our interest, i.e. system object manipulations; therefore such specifications are not directly suitable for automatic processing. In contrast, our AD formalism is defined in the object operation domain, thus allowing for automatic processing and generalization.

Second, the authors utilize a CPN as a behavioral specification language, however we utilize a CPN solely as a recognition mechanism. In [2] we discovered that functionality specification through a CPN in the system call domain is tedious and hardly feasible for complex cases, hence, we developed a simple and generic AD formalism for a functionality specification. Afterwards, our ADtoCPN procedure translates the generic AD to the system call domain with a CPN serving as a recognition model.

Finally, our AD based specification is generic due to the use of functional blocks that abstract certain system implementation details yet provides enough agility for fine tuning of the specifications.

4.1.2 Declarative and Algebraic Specification Languages

Examples of declarative languages are LAMBDA [31], ADeLe [32] and Sutekh [33]. Specifications based on such languages define patterns of actions involved in an attack, such as the system pre and post conditions defined as a set of high level predicates and then mapping

between the actions and observable system events manifested by the attack actions. While such languages can express fairly generic attack scenarios, the complex scenario descriptions would require specifying many abstract predicates that would have to be identified by the detection mechanism. Runtime detection of such predicates may impose high overhead, and post alarm verification of the predicates could be unfeasible due to changes in the system state [31]. In contrast, we do not specify pre-conditions, because validity of the system patterns is verified by CPN at runtime by examining system call outputs.

The aforementioned declarative languages allow for relating attacks to one another by matching the post-condition of an attack with the pre-condition of another one. Our approach also allows for including one AD as a functional object to the specified AD. At the CPN level this would be realized as hierarchal CPNs.

The language of the CARDS [34] system introduces an abstract system view that allows distributed hierarchical attack specifications to be defined in the domain of abstract events. The detection language SHEDEL [35] takes advantage of both algebraic and state-transition approaches. As with [34], the SHEDEL specifications are explicitly defined at the abstract event level.

Declarative languages are advantageous over state-transition specifications in their ability to describe signatures at the pure abstract level, omitting operational details related to the attack execution domain. To overcome this inherent limitation of ST languages, we proposed so-called functional objects that serve as a system abstract level on which AD signatures could be specified.

Unlike state-transition specifications, most declarative and algebraic specifications do not posses execution semantics. For the system call domain, IDSs based on declarative specifications require a recognition mechanism that can explicitly match system calls with signatures. As the result an IDS's scalability ultimately depends on the efficiency of the recognition/matching algorithm. Usually runtime pattern matching is performed through some sort of state machine or rule based detector as in [33].

However, ST models are more efficient for handling multiple instances of the same pattern. If a pattern is observed more than once and each instance is yet incomplete, a state machine has to accommodate extra states for each instance of the pattern to trace them in parallel. In contrast, a ST model represents an executed event pattern as one token residing in the corresponding place that allows for processing of multiple pattern instances with low overhead. Moreover, in the CPN, the necessary attributes propagate as token fields that allow for relating system calls by process and thread ID. This makes it possible to recognize an inter-process activity. Consequently, ST signatures are executable in the event domain. With ST the detection becomes the execution, and it should be faster than signature-event matching. Based on the above arguments, we believe that state-transition specifications are more preferable to declarative languages with respect to scalability and efficiency.

Our AD specification can be directly converted to a recognition mechanism configuration, i.e. high level CPN. As a result, in our case, the recognition mechanism (CPN) incorporates the signature in its execution semantics ensuring low detection overhead. In other words, the detection becomes the direct execution of a CPN, hence no cross-matching for signature vs. event has to be performed.

4.1.3 System Call Domain Specifications

The system calls present a perfect domain for behavioral based misuse detection because system calls are executed in a safe Kernel mode and monitoring them is much more resilient against user mode malware.

Publications [36], [37], [38], and [39] propose tracing sequences of system calls to reveal misuse in OS object manipulations that are indicative of malicious activity. These methods discard the semantic relationships between object manipulations. A few attempts to deduce this semantic information on the level of primitive functional blocks [36], [37] failed to define a complete picture of the process behavior. PC Tools ThreatFire antivirus [38] allows the user to specify rules that describe only individual operations on objects such as process, file, etc., facilitating the detection of primitive and obvious misuse such as system file access, or starting a particular executive object. In contrast, our approach allows for the recognition of complex functionalities (such as self-mailing) that involve interrelated sessions of object operations.

Publications such as [40], [41], [3], and [42] target dynamic behavior analysis. The methodology presented in [42] describes the detection of virus activity through tracing system call events with an emphasis on the order of events without functional context. In contrast, our approach allows for specifying both patterns of interrelated manipulations and primitive functionalities. The Dynamic Code Analyzer (DCA) approach allows for constructing a so called "gene of self-replication" from primitive object operations and activity blocks, but it lacks an efficient recognition mechanism [3].

Authors in [41] utilize behavior graphs that is in fact an extension of the malware specification graphs presented in [40]. Our work is different/advantageous over [40] and [41]. First, in [40], specification matching is performed through static analysis and implies mapping to some sort of a call flow graph of the tested executables. Obviously, such behavior graphs do not address execution semantics, and could not be utilized as a run-time recognition mechanism. In order to recognize the specified behavior in [41], the authors mention the use of a behavior matching algorithm, unfortunately they do not formalize the algorithm itself. In particular, it is unclear how they might handle multiple system call chains. In contrast, as mentioned above, a CPN effectively handles multiple chains due to the use of token dynamics. Second, the specification in [41] can define alternative realizations, but their behavior graphs are constrained to a single process. In contrast, our AD allows for correlating operations invoked by different processes, thus we can specify malicious inter-process and inter-host activity. Finally, the authors in [41] did not analyze and address possible behavioral obfuscations. At the same time they did not propose a solid behavior formalism, hence it not feasible to automatically process a specification. In [41] an expert is responsible for generalization of the specifications, for instance, to avoid handle duplication that could significantly complicate the design process. In contrast, we formalized functionality in the object operation domain. This formalization allows for the automatic processing of AD specifications to address possible behavioral obfuscations. We understand that we addressed a rather limited set of obfuscations, but given the flexibility and fidelity of our functionality formalization, developing new generalization algorithms for anti-obfuscation should be feasible for an expert.

4.2 Functionality Definition and Specification

4.2.1 Formalization of the Specification

Before formalizing functionality specification, let us inspect functionality from the OS perspective as presented below. The MS Windows OS provides system recourses and services to processes through executive objects maintained in the Windows Kernel. In order to access a particular resource or service, a process creates a corresponding object such as a file, process, thread, memory section, etc [11]. Every object has its own set of operations which are exported to the user mode processes through system services (system calls). In the user mode, such system calls are invoked directly or more conveniently through subsystem API functions.

Processes invoke API functions or system calls to perform object operations (manipulations)[7] that complete some semantically distinct actions, such as writing data to a file or sending data to a specified IP address. Consequently, we define individual functionality as a combination of actions that achieve a certain high-level objective. It is important to understand the difference between a functionality and behavior. The behavior of a process is what the process does at the particular stage, while the functionality determines semantic goals of the process. In other words, behavior simply manifests the realization of functionality. As a result, the major limitation of the existing behavior-based specifications is that they fail when dealing with multiple realizations of the same functionality. This motivated us to develop a novel specification that is free from this shortcoming.

Note that processes may utilize both user level objects exported by the windows environment subsystem, such as Socket, Memory Mapping etc., and kernel level objects exported by the object manager, such as File, Named Pipe, Memory Section, etc. For user level objects, we consider subsystem level operations exported by API functions of subsystem libraries (such as kernel32.dll, ws2_32.dll). However, each subsystem level object is based on a kernel level executive object.

Table 2 features a simple functionality, “Remote Shell”. This functionality creates a backdoor allowing an attacker to remotely execute system commands. Remote Shell has at least two possible realizations: Bind Shell and Reverse Shell. Both realizations create the “cmd.exe” process with input and output buffers (*hStdInput*, *sStdOutput*) being set to a connected port. The difference between these realizations is that the Bind Shell accepts a connection via named pipes and Reverse Shell connects to the attacker via sockets. Both realizations invoke the *CreateProcess* API with specific flags that allow for using socket/pipe handle as an input/output. Ultimately, this makes the command interpreter listen to incoming commands and execute them.

---

[7] Here, we use terms “Operation” and “Manipulation” interchangeably, because both of the terms are used extensively in the literature.

Table 2. "Remote Shell" Realization

Bind Shell realization	*Reverse shell realization*
1. ***h_In*=CreateNamedPipe**(*dwOpenMode*= PIPE_ACCESS_INBOUND *SecurityAttributes.bInheritHandle*=TRUE); 2. ***h_Out*=CreateNamedPipe**(*dwOpenMode*= PIPE_ACCESS_OUTBOUND *SecurityAttributes.bInheritHandle*=TRUE); 3. **ConnectNamedPipe(*h_In*)**; 4. **ConnectNamedPipe(*h_Out*)**; 5. **CreateProcess**("cmd.exe", *bInheritHandles* = TRUE, *STARTUPINFO.dwFlags* = STARTF_USESTDHAND, *STARTUPINFO.hStdInput*= ***h_In***, *STARTUPINFO.hStdOutput*= ***h_Out***);	1. ***s*=socket**(); 2. **connect**(s, *sockaddr.s_addr*=Attacker_IP *sockaddr.sin_port*=Attecker_port); 3. **CreateProcess**("cmd.exe", *bInheritHandles* = TRUE, *STARTUPINFO.dwFlags* = STARTF_USESTDHAND, *STARTUPINFO.hStdInput*= ***s***, *STARTUPINFO.hStdOutput*= ***s***); *STARTUPINFO.hStdOutput*= hSock);

Consider requirements for specifying functionality. Based on the above example, [40], and [41] we formulate the following requirements for the functional specifications:

1. ***The specification must define the control flow for object operations***. It must support conditional branching and concurrent execution. *Conditional branching* allows for specifying alternative realizations that may utilize different objects and operations; however, they achieve the same goal determining the functionality. For instance, two realizations in Table 2 utilize two different objects: "Named Pipe" and "Socket". *Concurrent execution* allows for specifying independent object manipulation sessions, that could be executed in any order. However, the sequence of dependent operations must remain intact within the session. For instance in the Bind Shell realization (Table 2, left side), there are two independent operation sessions: create inbound pipe and get it connected (APIs 1 and 3) and create outbound pipe and get it connected (APIs 2 and 4). Since these two sessions are independent, the API functions (1, 2 and 3, 4) could be invoked in any order, as long as API 3 follows API 1 and API 4 follows API 2.
2. ***The specification must define data/information flow among object operations.*** This requirement allows for specifying how output attributes of operations become the inputs of consequent manipulations. An attribute data flow determines the discriminatory power of the specification. For instance, with "Remote Shell" functionality, we have to show that "Process" object is created with *STARTUPINFO.hStdInput* and *STARTUPINFO.hStdOutPut,* i.e. attributes set to the socket handle. We should point out that data may be passed by value (data flow) or by information (information flow) [43]. Information flow between source and destination attributes indicates that the value of the destination is a transformation of the value of the source. In addition to the above two requirements, we introduce a third one that overcomes certain limitations in [41] that are related to multi-processes activity.
3. ***The specification shall not be constrained to the context of one process.*** This allows for specifying and relating operations of different processes. In fact, this allows for specifying inter-process functionalities.

The specification must offer enough expressive power and convenient graphical notation to simplify its design. Many specifications provide graphical notations such as state

diagrams/machines, simple flowcharts, and workflow diagrams. However, a state machine is a uniprocess model that does not meet Requirement 3 and cannot directly express data flows as per Requirement 2. Flowcharts do not support concurrent execution and do not meet Requirement 1. However, a workflow diagram such as a UML Activity Diagram (AD) can generally meet the above requirements. Consider the formalization of an AD in terms of an OS object operation for the purpose of functionality representation.

A basic UML AD is a semantically weighted directed graph:

$$G = (Nodes, Arcs, Guards) \qquad (1)$$

where $\texttt{Nodes} = \texttt{State} \cup \texttt{Pseudo}$.

The set `State` contains state nodes that represent executed activities; it also contains *Initial* and *Final* nodes that represent the beginning and end of the process. The set `Pseudo` comprises pseudo state nodes that control the execution flow. Pseudo state nodes include decision/merge (for conditional flow branching), and fork/join (for concurrent flow execution). The set `Guards` contains guard expressions (for conditional branching) that represent the semantic weight of the corresponding edges.

Consider the "Remote Shell" functionality in Table 2. Figure 6 depicts an AD of the functionality in the graphical (left side) and analytical (right side) forms. According to UML 2.x standards, the graphical notation displays decision/merge nodes (a, d) as diamonds, and fork/join nodes (b, c) as bars. We also assigned a sequential index to each node for explanatory purposes.

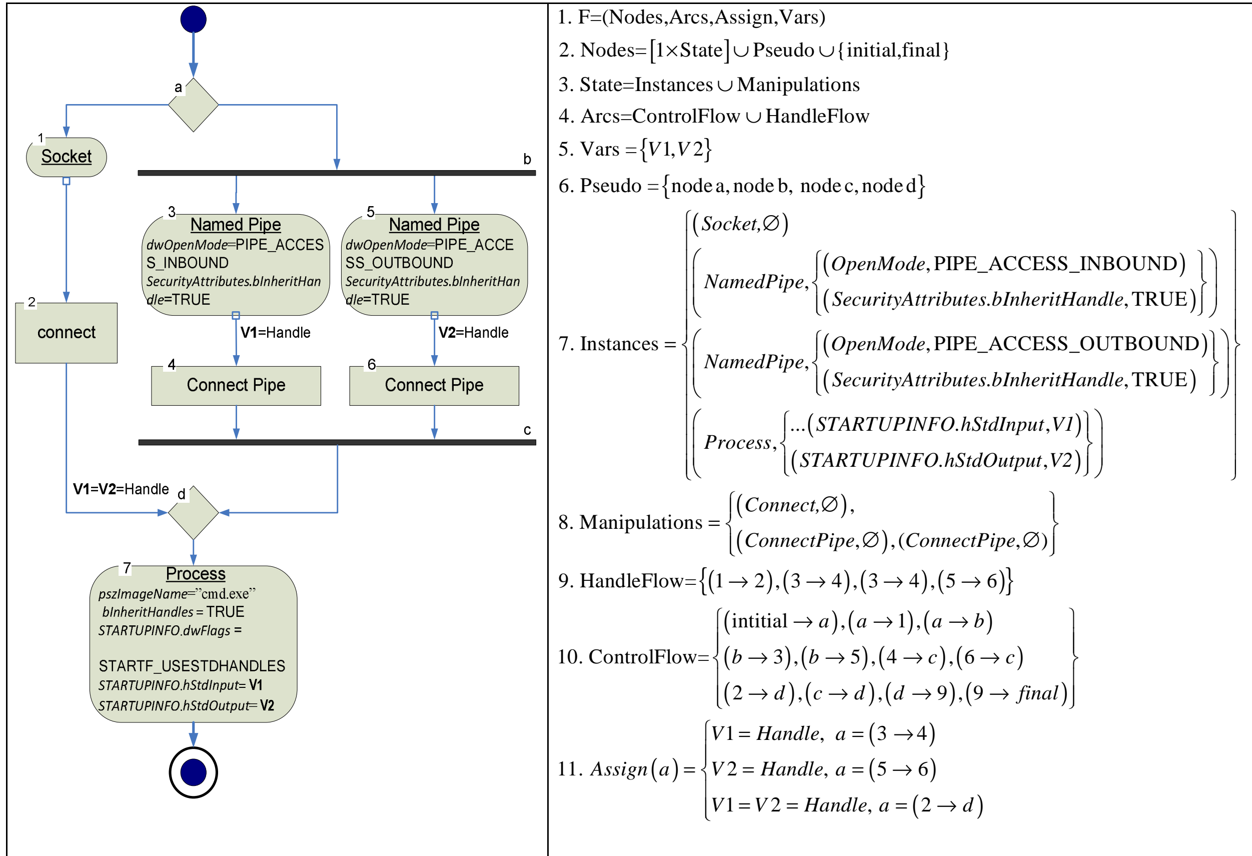

Figure 6. Activity Diagram of the "Remote Shell" Functionality

The AD contains seven state nodes and four pseudo-state nodes. The state node set includes four created object instances (Nodes 1, 3, 5, 7) and three operations (Nodes 2, 4, 6) on these objects. The pseudo-state nodes determine control flow of the functionality. The decision node “a” starts two alternative realizations of the functionality. The left branch (Nodes 1-2) represents the first step of the Reverse Shell realization, while the right branch (Nodes 3-6) represents the first step of the Bind Shell realization. Node 7 is the common step for both realizations. Note that the first step of the Bind Shell realization (Nodes 3-6) has two independent sessions (Nodes 3-4 and 5-6).

This graphical notation is convenient for an expert designing a specification. However, the analytical representation is crucial for automatic processing of the specification. The analytical form of “Remote Shell” functionality shown in Figure 6 (right side) is very consistent with the UML AD formalism (1). The only modification we made was the inclusion of the `Vars` component that represents a set of local variables. Next we provide a generic explanation of all of the components of our formalism (the detailed formalism of the AD specification is given in Appendix A).

The functionality specification is defined as an AD tuple:

$$F=\left(\text{Nodes,Arcs,Assign,Vars}\right) \tag{2}$$

where

`Nodes` is a multi-set defined in line 2 (Figure 2). It consists of `State` and `Pseudo` nodes. As defined in line 3, there are two types of State nodes: `Instances` and `Manipulations`. Each *Instance* node represents an object created and operated on in the context of the functionality, and its attributes. Line 7 shows the set of Instance nodes for the “Remote Shell” functionality. The set includes four nodes corresponding to the following objects: socket (Node 1), two named pipes (Nodes 3 and 5) and the process (Node 7).[8] Line 7 indicates that each object is defined with the attributes used in the corresponding API that is creating the object. Each *Manipulation* node represents an object manipulation with its appropriate parameters. Line 8 shows the Manipulations set for “Remote Shell”. This set includes three manipulations: a connect socket (Node 2), and two connect named pipes (Nodes 4 and 6).

`Arcs` is a set of directed arcs connecting the AD nodes. As defined in line 4, the arcs could be either of two types: Handle flow and Control flow. Handle flow arcs correspond to the execution flow with handle inheritance. A handle arc indicates that the destination operation (node) utilizes the same object handle as a source operation and is executed right after the source. In other words, the handle arc passes the handle of the source node to the destination node. For instance in Figure 6, the arc between Nodes 1 and 2 is a handle arc. It passes the handle of the socket created in Node 1 to the connect operation in Node 2 (this indicates that the socket from Node 1 is operated on at Node 2). To distinguish the handle arcs from control arcs, we display the handle arcs beginning with rectangle.

Control Flow arcs define the control flow without handle inheritance. An arc from this set indicates the execution order and does not imply any data binding (via handle or attribute). For instance, the arc between Node “d” and Node 7 is a control arc that does not transfer any handles, simply showing that Node 7 should be executed right after the

[8] In the brackets, we show the node index as presented in the graphical form.

two sessions (Nodes 1-2 and Nodes 3-6). In order to transfer data between nodes, variables are used. For instance, pipes or socket handles are transferred to Node 7 through variables V1, V2.

`Assign` is a function that binds variable assignment expressions to corresponding arcs. Line 11 shows the definition of such a function. The function indicates that variable V1 is assigned with a handle that was utilized in Node 2 or Node 3.

`Vars` is a set of local variables that are used to define data flow. Utilization of local variables with an assignment expression allows for specifying a data flow between object operations (the second specification requirement). To specify an information flow, an expert should use transformation notation *T*() as depicted in Appendix A. Note that it is possible to define informational dependency explicitly as well as implicitly. Explicit information flow implies specifying a formal mapping between source and destination. Implicit flow does not specify any mapping and simply states that the destination value should at least partially depend on the source value.

To address the third specification requirement each *State* node is assigned a unique index of the process that performs the manipulation represented by the node. Note that in Figure 6, "Remote Shell" is defined as an intra-process functionality, hence, all the operations are invoked by the same process, and every *State* node of the AD has the same process index which is 1 (see line 2).

From Figure 6, one can see that the graphical representation of an AD is much more revealing than the analytical representation; however, both representations are formally identical. In the rest of the paper, the graphical representation will be used for explanatory purposes, while the analytical representation will be used in specification processing algorithms.

4.2.2 Specification Abstraction

The detection success of our system highly depends on how comprehensive the specification is. If an expert misses a functionality realization, the system will be prone to false negatives and miss an attack vector. Each specification must be as generic as possible. It should abstract certain implementation details enabling experts to concentrate only on conceptual realizations. This is accomplished by the introduction of so-called functional objects that represent some complex but rather standard OS functionalities/mechanisms such as Inter-Process Communication (IPC), File Download, etc. Note: a functional object abstracts several alternative realizations of the particular OS functionality by encapsulating the necessary Windows objects utilized in these realizations. Each functional object has a set of operations representing certain high-level activities. When specifying an AD, experts may create and manipulate functional objects like ordinary Windows objects. Table 3 shows the set of functional objects facilitating data transfer. This set is just an example and is far from being complete, however, it demonstrates the expressiveness of our approach, namely the ability of operating on abstract objects by other objects. For brevity, we discuss only a few functional objects.

Table 3. Functional Objects for Data Transfer

Functional object name	Based on objects	Operations	Attributes (input => output)
GenericFile	File, File Mapping	Create	FileName
		Read	BufferLength => Buffer
		Write	FileName, Buffer
RemoteIPC	Socket, Pipe	Create	EndPoint (server, client), ID => Type, Handle

	MailSlot	Wait	Type, Handle
		Recv	Type, Handle => Buffer
		Send	Buffer, Type, Handle => Buffer
LocalIPC	GereicFile, FileMapping RemoteIPC	Create	EndPoint (server, client), ID => Type
		Recv	=>Buffer
		Send	Buffer
FileTransfer	GenericFile RemoteIPC	Create	FileName, RemoteHost => Type, FileHandle, IPCHandle
		Send	(Type, FileHandle, IPCHandle)
		Recv	(Type, FileHandle, IPCHandle)

Object "GenericFile" abstracts file access operations; it encapsulates both the "file" object and "file mapping" object. Object "RemoteIPC" represents an IPC resource for inter-host data transfer. It abstracts three alternative IPC mechanisms: socket, named pipe and mailslot. The "RemoteIPC" object exports the following operations: `Create`, `Wait`, `Recv` and `Send`. The ADs for these operations are shown in Appendix B in figures B-1, B-2, B-3 and B-4. Some operations have input and output attributes. For instance, the operation "RemoteIPC Create" requires two input attributes: Endpoint class (either server or client), and EndPoint ID (host IP and Port for the socket and a name for the pipe/mailslot). The operation returns two outputs: EndPoint type (socket, pipe or mailsot) and a handle value of the corresponding object. The operation `Wait(h)` waits for an incoming connection to the newly created IPC endpoint with handle `h`.

Note that from the expert's perspective, the utilization of such functional objects is transparent. For instance, when using RemoteRPC in a specification, the expert should not make any assumptions on how a malware will perform IPC, through a socket, pipe or mailslot. Such a transparency is best exemplified by FileTransfer operations. Table 2 indicates that FileTransfer operations are based on sheer functional objects such as GenericFile and RemoteIPC. This demonstrates the generalization power of the proposed specification formalism. Armed with such functional objects, an expert can build quite generic specifications yet preserve discriminatory properties that would leave little room for detection evasion.

4.3 Behavioral Obfuscation

The discriminatory power of a behavior signature (functionality specification) defined by an expert could be quite subjective and may exclude some of the realizations of the functionality. In addition, an attacker may perform some sort of behavioral obfuscation to evade detection. To address this issue, we developed a set of algorithms that automatically generalize (augment) the specification of the functionality. In the rest of the section, we will discuss possible behavioral obfuscations then we introduce the generalization algorithm and show how it addresses various obfuscation techniques.

4.3.1 Behavior Obfuscation Techniques

By utilizing functional objects, experts may specify most of the realizations of the functionality. Then it would be difficult for an attacker to discover yet another conceptually different realization utilizing different Windows objects. However, to evade detection, an attacker does not have to implement a completely new realization. He may simply obfuscate a known realization in such a way that it would break the specification. We distinguish inter-

process and intra-process approaches to obfuscate a realization without affecting the functionality. Inter-process obfuscations utilize multiple interrelated processes that, at high level, jointly perform a particular malicious functionality. Intra-process obfuscation locally alters a realization of the functionality while preserving its behavioral semantics.

First, consider possible inter-process approaches to behavioral obfuscation.

1. Utilization **of legitimate third party utilities to perform a required activity.** A malicious process may run third party utilities to execute some important tasks that may be a part of the functionality. In this way, the process executes the functionality without performing some key object manipulations involved in the task. For instance, a file virus usually searches for executables using the "FindFirstFile" and "FindNextFile" API. Instead, the virus may utilize the system command interpreter (e.g. "cmd.exe") to retrieve a list of executable files in a folder and then access the files one by one.
2. **Distribution of the functionality among several processes a.k.a. multipartite approach.** A multipartite malware consist of several agents that perform coordinated activity to achieve a common goal. Such malware can distribute a malicious functionality among several processes by injecting its code into active benign processes or by creating new ones. Then the combined activity of these processes will perform an inter-process malicious functionality. A real life example of such a malware is a KeyLogger which is described in the next section. Another example is a File Virus that consists of two processes. The first process opens an executable file and passes the file handle to the second process. Then the second process attaches the code from the first process to the opened victim file. Neither process performs a typical malicious functionality individually: the second process does not open the victim file and does not inject its code, while the first process is replicated into the victim file without performing write or self-access operations.

Now let us consider intra-process obfuscation approaches.

1. **Object relocation and duplication.** Since a functionality may be constrained by a particular object name (e.g. file name), an attacker may change the name of the object before manipulating it. For instance, an attacker can copy, rename or move a file before manipulating it. In addition, a malware may duplicate an object handle in the middle of the manipulation sequence to break system call binding. Additionally, an attacker may access objects through symbolic links instead of handles.
2. **Non-direct object manipulation.** This is achieved by specific, low-level system tricks such as utilization of non-trivial OS resources that allow for accessing objects either in a non-trivial way or through a "middleman" object. For instance, an attacker can create reparse points or can access files by their streams. He may also add an alternative path to a target file through relinking system calls. Such activities are performed only through Kernel objects using system calls.

4.4 Specification Generalization, Anti-Obfuscation

In the system architecture presented in Figure 5, the "Specification Generalizer" module addresses the above obfuscation techniques. Effectively, this module attempts to fill the expert's experience/attention gap, thus alleviating limitations related to the human factor. The module applies a set of generalization algorithms that automatically augment a given AD to make it less prone to obfuscations. Herein, we propose the following generalization algorithms:

`TraceFiles` – Augments the given AD with functionalities tracing the renaming and relocation of all files involved in the specification. This algorithm addresses the third obfuscation technique.

`TraceHandles` – Augments the given AD with functionalities that trace object handle propagation among processes, which requires tracking handle duplication and the IPC used for handle transfer. This addresses the first three obfuscation techniques.

`TraceProcesses` – Augments the given AD with functionalities that track process generation, remote code injection and inter-process coordination. This involves detecting several realizations of code injection including remote thread based and remote hook based injection. The upgraded AD would be able to relate object manipulations performed by multiple processes. This algorithm mitigates the first and the second obfuscation techniques.

To address the forth obfuscation approach, one does not need any post-processing of the AD in the generalization stage. Instead, we can simply extend functional objects with the necessary semantics that would trace low-level objects involved in the obfuscation. This results in the obfuscation being resolved at the stage of specification, rather than automatic post-generalization. In particular, we add reparse points and file streams to the "GenericFile" functional object.

While augmenting an AD, each of the generalization algorithms incorporates special functionalities, termed generalization functionalities that trace certain activity involved in a particular obfuscation. Table 4 describes the generalization functionalities, whose ADs are given in Appendix C. Table 4. lists some functionalities that maintain certain global variables that qualify the traced activities, e.g. generated processes, duplicated files or established IPC channels.

We describe each of the generalization algorithms based on their pseudo-code, where we utilize several **primitive functions** defined in Appendix D and generalization functionalities as defined in table 4.

Table 4. Generalization Functionalities

Functionality	Updated variable	Description
FileRelocation	FList	Accepts a file name as input and updates a FList variable which a dictionary indexed by file names. Each element of FList is a list containing names of duplicates of the input file indexing the element. Such duplicates could be derived by copying, moving or renaming of the original input file or any of its duplicates.
ProcessGeneration CodeInjection	PList	These two functionalities trace process generation and inter-process code injection and constantly update the global variable PList. PList is a list containing the PID of the descendant (created or being injected) processes (up to a given generation limit) that originated from the initial process that starts an AD.
HandleDuplication	DupP DupH	Traces handle duplication and constantly updates two global variables: DupP and DupH. **DupP** is a dictionary indexed with the value of the initial handle produced at object creation. Each element of the dictionary is a list containing PIDs of the processes possessing duplicate handles derived from the handle indexing the element. **DupH** is a two dimensional dictionary indexed by value of the original handle

		and PID. Each element of the dictionary is a list of handles possessed by indexing PID and derived from the indexing handle. For instance, DupH[H1][P1]={H2,H3} means that handles H2, and H3 were derived from the original handle H1 and are possessed by the P1 process.
LocalIPC Establishment	IPC_P IPC_H	Traces Local IPC establishment and constantly updates two global variables IPC_P and IPC_H. **IPC_P** is a dictionary indexed by ID of the IPC. Each element of IPC_P is a list of PIDs of the processes that own endpoint handles (including duplicates) of the IPC with ID indexing the element. For instance, IPC_P[id1]={PID1,PID2,PID3} means that IPC, identified by id1, has endpoints which handles are possessed by processes PID1, PID2 and PID3. Note that some IPC could serve as data share points, hence they may have multiple endpoints, for instance a file or a shared memory. **IPC_H** is a two dimensional dictionary indexed by IPC ID and PID. Each element of the dictionary is a list of endpoint handles (including duplicates) that are possessed by indexing PID and shared by IPC with ID indexing the element. For example, IPC_H[id1][PID1]={h1,h2,h3} means handles h1,h2,h3 represent endpoints of the IPC with id1 ID and are possessed by a process with PID1 ID.

`TraceFiles` pseudo-code is given below (see listing 1). The algorithm iterates over operations and instances presented in an AD (line 1). If an operation has "file name" as an argument (line 2), the procedure adds "FileRelocation" functionality to the AD (line 6). Note: the function *AttList(x)* returns a list of attribute names for the input operation *x*.

While adding a parallel functionality is trivial, it is not obvious where to insert "FileRelocation" so that file tracing does not interfere with the rest of the original functionality. Thus, we insert a parallel flow with "FileRelocation" in the following way: if the target file name is a constant string, i.e. it is independent from other operations of the AD, we start the parallel flow right after the *initial* node; if the file name is a variable, we start the parallel flow right after the node where the variable is assigned for the last time; finally, we join the "FileRelocation" parallel flow with the original AD right before the node that is performing the operation on the target file.

Listing 1. Trace Files Algorithm

Algorithm **`TraceFiles`**

Input: **AD** - An activity diagram specification

Output: Generilized AD

---

1. **foreach** $Operation \in \{\text{AD.Instances} \cup \text{AD.Manipulations}\}$ **:**
2. **if** $\left(lpFileName \in \text{AttList}(Operation)\right)$ **:**
3. $TargetFileName := \text{GetAttributeValue}(Operation, lpFileName)$;
4. **if** $\left(isVariable(TargetFileName)\right)$ **:** $RelocStartNode = GetAssignNode(TargetFileName)$
5. **else :** $RelocStartNode = \text{AD.initial}$;
6. $\text{AddParallelFunct}\left(\text{AD}, \text{FileRelocation}(TargetFileName), RelocStartNode, Operation\right)$;
7. $\text{SetAttributeValueExpression}\left(Operation, lpFileName, "lpFileName\ in\ FList[" + TargetFileName + "]"\right)$;

In the algorithm, the parallel flow with "FileRelocation" functionality is added by the function ***AddParallelFunct****(Origin,New,Start,Merge)* (see Appendix D). It adds an AD named *New* to an AD named *Origin* as a parallel flow that starts right after the node *Start* and joins it to the AD *Origin* just before the node *Merge*. The node *Start* is determined in lines 4 and 5. If the file name is a variable, the node *Start* is defined through ***GetAssignNode*** function (line 4). Function ***GetAssignNode****(x)* returns the node whose output arc has an assignment expression for variable *x*. Line 7 modifies the AD to make it consistent with the AD formalism (2) given in Section 3.

`TraceHandles` pseudo-code is given below (see listing 2). In the code, line 1 introduces the "HandleDuplication" functionality as a parallel flow to the original functionality. Lines 2-8 constitute a loop that iterates over all object instances of the AD so that for each instance, a new element in *DupH* dictionary is initialized with the instance *PID* and *Handle* (lines 3, 4). This would allow "HandleDuplication" functionality to trace handle duplicates of the current object instance. Line 6 iterates over object operations performed on the current object instance. For each object operation, the algorithm redefines the *PID* and *Handle* expressions so the operation may utilize any duplicated handle belonging to the original object instance.

Listing 2. Trace Handles Algorithm

Algorithm **`TraceHandles`**

Input: **AD** - An activity diagram specification

Output: Generilized **AD**

```
----------------------------------------------------
1. AddParallelFunct(AD, HandleDuplication, AD.initial, AD.final);
2. foreach Object ∈ AD.Instances :
3.     SetAssignExpression(OutputArc(Object), "DupH[Handle][PID] = {Handle}");
4.     SetAssignExpression(OutputArc(Object), "DupP[Handle] = {PID}");
5.     HandleVarName = CreateNewVar(OutputArc(Object), "Handle");
6.     foreach Operation ∈ GetObjectOperations(AD, Object) :
7.         SetNodePIDExpression(Operation, "PID in DupP["+ HandleVarName + "]");
8.         SetAttributeValueExpression(Operation, Handle, "Handle in DupH["+ HandleVarName + "][PID]");
```

`TraceProcesses` addresses the first and the second obfuscation methods. In the first obfuscation, a malware runs an external utility to perform some tasks. As a result, the external utility has to utilize the OS resources the same way as the malware. In other words, malware simply outsources its operations or functionalities to the utility. We can recognize the outsourced functionality in the utility's behavior using our specifications. Consequently, in the specification some object manipulation sessions must have a PID tag assigned to the PID of the utility. If the utility has started, we must record the PID of the utility process and assign the PID in the object operation sessions that it outsources.

From the above perspective, starting a utility to perform a part of the malicious functionality represents a multipartite approach. Hence, the first and the second obfuscation techniques should be addressed similarly: by tracing the functionality distribution among several processes. This requires tracking processes generated by the malware as well as processes to which malware injected its code (infected). Then we attribute object operations to the generated processes and infected processes.

`TraceProcesses` algorithm introduces "ProcessGeneration", "CodeInjection" and "LocalIPCEstablishment" to the input AD. It also introduces the IPC required for coordinating

multipartite agents and/or communicating with the utility. To reduce the false positive rate we additionally trace data transmission between processes that represent technical yet vital activity. For instance, a process retrieves (reads) data through an object that represents data source, and then this data or its informational dependency is transferred (written) through another object, called a data sink (see Table 5). Distributing this activity in such a way that one process would access a source object and another process would access a sink object requires using the IPC responsible for data transmission from the source process to the sink process. Such distributed functionality in fact implements an inter-process information link between source and sink objects (recourses).

For the sake of clarity, in table 5 we present OS/functional objects and their corresponding operations that could be used for data source and sink. Note that some objects share the same source/sink operations.

Table 5. Source and Sink Operations

Objects	*Source operation*	*Sink Operation*	*Based on API*	
			Source operation	*Sink Operation*
File, Pipe, MailSlot	Read	Write	ReadFile.kernel32,	WriteFile.kernel32
Socket	Recv	Send	recv.ws2_32	send.ws2_32
Registry	ReadValue	WriteValue	RegGetValue.Advapi32.dll	RegSetValueEx.Advapi32.dll
RemoteIPC, LocalIPC	Recv	Send		

Below we give pseudo-code for `TraceProcesses` algorithm (see listing 3). Initially the algorithm introduces three functionalities (lines 2-4), LocalIPCEstablishement, ProcessGeneration and CodeInjection (see Table 4). The functionalities are incorporated through AddParallelFunct function that adds a functionality to the AD as a parallel flow to the entire original activity. In line 5, the algorithm iterates over all objects of the input AD and changes their PID assign label to "PID in PList". This means that a PID must belong to the global list PList that contains PIDs of the children processes derived from the original malicious process or its children (see Table 4).

Lines 6-20 constitute the main loop that locates source-sink operations (Table 5) and introduces the data transmission functionality between source and sink processes. Line 6 iterates the overall operations of the AD. If an operation is a sink (line 7), then the algorithm obtains a writable buffer/"*pBuffer*" attribute value. If the *bBuffer* attribute value is a variable (checked in line 9), the algorithm obtains the node that assigns the variable (line 10). Such an assigning node is viewed as a data source and data is transferred to the sink operation through the *SinkBuffer* variable. Next, the algorithm introduces IPC between a source process and a sink process. To achieve this, the algorithm adds *send* and *recv* operations as a parallel flow (lines 10-14) between the node assigning the variable (source node) and the current operation writing the assigned variable (sink node). Lines 15-20 set PID and attribute expressions of the newly introduced *recv* node and the current sink operation node, so that the two operations belong to the same process.

Listing 3. Trace Processes Algorithm

```
Algorithm TraceProcesses
Input: AD - An activity diagram specification
Output: Generilized AD
_____________________________________________________
1. Sinks = {Write, Send, WriteValue}
2. AddParallelFunct(AD, LocalIPCEstablishment, AD.initial, AD.final);
3. AddParallelFunct(AD, ProcessGeneration(ThisPID), AD.initial, AD.final);
4. AddParallelFunct(AD, CodeInjection(ThisPID), AD.initial, AD.final);
5. foreach Object ∈ AD.Objects : SetNodePIDExpression(Object, "PID in PList");
6. foreach Oper ∈ AD.Operations :
7.     if Oper ∈ Sinks :
8.         SinkBuffer = GetAttributeValue(Oper, bBuffer);
9.         if isVariable(SinkBuffer) :
10.            BuffAssingNode = GetAssignNode(SinkBuffer);
11.            AssignPIDVarName = CreateNewVar(OutputArc(BuffAssingNode), "PID");
12.            NewSendNode = AddParallelNode(AD, Send({PID, AssignPIDVarName}, {pBuffer, SinkBuffer}), BuffAssingNode, Oper);
13.            IPCIDVarName = CreateNewVar(OutputArc(NewSendNode), "ID");
14.            NewRecvNode = AddNextNode(AD, Recv(), NewSendNode);
15.            SetNodePIDExpression(NewRecvNode, "PID in IPC_P[" + IPCIDVarName + "]");
16.            SetAttributeValueExpression(NewRecvNode, Handle, "Handle in IPC_H[" + IPCIDVarName + "][PID]");
17.            RecvBufferVarName = CreateNewVar(OutputArc(NewRecvNode), "pBuffer");
18.            RecvPIDVarName = CreateNewVar(OutputArc(NewRecvNode), "PID");
19.            SetNodePIDExpression(Oper, "PID := " + RecvPIDVarName);
20.            SetAttributeValueExpression(Oper, pBuffer, "pBuffer = " + RecvBufferVarName);
```

To demonstrate our generalization algorithms, we tested them with a simple functionality that uploads a file through the IPC. We utilized Visual Paradigm for UML software [44] to design the “File Upload” functionality, see Figure 7. Then we ran the prototype of a Specification Generalizer module (Figure 5) that automatically generalized the functionality AD using all three algorithms. Figure 8 shows the AD of the augmented (generalized) functionality. Note that Visual Paradigm designer, with minor manual alignments, automatically produced both AD layouts of Figures 7 and 8. It can be seen that in our prototype, the entire process of AD generalization is completely automated including computer aided AD design, automatic generalization, and finally visualization of the resultant AD.

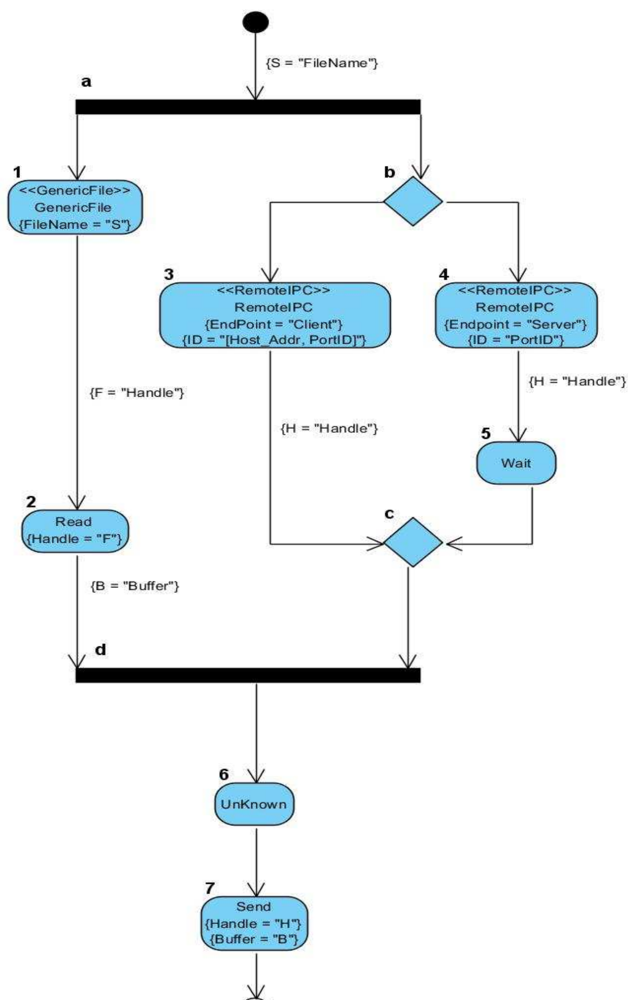

Figure 7. File Upload AD

As per the example in Figure 7, one can see the "File Upload" AD that has two independent sessions such as IPC establishment (Nodes 3-5) and reading a file (Nodes 1, 2). After establishing an IPC, a buffer is received through the IPC and written to the opened file. Node 6 represents some additional activity not related to data variables of the functionality. Node 7 sends the buffer with the file content via the established IPC. Note that in this AD, we utilized only functional objects and manipulations, hence covering most of the realizations of the "File Upload" functionality.

In the generalized AD (Figure 8), it could be seen that HandleDuplication, ProcessGeneration and IPCEstablishment functionalities are introduced in the original functionality as independent sessions. Since the name of the uploaded file is the input parameter, the Specification Generalization has introduced "FileRelocation" functionality right before the file open operation (Node 2). In Nodes 4 and 6, global handle dictionaries (DupH, DupP) are initiated with IPC object handle. HandleDuplication (Node 13) traces handle duplication and updates the dictionaries. Node 6 accepts a connection to the IPC server endpoint. One can see that in the process of generalization, the `TraceHandles` algorithm has modified attributes of several operations: in Nodes 3 and 6, PID the assignment was set to "PID *in* DupP[H1]"

expression which means that the PID of the operation must belong to the set of PID's that posses the original handle H1.

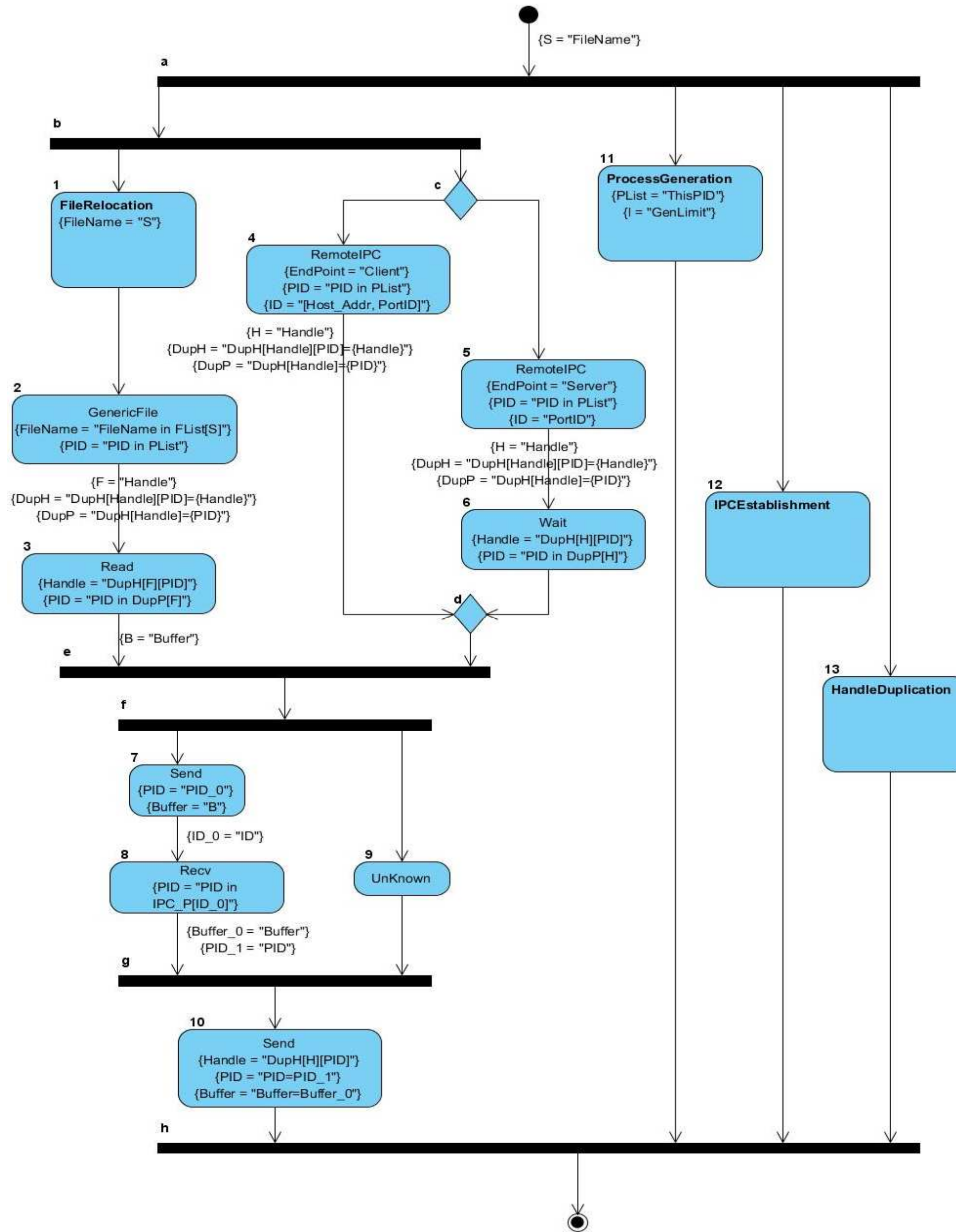

Figure 8. Generalized File Upload AD

The `TraceProcesses` algorithm introduced an IPC in Nodes 12, 7 and 8 that transfers a buffer from the source process (reading the content of the file to be uploaded) to the sink process (sending the buffer to a remote host). One can see that IPC nodes are introduced as an independent session from the other activity (Node 9).

Figure 8 indicates that the generalized AD is structurally more complex than the original AD (Figure 7). However, with respect to the number of state nodes it is comparable to the original. At the same time, the generalized AD addresses all three obfuscations presented above.

This shows that the proposed anti-obfuscation generalization results in an acceptable complexity penalty.

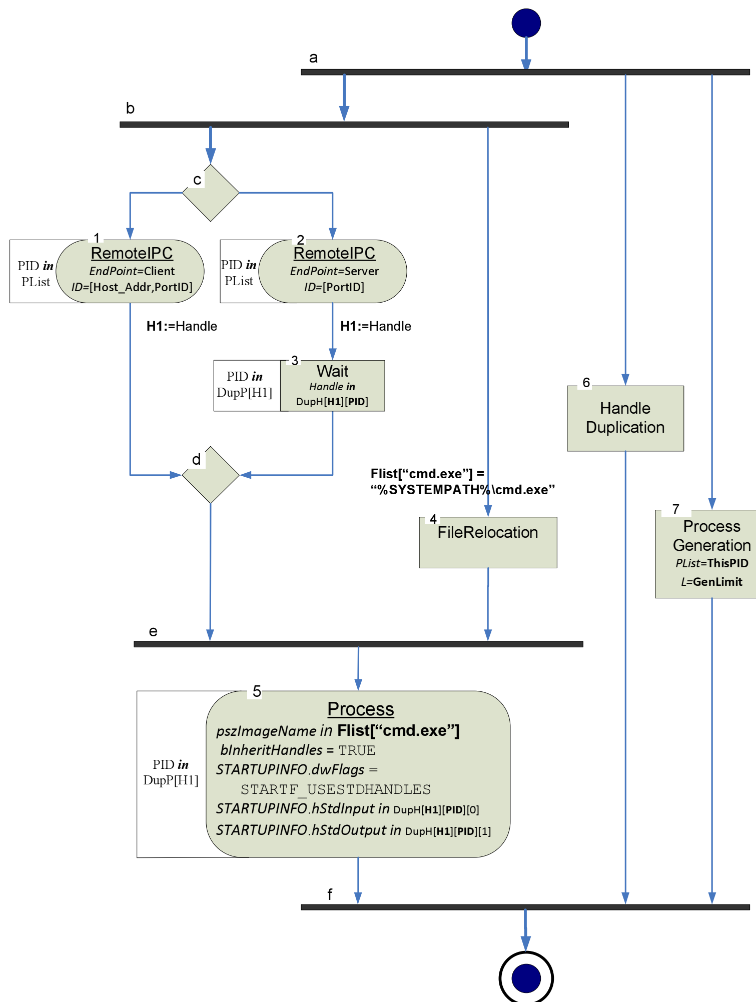

Figure 9. Generalized AD for Remote Shell

Furthermore, we applied the above algorithms to generalize the AD presented in Figure 6. The generalized AD is shown in Figure 9. One can see that fork node "b" starts two sessions. The left session (nodes 1-3) corresponds to the first steps of "Reverse Shell" and "Bind Shell" realizations. The right session is represented by one operation (Node 4), "FileRelocation" that traces "%systempath%\cmd.exe" file and outputs a list of files that descended from it.

The "Remote Shell" realization is started in Node 1. It creates a RemoteIPC object as a client that connects to the attacker host. The "BindShell" realization creates a RemoteIPC server in Node 2. The RemoteIPC object handle is traced by "HandleDuplication" functionality (Node 6). Node 3 corresponds to accepting a connection with the IPC endpoint. Nodes 1, 2 and 3 have a PID index. Note that the PID index is a part of the AD formalism presented in Section 3. The expression "PID in PList" means that the PID of the process performing the operation must belong to the PList. Nodes 1, 2 and 3 represent the inter-process part of the "Remote Shell" functionality. Such an inter-process part, along with Node 7, address obfuscation techniques 2 and 3. Indeed, nodes 1, 2 and 3 outsource the IPC creation to other processes.

The final step of the "Remote Shell" is to run "cmd.exe". Node 5 creates a process whose image belongs to the list of files that originated from "cmd.exe". Note that this FList was produced by the "FileRelocation" operation (Node 4). Moreover, the process is created with standard input set to the duplicate/original handle of the IPC endpoint, server or client.

Let us compare the generalized AD of the "Remote Shell" functionality (Figure 5) with the original AD (Figure 2). The generic specification defines six different realizations from the two original ADs. All generic realizations are effective against the obfuscation techniques presented above. In spite of generalization, the structural complexity of the generalized AD is commensurable with the complexity of the original AD. In fact, the overhead imposed by generalization is managed via algorithm parameters. For instance, generation of the threshold parameter in `TraceProcesses`,to some degree, determines the overhead of the tracing functionality. This demonstrates the effectiveness and flexibility of our approach.

Understandably, the more obfuscation techniques we address, the more complex the generalized specification is expected to be, however, the specification is not yet a complete recognition mechanism since it merely represents how the functionality is implemented in terms of object manipulations. Hence, the efficiency of the recognition mechanism ultimately determines how many obfuscations we can address. We proposed a highly efficient way to detect the specified functionalities. The proposed recognition model is scalable enough to detect specifications with all currently known obfuscations.

4.5 Functionality Recognition

As indicated in the system architecture (Figure 1), the functionality recognition process consists of two stages. In the first stage, we recognize individual object manipulations by identifying their dedicated APIs in the system call domain. However, a manipulation may be performed through several alternative APIs operating on the same Kernel objects. Additionally, an API function may invoke several additional minor system calls that are not critical to the manipulation implementation. Hence, only the essential, semantically critical part of an API function should be recognized and attributed to the corresponding manipulation. This recognition approach is resistant to certain evasion techniques when the malware does not invoke the entire API but only its critical system calls, thus only partially realizing the API yet achieving the required manipulation.

In the second stage, we recognize functionalities through the identified object manipulations, i.e., APIs. Note that system calls represent APIs, and APIs represent functionalities in that are consistent with the AD (2). Hence, the same type of models can be employed to recognize subsystem object manipulations and malicious functionalities. The selection of a particular recognition model must be justified with respect to both expressive power and implementation efficiency (computational and memory complexity).

4.5.1 Justification of the Recognition Model with Respect to Expressive Power.

Consider the following simple functionality that could be a part of a virus: "open all executables in a folder; do not access files until some predetermined point in time; then check to see if all of them are ready for code injection; then if all the files are ready, inject the code, otherwise close all of the opened files". One realization of this functionality could utilize the CreateFile, ReadFile (to read the PE header) and WriteFile API (to inject code). Since this functionality requires the synchronization of file reading and file writing, then the sequence of the APIs invoked by the functionality would represent the following pattern:

$$\underbrace{\text{CreateFile, CreateFile, ... CreateFile}}_{\text{n times}},\ \underbrace{\text{ReadFile, ReadFile, ... ReadFile}}_{\text{n times}},\underbrace{\text{WriteFile, WriteFile, ... WriteFile}}_{\text{n times}} \quad (3)$$

This pattern constitutes a formal language:

$$L=\left\{\text{CreateFile}^{i},\text{ReadFile}^{\text{i}},\text{WriteFile}^{\text{i}} : i \in \text{Z}^{+}\right\} \quad (4)$$

According to Pumping lemma, this language is not context-free [45], but it can be generated by a context-sensitive grammar. We believe that a context-sensitive grammar can express all functionalities presented in AD formalism (2). Object parameters and handle values could be represented by a large non-terminal alphabet covering the entire parameter space (i.e. all possible values of the parameter's type). Hence, a functionality can be at least recognized by Linear Bounded Automata (LBA) that is an accepter for a context-sensitive language [45].

4.5.2 Justification of the Recognition Model with Respect to Computational Complexity.

According to [46], a LBA of size n can be simulated by a Place Transition Net (PT-net) of size $\text{O}\left(n^{2}\right)$. The LBA and equivalent PT-net would have identical time complexity for an acceptance problem. A PT-net can be translated into an equivalent CPN in such a way that the structural complexity of the PT-net (number of places) would be converted into the inscriptional complexity of the CPN (arc expressions). Since a CPN would have far fewer places, we prefer using the CPN rather than a PT-net. Moreover, a CPN has an advantage over a LBA when processing multiple instances of the operation chains (words): should a chain be executed more than once, an LBA model accommodates extra states for each instance of the chain. In contrast, a CPN represents an executed operation chain as one token residing in the corresponding place that allows for processing of multiple chain instances with low overhead. Consequently, a CPN was chosen as a recognition model.

A CPN could formally be defined as a tuple [47]:

$$\text{CPN}=(\text{S,P,T,A,N,C,G,E,I}) \quad \textbf{(4)}$$

where: S – color set, P – set of places, T – set of transitions, A – set of arcs , N – node function, C – color function, G – guard function, E – arc expression function, I – initialization function.

Next, we formulate a CPN configuration that provides execution semantics for the AD specification defined in (2). To recognize the functionalities specified in an AD, a CPN configuration must reflect the objects and manipulations. Additionally, we need to recognize several distinct functionalities that may or may not have common implementation patterns. Hence, CPN places must represent the following states: created objects, object manipulations, pseudo states routing the control flow of ADs, and individual functionalities.

The above considerations indicate that the *set of places* of the CPN should consist of four disjoint sets:

$$\text{P}=\text{P}_{obj} \cup \text{P}_{manip} \cup \text{P}_{fun} \cup \text{P}_{pseudo} \quad (5)$$

These disjoint sets determine the following four types of places:

Object place (P_{obj}) is associated with a unique OS object. In this place, a token represents an instance of the object associated with the place. Such a token is defined as a tuple: a descriptor (handle) of the object instance, and a set of necessary object parameters. Hence, the color set of Object-places typically constitutes a pair of two types: the system handle (unsigned int32), and the set of attribute types utilized in system calls for creating objects such as strings, int32 (for access flags), pointer, and others.

Manipulation place (P_{manip}) represents a particular operation (manipulation) on an object. Such a place contains tokens representing the successful execution of a corresponding operation. A token comprises a handle of the manipulated object and critical parameters of the operation represented by the place. Thus, the color set of a Manipulation place consists of the space of system handles and a set of selected operation parameters associated with the place.

Functional place (P_{fun}) corresponds to a unique functionality. These places contain tokens that represent the successful recognition of a given functionality. Note that functionalities represent not a particular object, but a pattern of manipulations on several objects. The color set of Functional places includes only selected attributes of the necessary objects involved in the respective functionality as well as the objects' operation parameters that individualize the functionality.

Pseudo place (P_{pseudo}) is associated with the pseudo states of the AD. A manipulation/object place represents an executed object operation. An input transition of an object place must be attributed to the execution of one of the functionally equivalent APIs or system calls implementing the respective manipulation. An input transition of a functional place should be enabled when the corresponding functionality is executed. Hence, the set of transitions consists of three disjoint sets:

$$T = T_{man} \cup T_{pseudo} \cup T_{fun} \quad (6)$$

where T_{man} - manipulation transitions representing system calls or a subsystem level operation (exported by API); T_{pseudo} - pseudo transitions that are utilized to reflect AD pseudo states; and T_{fun} - functional transitions such that their input and output places constitute functionalities or functional object operations.

We should point out that technically, a functional transition would coincide with the appropriate system call in a discrete event scale. However, the occurrence of such a transition is a semantically important event, thus we deliberately do not associate it with a system call.

Each manipulation transition (T_{man}) is enabled upon execution of any of the equivalent APIs performing the manipulation. Consequently, the guard expressions of such transitions must be defined over the object descriptor space (handle and buffer address) as well as over the manipulation parameter space. Guard expressions ensure that only manipulations with parameters determined in the corresponding AD would enable the transitions. The expressions of the output arcs may include variables of any type from the color set which covers the necessary attributes of the system calls and API functions. This provides enough flexibility to distinguish similar, yet semantically different functionalities.

We developed procedure "ADtoCPN" that produces a CPN from the given functionality AD (see listing 4). Such a CPN possesses the necessary execution semantics to recognize the functionality. Here we only outline the high-level steps of such a procedure.

Listing 4. ADToCPN procedure

```
Procedure ADtoCPN
Input: F – an AD of the functionality defined by the formalism (1).
Output: CPN – a CP-net that recognizes the given functionality F.
```

```
1. Compose the CPN structure (P, T, A) corresponding to the constructs of
the AD of the functionality. Replace the AD arcs with transitions and
replace the nodes with places.
```

```
  1.1 Form a set of places P and set of transitions T that correspond to the
  state and pseudo state nodes of the functionality F.
  1.2 Form a set of the CPN arcs (A) connecting the places and transitions
  created in the previous step (1.1)
  1.3 Form a set of functional places, transitions and corresponding arcs.
```

```
2. Define place colors (C), guard expressions (G) and arc expressions (E)
that define execution semantics of the functionality F in the given domain.
```

```
  2.1 Define guard expressions of the manipulation transitions that check
  the executed manipulation parameters against parameters specified in the
  functionality's AD.
  2.2 Define guard expressions at the transitions that represent branching
  arcs of the AD decision nodes.
  2.3 Define a color function (C) that would reflect variables of the
  functionality.
  2.4 Define arc expressions representing variable assignment in the
  functionality's AD.
  2.5 Induce Color set (S) and the rest of the arc expressions from the
  color function (C) and the CP-net structure (P,T,A)
```

```
3. Compile a CP-net (CPN=(S,P,T,A,N,C,G,E,I)) from the component sets obtained
in steps 1 and 2.
```

Consider low-level CPNs recognizing subsystem object manipulations in the system call domain. These CPNs are obtained from the system call level ADs specifying object manipulation. Here, manipulation transitions are enabled by system call execution and therefore it is an open network driven by external events (OS calls). Moreover, manipulation transitions do not have input arcs representing *inlet* transitions. We also distinguish outlet transitions that represent handle/object elimination. For instance, the *NtClose* system call enables an outlet transition that destroys a token from the corresponding place.

Using procedure ADtoCPN, we obtained both high and low level CPNs for "Remote Shell" functionality (Figure 10).

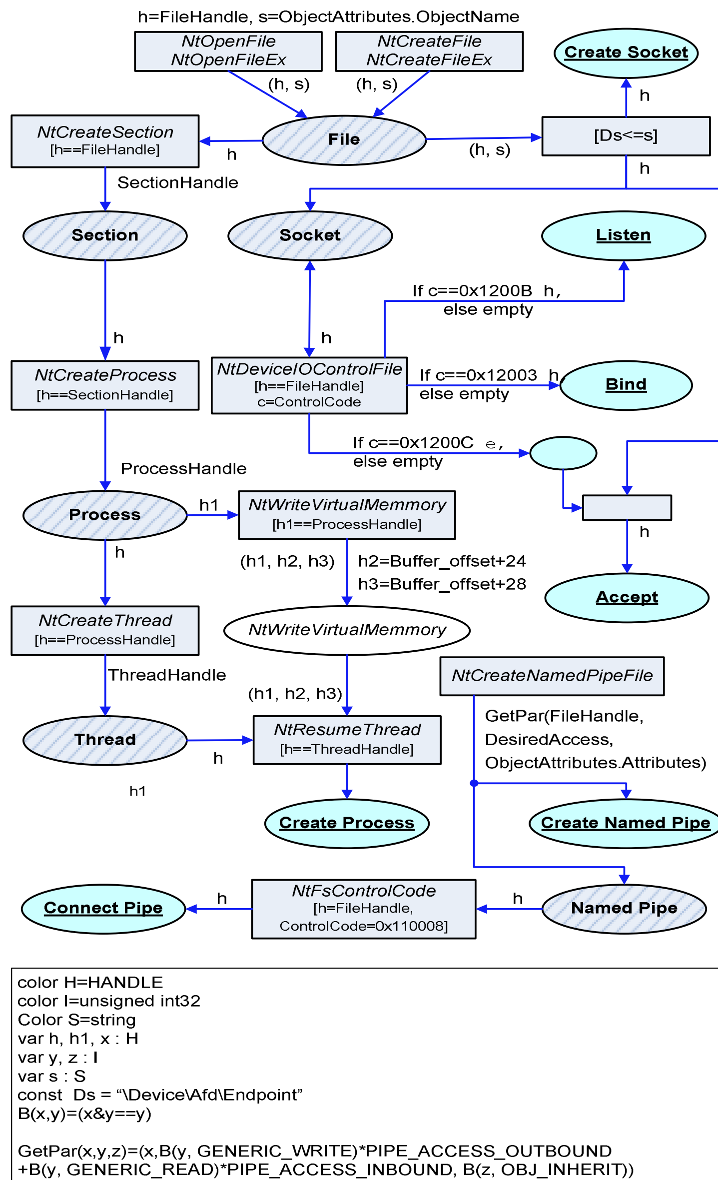

Figure 10. High Level (Subsystem Level) CPN for the "Remote Shell" Functionality

Figure 10 shows the high-level CPN. where places are shaped as ellipses and transitions as rectangles. The CPN node indices correspond to the AD nodes they recognize. For instance, transition # 7.1 and place #7.2 will recognize Node #7 of the Remote Shell AD. In Figure 10, the cloud shapes symbolize external CPNs such as Remote IPC CPN, Low level CPN, and others. These external CPNs recognize the corresponding functional/ subsystem manipulations and enable relevant transitions. For instance, transition #1.1 is enabled when functional object "Remote IPC" is created. The transition's guard, "PID in PIDList", checks whether the process performing "Remote IPC" belongs to the list of descendant processes. This requires tracing a generated process as specified in the AD in Figure 9. The process tracing is performed by the "Process Generation" CPN that provides a descendant PID list as tokens to transition #7.1. The Object places are highlighted with bold fonts. Place #5.2 ("RemoteShell") is a recognition/functional place that represents a successful functionality recognition.

A low-level CPN is shown in Figure 11. It recognizes the following subsystem level manipulations: "Create Socket" (socket API), "Bind", "Listen" (listen API), "Accept" (accept API) which are exported by ws2_32.dll; and "Create Named Pipe" (CreateNamedPipe API), "Connect Pipe" (ConnectNamedPipe API) which are exported by kernel32.dll. In parentheses, we provided an API function that belongs to the group of equivalent subsystem APIs performing the associated manipulation. The CPN has three color sets (types): handle (H), whose variables represent object handles; string (S) for the file names, and uint32 (I) for access flags. Color and variable declarations are written using CPN markup language (CPN ML) syntax [47]. The CPN has 10 inlet transitions corresponding to system call execution. These transitions generate tokens representing attributes of the system calls that are processed by the CPN.

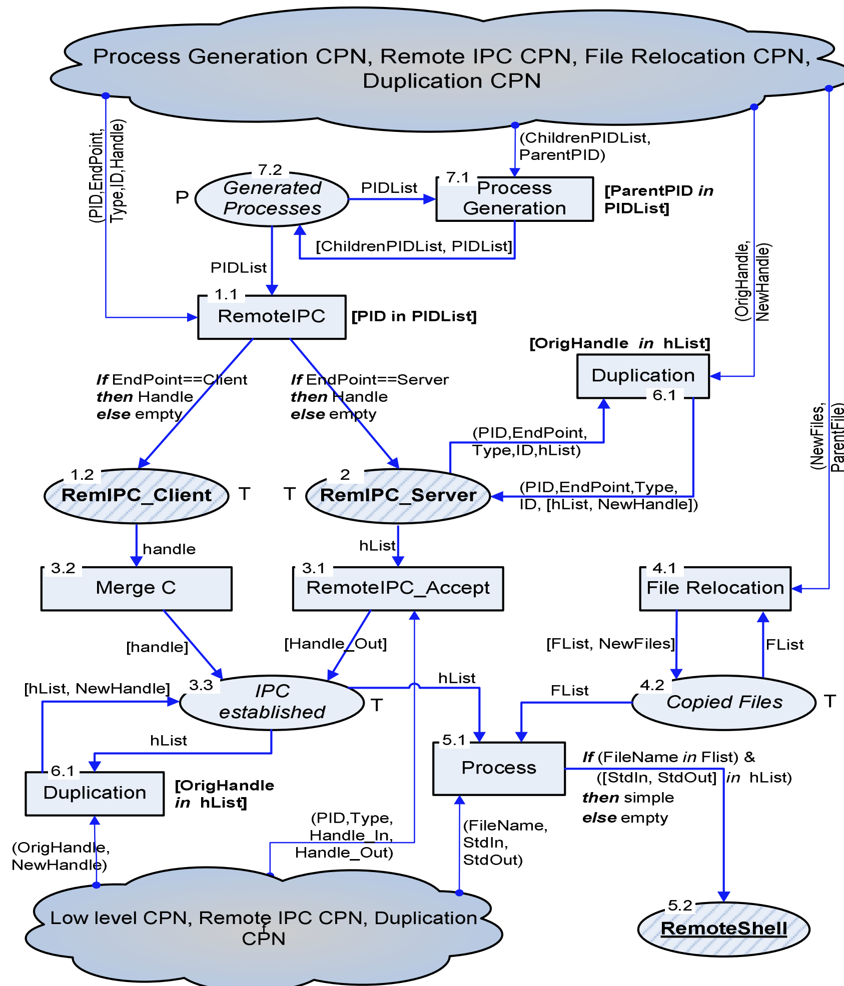

Figure 11. Low Level (system call level) CPN for the "Remote Shell" Functionality

It can be seen that a CPN's structure is very similar to the structure of the AD. A CPN is a very efficient recognition mechanism due to token dynamics. Hence, a CPN causes a minimal performance penalty for the anti-obfuscation generalization we introduced in Section 3. This ensures that our approach is highly scalable, allowing us to address most of the known high-level obfuscation techniques.

4.5.3 Dynamic Information Flow Tracing

Depending on the specification, our IDS could employ a coarse-grained detector or fine grained detector. Coarse-grained detectors only trace system call execution discarding information dependencies. Fine-grained detectors trace information flows using dynamic data tracing techniques such as the taint propagation [48, 49], thus potentially providing additional discriminative power. However, it was shown that purely dynamic techniques cannot trace data transmitted through covert channels such as implicit flows [50]. A particular attack on the taint propagation technique was described in [43] using implicit flow techniquethat is hard to defend against as confirmed in [51]. Since the implicit flow allows transmitting a bit by *not* executing a branch conditioned by a tainted value, control flow analysis techniques such as [48, 49] are useless in this case. Note that the static analysis of the *non* executed branch would not help either because, in general, the branched code could be encrypted. Forced execution of such a branch may fail if it has implicit jumps depending on tainted values.

In general, malware can evade data tracing by using an implicit flow that is easy to implement [43]. However, malware cannot avoid using system calls. Consequently, dynamic information flow tracing would not decrease false negatives compared to purely system call based detection. On the other hand, taint propagation may decrease false positives since there is no reason for legitimate software makers to use covert channels in their codes unless they want to protect their products against reverse engineering.

The proposed AD formalism (2) allows for specifying functionality with informational dependency between the operation attributes. Recognition of such a functionality would require the utilization of the taint propagation engine [48, 49] coupled with the system call monitor.

4.5.3.1 Taint Propagation Engine

While implementing the tainting engine, we generally followed the methodology given in [52], [48], [51], and [49] but our implementation differs in the following aspects:

- Taint source and sink utilization
- Taint dependencies and propagation

4.5.3.1.1 Taint Source and Sink

In our system, the objective of the tainting engine is to trace information flow between object operations. In the AD formalism (2), an information flow can be specified through a variable x that is referred to by the content through a transformation $T(x)$. In this case, the source of the information flow is the operation whose output attribute defines the variable. The destination of the flow is the operation whose input attribute depends on the content of the variable. Note that in the case when the variable representing the flow is referred to by the content in several operations/attributes, an information flow may have multiple destinations.

Functional object operations are based on subsystem operations that in turn are implemented through APIs and system calls. Hence, technically, information flow tracing is initiated by tainting the output argument of a system call implementing the source operation. At the same time, the flow is recognized by checking the taint of the input arguments of the system calls realizing the destination operations. To avoid false positives, we utilize a unique taint label for each particular instance of the specified information flow.

4.5.3.1.2 Taint Dependencies and Propagation

As described in [51], our system propagates the taint label according to three dependency vectors: explicit data flow, system call and control flow.

For the x86 architecture, data flow dependency could be represented by data transfer and stack instructions such as MOV, MOVX, PUSH, POP, etc., or arithmetic and logical instructions such as ADD, SUB, AND, OR, etc. In the case of direct data transfer, the engine propagates tainted bytes of the source to respective bytes of the destination. However, if the source is a register and it is tainted, the engine marks all of the destination bytes. Note that the source operand could be indirect, for instance MOVZX ecx, word ptr [ecx+eax*2]. In this case, the engine taints all of the destination bytes if either the index or displacement registers of the source is tainted. Such a policy enables tracing array manipulations indexed by tainted values.

Control data flow usually takes place when a variable is assigned within the scope of *if*, *else* or *switch*, case blocks that are conditioned by the tainted value. To resolve such a dependency, we tried to follow the methodology in [49] that implies tainting every destination within the scope of the conditioned branches. However, based on our experience, sometimes one cannot take into account the entire scope, because if any of its branches lead to the return of the current function, the system taints everything in the rest of the function resulting in false positives. In such a case, we mitigate false taint propagation by pruning such branches from the flow graph and limiting the depth of the scope.

System call dependency is represented by data processing system calls/APIs such that they do not perform any system related activity and are only responsible for generating output data from the input data. Such data processing system calls are best exemplified by the RTL functions RtlInitAnsiString, and RtlAnsiStringToUnicodeString. For instance, RtlAnsiStringToUnicodeString(outbuff, inbuff, …) creates a null terminating UNICODE-string (outbuf) from an input null-terminated ANSI-string (inbuff). Upon execution of this system call our engine would respectively taint the Unicode characters (words) of the output string (buffer) corresponding to the tainted ANSI characters (bytes) of the input string (buffer). Note, here we perform a one to one tainting to exclude false taint propagation. Some system calls may untaint the input argument. For instance, if RtlFreeHeap is to be invoked, then the input buffer is freed from the heap causing our system to untaint the content of the buffer.

Unlike [49], in our system, a particular taint label may become obsolete (retired). When the information flow is recognized, the taint label of the flow retires, meaning that the system will untaint any object tainted by the retired label. A label may also become obsolete if the system identifies that the source system call was executed in the frame of a wrong (non-source) operation. The latter situation may occur if two different operations begin with the same subsequence that originated in the source system call and then split. In this case, the operations only will be recognized at the end of the execution, after the source system call. That means that the engine must start tainting before recognizing the entire operation.

4.5.3.2 Taint Utilization in CPN

While the taint engine is responsible for taint propagation, in order to recognize the information flow in the operation session the CPN-based recognition engine is responsible for taint label management. Upon execution of a system call, the corresponding enabled transition creates a token representing the system call. If the system call is the source of a specified information flow, the transition should also signal the taint engine to create a new taint label and add the label to the new token as a field. This way we transmit the taint label of this particular

information flow's instance. When the destination system call is executed its transition also checks to see if the taint label of the input token is equal to the taint label of the system call's input attribute. If the labels match (meaning that the instance of the information flow reached its destination) the CPN recognizes the flow itself by enabling the corresponding transition and firing a token to the recognition place.

Token dynamics play a critical role for the efficiency of the information flow recognition. Since the taint label becomes a part of a token, the recognition mechanism verifies the taint label in only two transitions corresponding respectively to the source system call and the destination system call. In other transitions where there are no information flow endpoints, the taint label is not verified. This separates the tainting engine from the recognition engine, thus achieving the optimal overhead (complexity) distribution.

4.6 IDS Implementation

4.6.1 AD Designer

According to the architecture presented in Figure 5, an expert has to specify and supply activity diagrams of the functionalities defined in terms of the AD formalism (2). UML 2.x AD syntax provides enough constructs for specifying all components of the functionality formalism [53]. The state nodes are represented as UML actions, complex functional nodes as UML activities, and object operation attributes and variable assignments as UML tag values. Additionally, UML syntax allows for using so-called stereotypes that are convenient for creating simple node profiles that define the set of tagged values. For each object operation, we use an individual stereotype that determines a set of attributes in the form of tagged values. In our implementation, the choice of the UML AD designer is not critical as long as it is strictly compatible with the UML 2.0 standard. In our experiments, for UML we used Visual Paradigm, a commercial software that offers free community releases [23]. After finalizing the AD design, the expert exports the AD to the UML XMI, a format that is used to exchange diagrams among UML compatible applications.

4.6.2 Specification Generalizer and CPN Constructor

We utilized the Python language to implement prototypes of the Specification Generalizer and CPN constructor modules. The script for the Specification Generalizer module constitutes 710 lines of code and implements all three generalization algorithms and the specific functions defined in Appendix D. We also developed a function that imports a formal AD from the input UML XMI file created by the UML AD designer (Visual Paradigm). The importing is performed by interpreting and mapping UML constructs (e.g. tag values, actions, activities) to corresponding AD components (e.g. variables, object instances, operations) as defined by formalism (2).

Prior to execution, the Specification Generalizer module imports the input functionality AD along with the generalization functionalities' AD and the functional operations that were pre-designed in Visual Paradigm (see Appendices B and C). Then, the module applies the generalization algorithms to the input functionality and produces a generalized AD. Finally, the module exports the generalized AD to the XMI file. The resultant XMI file can be imported by the UML designer for on-demand editing or by the CPN constructor for producing a CPN recognition model.

The CPN Constructor module applies the ***ADtoCPN*** procedure to the given functionality AD to produce the recognition CPNs defined as a tuple (3). Finally, the Constructor translates the obtained Petri networks to a CPN ML like format and exports it as an XML file for the CPN recognizer modules.

4.6.3 Functionality Recognizer

We developed two versions of the functionality recognition modules. The first implementation was intended to evaluate the scalability and runtime efficiency of the methodology. The CPN recognizer was implemented in 3,500 lines of native C++ code. Here, the CPN configuration is mostly hardcoded and its modification usually requires recompilation. To minimize the complexity of token matching we utilized self-balancing trees to store tokens in places. The trees are indexed by corresponding colors (usually handles) utilized in guard expression of the output transitions. To store tokens that represent file derivatives (copies) of the original target file we utilized chained hash tables indexed by file names.

For the sake of efficiency, in this version the system call monitor operated as a Windows device driver. We used an SSDT substitution technique in the driver to hook Windows system services [3, 23]. Due to this type of driver, such an IDS is not completely transparent, however, the IDS's activity could be concealed through applying driver hiding techniques and covert user/kernel communications [23].

The second version is less efficient, but much more generic. For this version we developed a highly scalable and generic CPN simulator in C# .NET with Linq extension. The source code of the most of the CPN simulator components is available at http://apimon.codeplex.com. The program package includes several projects constituting 7,900 code lines in total. The projects are responsible for system/API call hooking, call data parsing and transmission, and CPN simulation. The CPN is built for simulation by translating arc and guard inscriptions into generative and filter expressions backed by Linq objects.

For the sake of performance and operability, we introduced some simplifications to the CPN simulator. The first simplification is that we treat the CPN as an open model that is fed with tokens from external systems such as the system call monitor and the taint propagation engine. Next, we did not implement binding of variables belonging to different arc expressions in order to avoid computationally expensive cross list matching. Finally, we eliminated the possibility of specifying the number of tokens retrieved by an arc from a place.

In spite of these simplifications, we preserved most of the CPN execution semantics. In particular, we treat arcs as token generators and guards as token binders. Hence, the CPN simulator is not limited to any particular execution domain and can process events of any nature from multiple sources. For example, the CPN simulator can process system/API calls, API functions, and functional object operations supplied from other CPN simulators. Such diversity allows us to build and simulate complex, hierarchical CPNs with low execution overhead.

4.6.4 Taint Propagation Engine

For the prototype, we did not attempt to achieve low tainting overhead because we were primarily interested in evaluating the recognition mechanism in tracing information flows specified in the functionality. The taint propagation engine was implemented using an IDA debugger with a IDA python debug management script. The engine runs the traced process in debug mode and analyzes each instruction and its operands. For each library function call, our system resolves the name of the function and input attribute. When a function is called, the IDA debugger breaks with a *dbg_step_into* event and passes control over to our script. The script

verifies the function entry address and parses its attributes as well as disassembles the body of the function.

To determine entry point addresses of the system calls our script verifies whether the process loaded the native system library (“ntdll.dll”). To achieve this, every time a library is loaded, the debugger breaks with *dbg_library_load* event, then our script parses the ntdll.dll image and records its export functions entry addresses aligned to the base address of the loaded module.

In our system, an expert has to provide system call declarations that are used as call dependencies as well as related structure declarations. The script parses standard declarations so that the expert may directly feed the engine with declarations from the MSDN website. Moreover, the expert has to provide dependencies between input and output attributes of each particular system call that is a call dependency. Such dependencies are specified in a simple XML format.

4.7 Conclusions

In this chapter, we addressed present and future limitations of the current Behavior Based IDS (BBIDS) associated with signature expressiveness, behavioral obfuscation and detection efficiency. We advocate for the separation of the specification and detection domains. We presented a new approach for formal specification of the malicious functionalities based on activity diagrams defined in an abstract domain. We introduced so-called abstract functional objects that along with system objects can be used for creating generic specifications that cover multiple functionality realizations while preserving perfect discriminatory power. We developed and tested an automated procedure, enabling human experts responsible for the formulation of malicious behavioral patterns to concentrate on conceptual realizations while omitting certain implementation details.

We analyzed and classified possible behavioral obfuscation techniques, both inter-process and intra-process, that can compromise existing BBIDSs. As a mitigating solution, we suggested the concept of specification generalization that implies augmenting (generalizing) otherwise obfuscation prone specification into a more generic obfuscation resilient specification. Generalization algorithms that make AD immune to the obfuscations were developed.

We proposed a methodology that uses a CPN to recognize functionalities at the system call level. Moreover, we developed an approach for incorporating information flows into a CPN to achieve fine-grained recognition. Finally, we proposed an automatic procedure for converting a given AD into a CPN that recognizes the defined functionality in the system call domain that was enriched with information flow data. In the end of the chapter, we described an implementation for all of the IDS modules.

5 EVALUATION OF THE DEVELOPED IDS

5.1 Experimental Setup

An experimental evaluation of the technology described in this document was conducted on the virtual network testbed at Binghamton University [2], [3]. The testbed was configured for a virtual network comprising dozens of victim hosts represented by virtual machines with vulnerable versions of the Windows OS and our prototype IDS. Using the testbed, we experimented with various types of replication engines as well as malware payloads representing the following set of potentially malicious functionalities:

Replication engines:

- Self code injection – A malware infects an executable file through injecting its own code into the executable body and replacing code entry points. It is used by file viruses.
- Self mailing – A malware emails its image as an attachment. It is used by e-mail worms
- Executable Download and Execute – Downloads a file from the Internet and executes it. Used as a part of self-propagation engine of network worms [2], hence it is exposed by exploited processes and network bot agents such as Trojan-downloaders.
- Remote shell – Described in Section 2. Used as a part of propagation engine for network worms; also exposed by network bots.

Malicious payloads:

- *Dll/thread injection* – Injects DLL/thread to the address space of a process. Used for password stealing or process control highjacking.
- *Self manage cmd script create and execute* – A malicious process creates a command script and executes it by running command interpreter. The command interpreter performs various operations on the malware image/dlls after its termination. This functionality relocates/deletes the malware image to conceal its footprint. Afterwards, a command script usually erases itself.
- *Remote hook* – Sets a remote hook into a victim process for a particular event; used for keylogging.
- *Password stealing* – Steals credentials and sends them to the Internet. This functionality is discussed in the next section.

These functionalities were specified, generalized and translated to CPN. Since functionalities share the same object operation sessions, to decrease simulation overhead we eliminated CPN structural redundancy by integrating the high level CPNs into a single universal CPN having several functional places recognizing all given malicious functionalities. The low-level CPNs were also integrated into a single Petri network capable of detecting object operations involved in the functionalities. The CPN configurations were then loaded into the Recognizer modules of the IDS.

We experimented with the malware known, according to AV descriptions, for perpetrating at least one of the malicious functionalities in order to verify the detection rate. The selected malware set included:

- *File viruses* – 7 instances (W32.Neo, Abigor, Crucio, Savior, Nother, Halen, HempHoper)

- *Network worms* – 10 instances (W32.Welchia.A, Sasser.C, Bozori, Iberio, HLLW.Raleka.A, Alasrou.A, Kassbot, Shelp.A, Blaster, Francette)
- *E-mail worms* – 9 instances (5 variants of w32.Netsky and 4 variants of w32.Beagle)
- *Network bots/Trojans* – SpyBot.gen, IRC.SdBot, RxBot families, Win32.Banker, Win32.lespy

We ran each malware image in the corresponding environment allowing it to execute its payloads and/or replicate properly. The replication activity was exposed when victim hosts were attacked by various worms [2, 13]. The set of test worms included strains that had been modified to assure their propagation success, as well as unmodified strains to assure test fidelity. To invoke the malicious payloads, we executed malware in certain preset conditions, e.g., we established an ftp/tftp server for the "executable download and execute" functionality. In some cases, we had to force malware strains to run their payloads by using debugging and run-time code modification.

We ran multitude of benign software including web-browsers, messengers, email clients, file utilities, network and system utilities and office tools in order to evaluate the false positive rate. We ran the tested software under various conditions/inputs to expose their functionalities. We should point out that our experiments did not cover all execution branches of the tested programs because some of them were missing certain minor behavior patterns. Nevertheless, we believe that in our experiments the tested software successfully exposed all of the main activities.

5.2 Detection Results

Tables 6, 7 and 8 capture the main results of our experiments. The upper part of Table 6 presents detection results for the legitimate software. Each cell indicates how many programs were detected based on the given functionality.

The lower part of table 6 features results for malicious software. For each malware set, we indicate how many instances possessing the given functionality were detected. For example, 4/4 means that there were four malware instances exposing the given functionality and all four were detected by our IDS.

The rest of this section discusses in detail results for the false positives and negatives.

5.2.1 False Positives

To assess the false positives, we performed two experiments.

In the first experiment, we manually ran a diverse set of 210 legitimate programs including web browsers, e-mail clients, system tools, file managers, office tools, hooking software, etc. We did not traverse all functionalities in all of the tested software because we were only focusing on main features of each tested program.

Table 6 indicates that eight programs out of 210 that showed false positives. Indeed, some known malicious functionalities could be exposed by certain legitimate software due to the following reasons.

- *Executable Download and Execute.* This functionality can be performed by Internet browsers or file managers, mostly on behalf of the end-user. In addition, many programs perform periodic checks for updates, if there is an update available, the program downloads and automatically executes it. This activity can also be tagged as download and execute.

- *DLL/thread injection*. This can be performed by user/system monitoring software. In particular, the Easy Hook library injects a DLL to trace API calls performed by an arbitrary program. The WinSpy program performs a DLL injection in order to retrieve the window objects data of a foreign program.
- *Self manage cmd script create and execute*. To uninstall hooks, the Easy Hook exiting functions run a *cmd* script that waits for the hooking process to end, then removes the hooked DLLs.
- *Remote hook*. Hooking can be performed by chat programs to indentify whether a user is idle. These programs hook into other processes for the input events such as a keystroke or mouse message.
- *Self-mailing*. When a user opens the “Save/Open file” dialog window, many programs show every file found in the directory by the proper icon in the dialog window. In this case if a user browsed to the programs image location in the dialog window right before sending an e-mail with attachment, the e-mail clients may show up as a false positive. However, such behavior coupled with the sending of email using a client image is considered to be atypical user activity. Since this type of situation only happened in a particular scenario artificially performed during testing, we did not attribute it to a false positive of the entire e-mail client program.

Table 6 clearly demonstrates the difference in discriminatory power of various functionalities that are frequently exposed by malware. According to Table 6, self-code inject, self-mailing and remote shell were never exposed by benign software, thus they have near perfect discriminatory power and can be used for malware detection. However, “Executable Download and Execute” (“ED&E”) that is exposed by benign software such as a web browser has low discriminatory power, hence it cannot be recommended for signature-based detection. Regardless of the discriminating power, our experiment demonstrates the ability to reliably detect individual functionalities. This ability could be beneficial for the detection of complex malicious payloads, such as password stealing, that may involve the combined use of several interrelated primitive functionalities.

Table 6. Functionalities Detection Rate and False Positive Rate

			Self-replication		Replication/ payloads		Payloads		
			Self code inject	Self mailing	Exec. Download & Execute	Remote Shell	Dll/thread injection	Self manage cmd script create & execute	Remote Hook
Legitimate software	203	Windows system tools, office apps, other utilities					1	1	
	2	Web browsers (Opera, IE)			2				1
	2	E-mail clients (Outlook Express, Eudora)		2(?)					
	1	Instant messaging client (Yahoo messenger)			1				1

	2	File managers (FAR,Win Exp)			1				
		Total detected			4/210		1/210	1/210	2/210
Malware	7	File viruses	7/7						
	10	Network worm shell codes			2/2	8/8			
	6	Network worm payloads			4/4	1/1	1/1		1/1
	9	E-mail worms		9/9					
		SpyBot.gen family			all	all			all
		IRC.SdBot family			all	all		all	all
		RxBot family (11)			all	all	all	all	all
False positive			0%	0%	1.92%	0%	0.48%	0.48%	0.96%
Detection rate			100%	100%	100%	100%	100%	100%	100%

The second experiment was performed utilizing a large set of system tools. The purpose of this experiment was to verify whether the MS Windows package has programs that expose malicious functionalities in their main operational mode. This was achieved by automatically running all binary executables from the Windows system folders (C:\Windows\ and c:\Windows\System32\). For each program, our system performed the following steps:

1. Create suspended process
2. Initiate CPN simulator and system call monitor
3. Resume process and collect data until program finishes execution or after 20 second timeout
4. Write place reachability statistics to a report file
5. Clear Petri Net contents for the next run

In total, our system ran 339 programs located in windows folders. All CPN statistics reports are summarized in Tables 7 and 8. Table 7 features the reachability statistics for the low (system call) level CPN (see Figure 5). The low level CPN recognizes object operations exported by the subsystem API. Hence, it has a recognition place for each necessary subsystem API exported by kernel32.dll and ws2_32.dll. In table 7, the first column shows the names of the libraries whose API are recognized by the CPN. The second column shows name of the API functions that export object operations recognized by the CPN. The third column shows the number of programs that invoked a particular API resulting in successful recognition of the corresponding API/operation. For instance, the table shows that CPN recognized the kernel32.WSASocket API in the system call flow in 8 out of a total of 339 programs.

It can be seen that the reachability for places associated with system call execution is higher than the reachability of the API (object operation) places. This happens due to the fact that a single API may repeatedly invoke several system calls resulting in many tokens for each system call. However, during the process of API recognition, most of the CPN transitions take several system call tokens and fire only one API related token. Note, since the number of processed tokens decreases towards the recognition place, the CPN simulation overhead also decreases while as we near the moment of functionality recognition.

Table 8 presents reachability statistics for the high (subsystem) level CPN. The low level CPN provides the high level CPN with tokens that are associated with system object operations

involved in particular functionalities. In the table, the second column depicts the name of the operation or functionality that is represented by its respective recognition place. Similar to Table 7, the third column of Table 8 shows number of programs that reached the particular place associated with the functionality or object operation.

Table 8 presents five functionalities of interest (marked by a grey background): *self code inject*, *self mailing*, *remote shell, and executable download and execute*. We should point out that each functionality is represented by a recognition place in the high level CPN. Therefore, a functionality is detected in the program if a corresponding CPN place is reached by a program during the test.

Tables 7 and 8 indicate that most programs opened files and read/wrote some data, however only a few accessed files of interest such as executables or libraries. Several programs created a socket and established a connection, however, none of them utilized that socket for remote shell or to download an executable. As a result, CPN recognition places (shaded rows) were never reached by any of the 339 programs tested, indicating that there were zero false positives exposed.

Table 7. Place Reachability Statistics for Low Level CPN

Library	System calls/API	# of programs reached the place
Ntdll (System call)	ZwClose	193
	ZwCreateFile	119
	ZwOpenFile	106
	ZwReadFile	94
	ZwWriteFile	64
	ZwCreateSection	50
	ZwMapViewOfSection	151
Kernel32 (API)	CreateProcess	36
	CreateNamedPipe	0
	ConnectNamedPipe	0
WS2_32 (API)	WSASocket	8
	connect	6
	bind	7
	listen	1
	accept	0
	send	4

Table 8. Place Reachability Statistics for High Level CPN

Object	Operation/functionality	# of programs reached the place
File	Map file	38
	Read file	15
	Write file	4
	Read itself of map itself	0
	Write to executable file	0
	Inject self-code	0
	Start process from edited or created executable	0
Named Pipe	Pipe created and connected	0
	Remote shell via named pipes	0
Socket	Socket connected	6
	Download and execute	0
	Socket bound	7
	Socket listening	1
	Accepted sockets	0
	Remote shell via socket	0
	SMTP protocol	0
	Self-mailing	0

The goal of the second experiment was to verify windows system tools using standard inputs in standard operating mode. Our experiments showed zero false positives. It seems that the reason for zero false positives could be attributed to the fact that windows tools only have only necessary and limited capabilities strictly defined by the purpose of the tool. Therefore there is no reason for redundancy on the functional level. For instance, the registry management tool would never download a file from a remote host, simply because this functionality could be achieved through another dedicated tool. Certainly, such tools would not perform unnecessary, potentially malicious, functionalities such as self code inject or self-mailing.

5.2.2 False Negatives (Detection Rate).

As Table 6.indicates, for each malware containing the given functionality, our IDS successfully detected the functionality with zero false negatives. Such a low false negative rate could be attributed to the signature generalization. For instance, the Beagle worm drops itself into the system folder, and then it e-mails its dropper. However, our prototype system

successfully detected this type of self-mailing activity because it traced the dropper as an object relocation functionality.

While creating the AD specifications for the tested malware, we observed an interesting fact - that malware strains within the same family rarely demonstrate a conceptually novel realization. Instead, new malware strains frequently introduce minor alterations to their functionality realizations such as the utilization of alternative APIs or changing a Local IPC, i.e., switching from a named pipe to shared files. We see two reasons for this trend. First, the attackers try to change the malware system footprint in order to avoid certain AV signatures. Second, in the case of botnets , they simply try to enhance the performance of malware by optimizing or simplifying their implementation.

5.2.3 Case Study - Password Stealing

Table 6 demonstrates that malware, such as botnets, carry many malicious functionalities as their payloads. Such diversity could be conducive to detection. We may target not just one malicious functionality, but rather a pattern of such functionalities that would determine the degree of hostility of the process.

As indicated in table 6, remote hooks had several false positives. While remote keystroke hooking may not be malicious (at least with chat programs), keystroke stealing is certainly malicious. The fact that the hooked (victim) process transmits some data to the master process and then the master process sends something to the Internet, is much more suspicious. This rather complex functionality can only be detected by analyzing the combined activity of both processes (master and victim) and correlating their invoked manipulations. In this case, such activity combines functionalities 7, 5 and Local IPC.

The functionality mentioned above is known as Password stealing and is presented in table 9. In the first step, the master malicious process sets keystroke hooks into the victim process. In the step 2, the hook handling function in the victim process transmits a keylog to the master process. Finally, in the step 3, the master process sends keylog data to the Internet. In the table, step 2 represents the combined activity of both the master and the victim processes. While steps 1 and 3 constitute individual activity, e.g. the master process does not need cooperation from the victim process to perform a remote hook or to send data to the Internet.

Table 9. Password Stealing Functionality

	Process 1 (master process)	Process 2 (victim, hooked process)
1	Hook to victim process keystroke events	
2	Establish Inter-Process Communication (IPC) with the victim process	
3	Send the data to Internet	

The CPN was designed with only functional objects recognized by external CPNs. The transition 1.1, 2.1 and 3.1 correspond to the activities in steps 1, 2 and 3 respectively in Table 9. These transitions are enabled upon recognition of the corresponding functional operations in the external CPNs such as: Remote Hook, Local IPC and Remote IPC. As shown in Figure 12, node 6 represents a successful recognition of the functionality. The remote Hook CPN recognizes the Remote Hooking functionality, which is step one in table 9. This CPN has one recognition node – named “Hook”. Each token in this recognition place represents the successful execution of a remote hook. The color of such a token defines the following: ID of the process that performed the hook, ID of the thread that is hooked, and the type of hook for an instance of the keystroke hook (WH_KEYBOARD). This CPN recognizes several realizations of remote hooking such as: DLL injection, direct windows hook and windows message parsing.

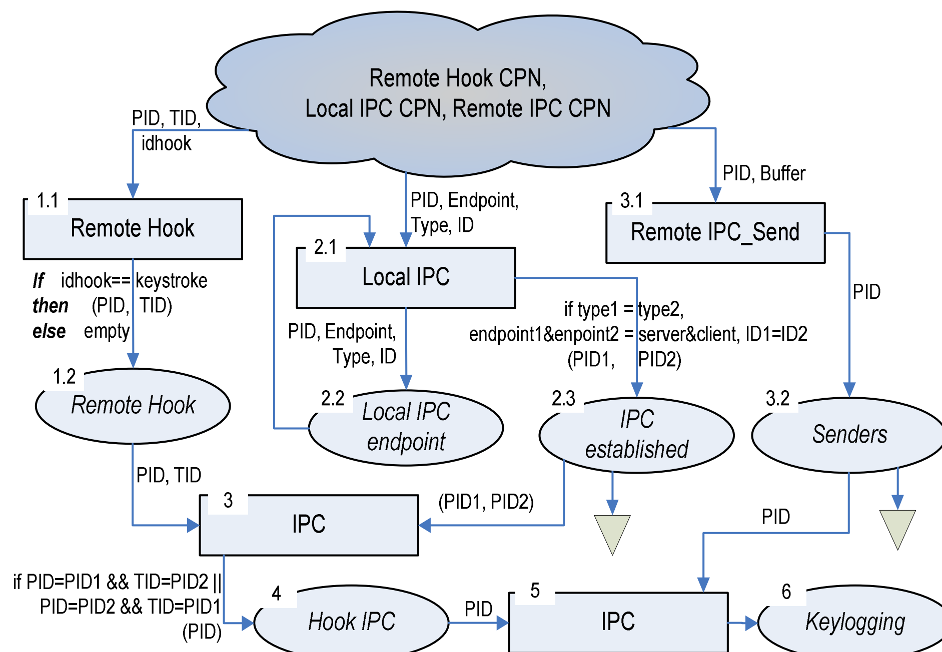

Figure 12. High Level CP-subnet for the "Password Stealer" Functionality.

Detection rate. We experimented with two families of malware that include four variants of the Win32.Banker and two variants of Win32.lespy. According to their description in viruslist.com, these malware expose a functionality that we can recognize, i.e. password stealing with IPC. Our prototype IDS successfully detected the password stealer functionality in all of the malware that we tested.

False positive rate. To estimate false positives we experimented with several popular programs: two messengers (QIP Infium, MS Messenger), two browsers (MS Internet Explorer, Opera), a file manager (Far), an email client (Outlook Express) and an automatic keyboard layout switcher (Punto Switcher). The results for place reachability of the CPN are summarized in table 10. This data indicates that all tested programs performed "Hooking" functionality (place 1.2 was reached) and that most of them opened Remote IPC (place 3.2 was reached) and sent some data. None of them connected to the process which they hooked to (place 4 was not reached). Hence, we did not observe any false positives on this set of software.

These results demonstrate that it is more effective to detect complex functionalities rather than primitive functionalities. This example shows the significant advantage of utilizing CPNs for processes behavior recognition - that is their ability to trace the activity of several processes in the context of a single CPN. Moreover, in the CPN, the necessary attributes propagate as token fields allowing for system call association by process and thread ID. This makes if possible to recognize an interposed (system-wide) activity such as password stealing that involves two processes (the master process and the victimized process with an injected DLL).

Table 10. Place Reachability of CPN for "Password Stealer"

	Remote Hook (1.2)	IPC established (2.3)	Senders (3.2)	Hook IPC (4)	Keylogging (6)
Far manager	v				
Internet Explorer	v		v		
QiP Infium	v		v		

MS Outlook Express	v	v	v		
MS Messenger	v		v		
Opera	v		v		
Punto Switcher	v	v			

5.3 Performance Overhead Evaluation

The scalability of our IDS depends on two main factors: the execution overhead of the monitored processes and the overhead penalty of CPN generalization. The first factor determines the quantitative restriction of our IDS, i.e. how many processes could be protected by our IDS. The second factor defines the qualitative restriction, i.e. how generic our IDS should be to address possible obfuscations.

Process execution overhead is mostly imposed by the system call monitor and to a much lesser degree, by CPN processing.

The system call monitor driver is always active in the Windows Kernel. When a system call of interest is invoked, the driver receives execution control from the system service dispatcher, reads the system call input parameters, invokes the original system call, reads the output parameters of the system call and returns execution control back to the dispatcher. Such reading and saving attributes contribute the most to process execution overhead.

Each system call of interest that is invoked by a process gets processed by the CPN. Hence, the more systems calls that we invoke per time unit, the higher the overhead that is imposed by CPN processing. However, our CPN execution semantics appeared to be very efficient in processing a large number of system calls.

Periodically, the CPN recognizer requests system call data from the monitor driver. Such User/Kernel communication imposes additional overhead that is minimized due to buffering system call data and desynchronizing system call input and output attributes. In other words, the driver does not wait for system call execution and may send input parameters before receiving the output parameters.

The IDS was executed in Windows XP Professional SP2 running on an AMD Athlon 64 X2 (2200 Mhz) processor with 2 Gb of memory. In our performance tests, we evaluated the overhead imposed by tracing different tasks and applications. Moreover, we estimated the performance penalty for tracing generalization functionalities caused by behavioral de-obfuscation.

5.3.1 Run-time Performance Analysis

We measured overhead of system and application tasks using commercial benchmarks and manual setup. To achieve consistent results on Windows XP, we deactivated the Windows prefetcher, scheduled tasks and only accounted for warm runs (to minimize cache influence). Some tests, such as file search and software installation were performed in a virtual machine where we reverted the machine to the initial snapshot state for each run.

The test results for the Remote Shell functionality are presented in table 11. For the sake of brevity, we only showa select set of standard tests that are representative of execution overhead. The table depicts two system tasks and three application tasks. These tasks intensively utilized OS resources (services) resulting in a large number of invoked system calls. Some tasks involved

user interaction with the GUI of the corresponding application. In these cases, we utilized TestComplete software [54] to simulate user behavior.

We also ran a series of benchmarks using the well-known PC Mark 05 suite [55]. Internet Explorer was tested using the Peacekeeper benchmark [55]. We ran each task/benchmark several dozen times with identical initial conditions and computed the mean value and standard deviation of the execution time/score assuming a normal distribution.

In order to estimate the qualitative scalability of our IDS we tested each task against two CPN configurations: Basic and Full. The Basic configuration covers alternative realizations of the functionality in question, but does not trace generic objects or obfuscations. In contrast, the Full configuration traces the necessary functionality generalizations and addresses all three obfuscations. To estimate the quantitative scalability, our IDS observed *all active* processes, but CPN recognizer in all performed tests.

For each task, table 11 shows: base execution time when the IDS is disabled (no system call monitoring or processing) and execution time when the IDS is enabled with both Basic and Full CPNs recognizing the Remote Shell functionality (with monitoring all active processes). One can see that even using the Full CPN IDS does not impose much overhead (less than 4% on average), while monitoring more than 50 (all active) processes. In fact, we also ran the IDS with highly loaded Windows XP (more than 100 processes) without any significant overhead. This result shows sufficient scalability to protect all processes of a modern OS.

It could be seen that generalization and de-obfuscation does not impose a significant overhead penalty (0.31% in average). Note that in some tests Base and Full CPN overheads were considered to be invariant under a statistical hypothesis with 80% power. This shows that our IDS is highly scalable and can address many additional behavioral obfuscations.

While the tasks in table 11 exposed some overhead, many other standard computationally expensive tests did not show any execution overhead. For instance, Matlab did not show any overhead because its benchmarks involved mostly memory manipulations and math computations which utilize few system services resulting in a low number of invoked system calls. Similarly, the MS Word search and replace task imposed significant overhead on the CPU, but virtually none on the OS itself.

Table 11. Execution Overhead due to IDS

	Benchmark/Application (Task discretion)	Execution (seconds / score)			Overhead (%)	System call count
		IDS disabled	IDS enabled			
			Basic CPN	Full CPN		
System tasks	Files Search (Search *.exe in c:\)	58.96 ± 0.907	62.01 ±1.04	63.66 ±2.04	**5.2** (Basic) **7.96** (Full)	250,168
System tasks	Application Installation (Install DirectX 9.0c)	112.3	The same	113.6	**1.15**	
Application tasks	MS Word (Save a big file as rtf)	35.9 ±0.787	The same	37.4 ±0.52	**4.18**	95,894
Application tasks	WinRar (Compress Windows system folder)	292	The same	298 (Full) 296 (Basic)	**2.05**	98,396

	Internet Explorer 8 (Peacekeeper Browser Benchmark, www.futuremark.com)	702 (Score)	665 (Score)	657 (Score)	**5.3** (Basic) **6.4** (Full)	
PC Mark 2005 (score)	Application loading (Mb/sec)	4.96 ± 0.0132	The same	4.87 ± 0.355	**1.84**	10,345
	Web page rendering (pages/sec)	2.0332 ± 0.04672	The same	1.8892 ± 0.1088	**7.08**	100,503
	File Encryption (Mb/sec)	36.827 ±0.134	The same	35.746 ±1.066	**2.93**	
	XP Startup (Mb/sec)	5.88 ± 0.022	The same	5.75 ± 0.28	**2.21**	7843
Average execution overhead		Basic CPN configuration *(with multiple realizations)*			**3.67%**	
		Full CPN configuration *(with generalization and de-obfuscation)*			**3.98%**	

5.3.2 Stress Test

The purpose of this test was to estimate the overhead of the IDS operating under a stress attack. The stress attack could be conducted by a malware in order to congest the IDS. Such an attack implies invoking many system call chains without closing handles causing the IDS to process all of the objects and bind their handles. Such an attack would only be successful if the malware could effectively congest our IDS before congesting the OS while keeping a low execution profile. In the case of congesting the OS, such malware would be forced to expose itself and could be detected and terminated by any system administration tool.

We utilized the Microsoft Performance Monitor (Perfmon) tool to measure the runtime overhead of the IDS. In this test, we evaluated the performance penalty for countering the obfuscation through object relocation. In particular, we measured the overhead imposed by the handle and file tracing functionalities that were introduced by the corresponding generalization algorithms. In this experiment, we ran accustom designed test program that opened 70 files (kernel32.CreateFile) in the Windows system folder and for each file it duplicated 20,000 handles (kernel32.DuplicateHandle) and 20,000 mappings (kernel32.CreateFileMapping). As a result, it creates 1,400,000 distinct file object handles and 1,400,000 mapping (section) object handles.

Figure 13 shows CPU usage for test program and our IDS module. Our results indicated that the test program consumes a substantial amount of CPU cycles (around 90%), while the IDS recognizer module imposed less than 2% overhead on average for the trace object relocation activity. Such a drastic difference in overhead can be attributed to the fact that each object creation and handle allocation imposes a certain amount of overhead due to parsing/updating the internal Kernel structures, manipulating low level objects by Object Manager and pre-processing system call attributes in the API implementation. Even for a simple handle duplication, the system call invocation requires user/kernel switching that is expensive for Windows OS. In contrast, the CPN handle binding only requires user mode memory manipulations with highly

efficient algorithms, e.g. balancing trees, or simple pointer resolution, e.g. hash tables. Hence, for each handle duplication or object creation the CPN imposes significantly less overhead.

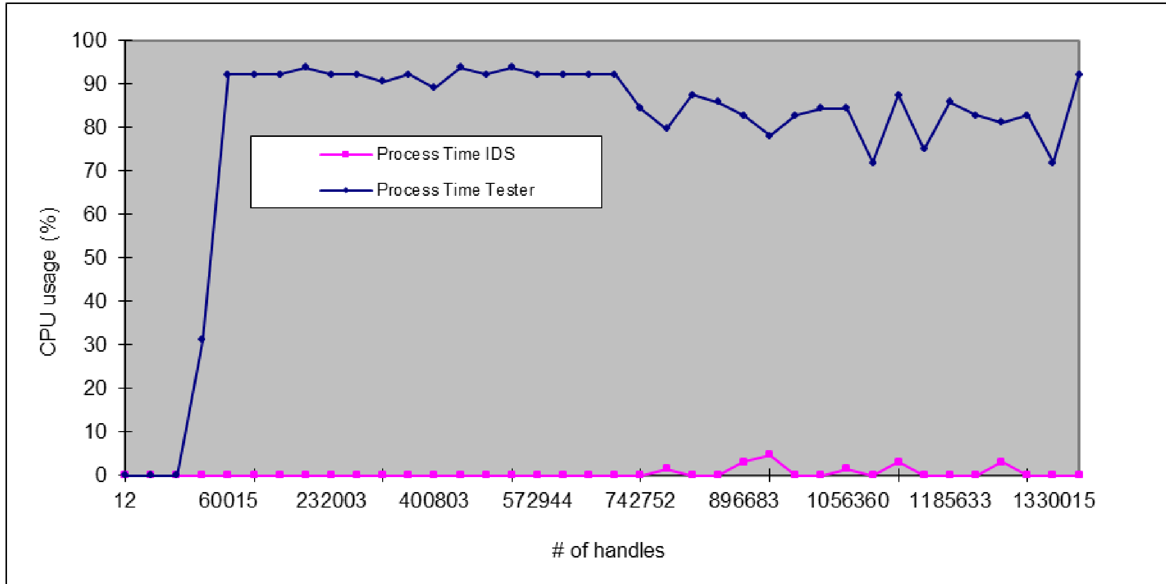

Figure 13. Handle Duplication Test

5.4 Conclusions

This chapter presented the experimental results for the signature-based prototype IDS. This IDS was evaluated on hundreds of legitimate programs and dozens of malware that had various types of replication engines and payloads. In general, the experimental results indicate low false positives and negatives. However, the experiment demonstrated the variability in discriminatory power of various functionalities that are frequently exposed by malware. The experiment also indicated that self-code inject, self-mailing and remote shell are never exposed by benign software, thus they have near perfect discriminatory power and can be used for malware detection. However, "Executable Download and Execute" is exposed by benign software such as web browser has low discriminatory power, hence it cannot be recommended for use in signature-based detection. Regardless of the discriminating power, the experiment successfully demonstrated the ability to reliably detect individual functionalities of any complexity. Additional experiments indicated that it is more effective detecting complex functionalities rather than primitive functionalities.

Finally, we performed a series of experiments to estimate the IDS runtime overhead using well-known benchmarks and manual setup. The results indicated two practical advantages. First, the IDS caused low overhead, which was less than 4%. Second, the overhead increase due to the anti-obfuscation generalization constituted only 0.3%. Such a low overhead difference between original and generalized CPNs indicates that an expert can always address many more obfuscation techniques with negligible execution cost.

6 SUMMARY AND FUTURE WORK

We are witnessing an on-going arms race in the cyberspace. The ever-increasing resources invested in the development of computer defenses are apparently outweighed by low-cost efforts of the hacker community. The term "asymmetric warfare" is perhaps the best way to describe the existing shaky balance between defensive and offensive forces in cyberspace. This report presented research on semantic approaches to malware behavior analysis. Such research aims at enhancing computer defenses, making them invulnerable to new, mutating and obfuscated malware. The developed approach is implemented and used to develop scalable IDSs.

In the second section, we studied modern threats and current anti-virus technologies. The analyses indicated that commercial host based malware detection technologies are not effective against sophisticated self-mutating malware.

In the third section, we introduced a taxonomy of malicious functionalities of typical malware that could be attributed to the essence of malicious activity. In particular, we analyzed basic self-replication mechanisms as well as several malicious payloads. Three types of the self-replication mechanism were discussed including: binary self-replication, server-side replication and client-side replication. Moreover, a wide range of malicious payloads was classified and analyzed. The study indicated that self-replication is an example of highly discriminative and indicative malicious functionality. Obviously, there is no reason for legitimate software to self-replicate since it can be distributed by legitimate means (e.g. downloads and install, trial etc.). Hence, self-replication has become of great interest to the network defense research community. One of our goals was to model self-propagation in order to investigate and estimate the possible impact of self-replicating software on network resources.

In the fourth section, we stated that malware maliciousness can be attributed to its goals, which can be viewed as high level functionalities. While a particular functionality may have several realizations, each realization would constitute a certain behavior. The behavior of each malware can be detected dynamically by observing its execution in a particular domain such as the system call domain. However, it is more important to infer high level functionality of the malware, rather than its simple behavior. To achieve this, one needs to address the three following aspects: signature expressiveness, vulnerability to behavioral obfuscation, and run-time efficiency of signature matching.

We justified the separation of the specification and detection domains. We presented a new approach for formal specification of the malicious functionalities based on ADs defined in an abstract domain (i.e. functional objects). We developed and tested an automated procedure enabling human experts responsible for the formulation of malicious behavioral pattern to concentrate on conceptual realizations omitting certain implementation details.

We analyzed and classified possible behavioral obfuscation techniques, both inter-process and intra-process, that can compromise existing BBIDS. As a mitigating solution, the concept of specification generalization that implies augmenting (generalizing) otherwise obfuscation prone specification into more generic obfuscation resilient specification was suggested. We developed generalization algorithms making our AD immune to obfuscations.

We proposed a methodology utilizing a CPN for recognizing functionalities at the system call level. Moreover, an approach for the incorporation of information flows into the CPN to achieve fine-grained recognition was developed. Finally, we proposed an automatic procedure

for converting a given AD into a CPN that recognizes the defined functionality in the system call domain, enriched with information flow data.

Our experimental results for a signature-based prototype IDS are presented in the 5th section. The IDS was evaluated on hundreds of legitimate programs and dozens of malware. In general, the experimental results indicate low false positives and negatives. However, the experiment demonstrated the variability in the discriminatory power of various functionalities that are frequently exposed by malware. The experiment indicated that self-code inject, self-mailing and remote shell are never exposed by benign software, thus they have near perfect discriminatory power and can be used for malware detection. However, “Executable Download and Execute” is exposed by benign software such as a web browser and has low discriminatory power, hence it cannot be recommended for a signature-based detection. Regardless of the discriminating power, the experiment demonstrated the ability to reliably detect individual functionalities of any complexity. Additional experiments indicated that it is more effective to detect complex functionalities rather than primitive functionalities.

Finally, we performed a series of experiments to estimate the IDS run-time overhead using well-known benchmarks and manual setup. The results indicated two practical advantages. First, the IDS causes low overhead which less than 4%. Second, the overhead increase due to the anti-obfuscation generalization constitutes only 0.3%. Such a low overhead difference between the original and generalized CPNs indicates that an expert can always address many more obfuscation techniques with negligible execution cost.

The experiments proved that signature-based behavioral approach appeared to be effective in detecting malware activity in the system call domain. While the anomaly propagation concept certainly has its advantages in decreasing false positive rate, it was observed that all such anomalies could be linked to various malevolent functionalities and detected as behavioral signatures. Indeed, in the behavioral domain the threat model is known and could be viewed as a set of malicious functionalities. This is especially true when the threat domain is represented as malicious functionalities and the normalcy domain as legitimate functionalities. Due to specific and well established goals, malware exhibits a very limited number of malicious functionalities. On the other hand, the number of legitimate functionalities is only limited by the imagination of software makers. Finally, a malicious functionality (threat) is known and deterministic and the only item that is not determined is a realization of the functionality (i.e. behavior). Hence, it is critical to specify and detect a functionality not just a behavior.

For future research, we plan to expand the list of possible behavioral obfuscation techniques and address them into the AD generalization. We intend to explore the increasing role of behavioral metamorphism as it implies the dynamic scattering of malicious functionalities among different benign processes so that none of the processes would have a consistent system call pattern, potentially resulting in offensive information warfare. We are interested in dynamic AD construction from the monitored behavior of processes of interest. First, this would allow for automatic retrieval of the functionalities for a particular program. Establishing a set of common functionalities representing the normal operation of a computer network, would result in a "customized normalcy profile" that will be invaluable for the development of dependable IDS. Second, the expansion of the data base of malicious behavioral signatures on the basis of automatic functionality detection, would result in enhanced misuse based IDS. The deployment of behavioral anomaly-based and misuse-based IDS would drastically improve computer defenses for "high value targets"

We believe that detecting malicious functionalities using generic signatures is the most promising approach. Such an approach raises the semantics of the detector from behavior to functionality, allowing us to identify classes of malware that achieve the same practical malicious goals. In other words, malware functionality represents the essence of maliciousness. Hence, detecting malicious functionality is the most accurate and precise method for distinguishing malware from benign software. Moreover, the proposed technology for dynamic functionality detection (CPN) was proven to be efficient enough for practical use in IDS.

REFERENCES

1. Internet Systems Consortium Domain Survey, https://www.isc.org/solutions/survey, accessed 2010
2. Tokhtabayev A. G., Skormin V. A. and Dolgikh A. M., "Detection of Worm Propagation Engines in the System Call Domain using Colored Petri Nets ", *In Proc. 27th IEEE International Performance Computing and Communications Conference (IPCCC)*, Austin, TX, Dec. 2008
3. Skormin V., Volynkin A., Summerville D., Moronski J. "Run-Time Detection of Malicious Self-Replication in Binary Executables" *Journal of Computer Security*, vol. 15, no. 2, pp. 273-301, 2007.
4. Wells, Joe (1996-08-30). "Virus timeline". IBM. accessed 2008-06-06.
5. "Defining Malware: FAQ", technet.microsoft.com, accessed 2009-09-10.
6. http://www.securelist.com/en/threats/detect/malware, Malicious Programs, accessed 03-Dec-2010.
7. http://www.symantec.com/business/security_response/glossary/, Symantec Glossary, accessed 03-Dec-2010
8. http://www.virusbtn.com, Virus Bulletin Magazine: Glossary, accessed Dec-2010
9. Szor, Peter, **The Art of Computer Virus Research and Defense**. Addison-Wesley, 2005.
10. Anti-Virus Comparative, "Proactive retrospective test", accessed May 2010
11. Russinovich M. E., Solomon D.A., **Microsoft Windows Internals**, Fourth Edition, Microsoft Press, 2005.
12. Skoudis E., Zeltser L., **Malware: Fighting Malicious Code**, Prentice-Hall, 2003.
13. Tokhtabayev A., Skormin V., Dolgikh A., Beisenbi M. and Kargaldayeva M., "Detection of Specific Semantic Functionalities, such as Self-Replication Mechanism, in Malware Using Colored Petri Nets", *In Proc. SAM'09*, Las Vegas, NV, July 2009
14. Tokhtabayev A., Skormin V. and Dolgikh A. "Expressive, Efficient and Obfuscation Resilient Behavior Based IDS" *15th European Symposium on Research in Computer Security (ESORICS 2010)*, September, 2010 Athens, Greece.
15. Zou C. C., Gong W., Towsley D.: "Code Red Worm Propagation Modeling and Analysis", *In 9th ACM Conference on Computer and Communication Security*, Washington DC, 2002.
16. Miller Ty "Reverse DNS Tunneling Shellcode" presented at Black Hat USA, Las Vegas, 2008
17. Russinovich M. E., Solomon, D.A. "**Windows Internals: Including Windows Server 2008 and Windows Vista**", Fifth Edition, Microsoft Press, 2009.
18. Alexander Sotirov, "Bypassing Memory Protections: The Future of Exploitation", *USENIX Security*, August 2009, Montreal
19. "*Symantec Global Internet Security Threat Report trends for 2009*", Volume XV, Symantec Corporation, April 2010
20. Sutton Michael, "Client Side Attacks Come of Age", ACSAC 2007
21. Chien Eric, "Techniques of Adware and Spyware", *White paper: Symantec Security Response, in proceedings of VB2005*, Dublin, Ireland, 2005
22. Porras, P., Saidi, H., Yegneswaran, V. "*Conficker C analysis*". Technical report, SRI International 2009

23. Hoglund G. and Butler J., “**Subverting the Windows Kernel – Rootkits**”, Addison Wesley, 2006
24. "*W32.Stuxnet Dossier*", Symantec Corporation, October 2010
25. Matrosov Aleksandr, Rodionov Eugene, Harley David and Malcho Juraj "Stuxnet under the microscope", ESET LLC, September 2010
26. Parampalli C., Sekar R. and Johnson R. “A practical mimicry attack against powerful system-call monitors” *In Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS ’08)*, 2008
27. Kumar S. and Spafford E. H. “A Pattern Matching Model for Misuse Intrusion Detection Approach”. *In Proc. of the 17th National Computer Security Conference*, 1994
28. Helmer Guy, Wong Johnny, Slagell Mark, Honavar Vasant, Miller Les, Wang Yanxin, Wang Xia, Stakhanova Natalia “A Software Fault Tree and Colored Petri Nets based specification, Design and Implementation of Agent Based Intrusion Detection Systems”. *International Journal of Information and Computer Security*, **Vol. 1**, no 1/2, pp. 109-142, 2007
29. Ho Y., Frincke D. and Tobin D., “Planning, Petri Nets, and Intrusion Detection” *In Proceedings of the 21st National Information Systems Security Conference*, Crystal City, Virginia, October 1998.
30. Eckmann S., Vigna G. and Kemmerer R. “STATL: an Attack Language for State-based Intrusion Detection”, *in Proc. of the ACM Workshop on Intrusion Detection*, Athens, Greece, November 2000.
31. Cuppens F., Ortalo R. “LAMBDA: A Language to Model a Database for Detection of Attacks, *in Proc. Third International Workshop on Recent Advances in Intrusion Detection*, October 02-04, 2000
32. Michel, Cedric; Me, Ludovic “ADeLe: An Attack Description Language for Knowledge-based Intrusion Detection”, *in Proc. International Conference on Information Security*, Kluwer, June 2001.
33. Pouzol, Jean-Philippe ; Ducassé, Mireille “From Declarative Signatures to Misuse IDS”, *In Proc. Fourth International Symposium on Recent Advances in Intrusion Detection*, LNCS 2212, Springer, 2001, pp. 1-21.
34. Ning, P.; Jajodia, S.; Wang, X. S. “Abstraction-Based Intrusion Detection In Distributed Environments”. *In ACM Transactions on Information and System Security*, Vol. 4, No. 4, November 2001, pp. 407-452.
35. Meier Michael; Bischof Niels; Holz Thomas “SHEDEL - A Simple Hierarchical Event Description Language for Specifying Attack Signatures”. In: *Proc. 17th International Conference on Information Security*. Kluwer, 2002, pp. 559-571.
36. Bernaschi M., Grabrielli E., Mancini L. "Operating System Enhancements to Prevent the Misuse of System Calls", *in Proc. ACM Conference on Computer and Communications Security*, pp. 174 – 183, 2000.
37. Kang D., Fuller D., and Honavar V. “Learning classifiers for misuse and anomaly detection using a bag of system calls representation”. *in Proc. 6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW)*, pp. 118-125, 2005.
38. ThreatFire. http://www.threatfire.com/, accessed 2009
39. Bayer Ulrich at al., “Dynamic analysis of malicious code”, *Journal of Computer Virology*, vol. 2, no. 1, pp. 67-77, 2006.

40. Christodorescu M., Jha S. and Kruegel C., "Mining specications of malicious behavior", *In Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2007.
41. Martignoni Lorenzo at al., "A Layered Architecture for Detecting Malicious Behaviors", *In Proc. 11th International Symposium On Recent Advances in Intrusion Detection (RAID'08)*, Boston, MA, Sep. 2008
42. United States Patent 6973577 B1 "*System and Method for Dynamically Detecting Computer Viruses Through Associative Behavioral Analysis of Runtime State*", Victor Kouznetsov, Dec 6, 2005
43. Cavallaro L., Saxena P. and Sekar R. "On the Limits of Information Flow Techniques for Malware Analysis and Containment" *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*, 2008
44. Visual Paradigm for UML http://www.visual-paradigm.com, accessed 2009.
45. Linz Peter "**An Introduction to Formal Language and Automata**", Fourth Edition, Jones & Bartlett Pub, 2006.
46. Jones N. D. at al.. "Complexity of Some Problems in Petri Nets", *Theoretical Computer Science,* 4:277–299, 1977.
47. Jensen Kurt **Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use**, **volume 1**, Springer-Verlag, Berlin, 1996
48. Newsome J. and Song D. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software" *In Proc. 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.
49. Egele M., Kruegel C., Kirda E., Yin H., and Song D. "Dynamic spyware analysis" *In Proc. USENIX Annual Technical Conference*, June 2007.
50. Volpano D. M. "Safety versus secrecy", *In SAS*, 1999.
51. Bayer Ulrich, Comparetti Paolo Milani, Hlauschek Clemens, Kruegel Christopher, and Kirda Engin "Scalable, Behavior-Based Malware Clustering" *in Proc. NDSS,* 2009
52. Moser A., Kruegel C., and Kirda E.. "Exploring multiple execution paths for malware analysis", *In proc. IEEE Security and Privacy*, 2007.
53. UML, http://www.uml.org/, accessed 2010
54. TestComplete, http://www.automatedqa.com/, accessed 2010
55. PCMark'05, http://www.futuremark.com/, accessed 2010

APPENDIX A – AD FORMALIZATION

Firstly, let us introduce some basic notations we use in the formalization:

O - set of OS objects.

M - set of object manipulations (operations).

Object or manipulations have attributes

$\mathrm{AttList}:\mathrm{O}\cup\mathrm{M}\rightarrow\mathrm{U}$, such that

$\forall x\in\mathrm{O}\ :\mathrm{AttList}(x)$ - set of attributes of the object x.

$\forall y\in\mathrm{M}\ :\mathrm{AttList}(y)$ - set of parameters of manipulation y.

$\mathrm{AttSpace}:\mathrm{O}\times\mathrm{AttList}(\mathrm{O})\cup\mathrm{M}\times\mathrm{AttList}(\mathrm{M})\rightarrow\mathrm{U}$[9], such that

$\forall x\in\mathrm{O},\forall att\in\mathrm{AttList}(x)\ :\ \mathrm{AttSpace}(x,att)$ - set of all possible values (space) of an attribute *att* of an object x.

$\forall y\in\mathrm{M},\forall att\in\mathrm{AttList}(y):\ \mathrm{AttSpace}(y,att)$ - set of all possible values (space) of a parameter *att* of a manipulation y.

The set of objects O includes both subsystem level objects (e.g. “File mapping”, “Socket”) and Kernel objects exported to the user mode (e.g. “File”, ”Process”, ”Find file”). The set of object manipulations M is induced by API functions as well as system calls performing the manipulations. The object manipulation parameter set is generated by attributes of semantically equivalent API functions that export the particular manipulation. The function AttList(*x*) returns list (set) of parameters of the operation *x*.

Based on the above terms, functionality is defined as an Activity Diagram (AD) in the following form:

$$\mathtt{F}=(\mathtt{Nodes},\mathtt{Arcs},\mathtt{Assign},\mathtt{Vars}) \qquad \text{(A-1)}$$

where,

$\mathtt{Vars}$- a set of local variables used in the object manipulations.

$\mathtt{Nodes}=[\mathtt{Ind}\times\mathtt{State}]\cup\mathtt{Pseudo}\cup\{\mathtt{initial},\mathtt{final}\}$ is a set of AD nodes such that,

$\mathtt{State}=\mathtt{Instances}\cup\mathtt{Manipulations}$ is a set of *State* nodes, where

$\mathtt{Instances}$ is a multi-set of object instances defined as:

$$\mathtt{Instances}=\{(Ob,Attr)\,|\,[Ob\in O]\}, \qquad \text{(A-2)}$$

where

$$Attr=\left\{(Name_i,Value_i)\left|\begin{array}{l}[i\in 1..k],[Name_i\in\mathrm{AttList}(Ob)]\\ \left[Value_i\subset\{\mathrm{AttSpace}(Ob,Name_i)\cup\mathtt{Vars}\}\vee Value_i=\mathrm{T}(x),x\in\mathtt{Vars}\right]\end{array}\right.\right\} \qquad \text{(A-3)}$$

where k is the number of critical attributes, *T*() abstract transformation of the input variable.

---

[9] U – universal set (set of all sets)

In the set Instances each element represents a particular object instance which is created in the context of the functionality execution. An object instance consists of the object name (*Ob*) and a set of attributes (*Attr*). Each object *i*-th attribute $Attr_i$ is represented by a tuple $(Name_i, Value_i)$. The first element of the tuple represents the name of the attribute that is unique for a particular object. The second element could define the following: value set from attribute domain, local variable or transformation of the local variable. Transformation *T*() is utilized for specifying informational dependency (flow) between attributes of the operations. Such transformation should not be defined to specify any information flow, e.g. data dependency of any nature including control related flows.

The variables are assigned during functionality execution. The set of attributes contains only those attributes that are critical for functionality execution. For example, in the functionality presented in Table 1, the instance of the "Process" object (created by CreateProcess) can be specified as:

$$\left(Process, \left\{ \begin{matrix} (\text{bInheritHandles, TRUE}), (\text{STARTUPINFO.dwFlags, STARTF_USESTDHAND}) \\ (STARTUPINFO.hStdInput, s), (STARTUPINFO.hStdOutput, s) \end{matrix} \right\} \right) \quad \text{(A-4)}$$

Manipulations - the set of invoked manipulations that is defined as:

$$\texttt{Manipulations} = \left\{ (M, \text{Params}) \middle| [M \in \text{M}] \right\} \quad \text{(A-5)}$$

where

$$\text{Params} = \left\{ (Name_i, Value_i) \middle| \begin{matrix} [i \in 1..k], [Name_i \in \text{AttList}(M)], \\ \left[Value_i \subset \left\{ \text{AttSpace}(M, Name_i) \cup \texttt{Vars} \right\} \vee b_i = \text{T}(x), x \in \texttt{Vars} \right] \end{matrix} \right\}, \quad \text{(A-6)}$$

where k is the number of critical parameters, T() abstract transformation of the input variable.

In the set Manipulations each element represents an object manipulation invoked by the functionality. A manipulation is defined by the operation name and the set of input parameters. Every parameter is represented by a parameter name and a parameter value set that is a subset of the corresponding parameter set or that could be specified as a local variable or its transformation. The set of parameters comprises only those critical parameters that determine functionality.

Ind – a set of process identities, such that each element of this set represents a local ID of the process that performs the object operation. Hence, every distinct process involved in the functionality has its unique index form the set Ind. This addresses the third requirement of the specification allowing for specifying an inter-process functionality.

Pseudo - pseudo nodes that route the control flow, presented by: decision, merge, fork or join.

$$\texttt{Pseudo} = \left\{ x \middle| Type(x) \in \{decision, merge, fork, join\} \right\}, \quad \text{(A-7)}$$

where $Type(x)$ is the type of the node x.

$\texttt{Arcs} = \texttt{ControlFlow} \cup \texttt{HandleFlow}$ is a set of directed arcs connecting operation nodes as a union of mutually exclusive sets ControlFlow and HadnleFlow.

$\texttt{HandleFlow} \subset \text{Nodes} \times \text{Nodes}$ is the set of arcs (handle arcs) that correspond to execution flow with handle inheritance. A handle arc indicates that the destination operation (node) utilizes the same object instance handle as a source operation and is executed right after the source. In other words, the source and destination operations are performed on the same object instance and are involved in the same manipulation session. In terms of the UML 2.x activity diagrams syntax [14], such arcs could be viewed as a fusion of the object flow with the control flow.

$\texttt{ControlFlow} \subset \text{Nodes} \times \text{Nodes}$ is a set of directed arcs that define the control flow without handle inheritance. The arc from this set indicates that the destination operation is executed right after the source operation. Note, such arc simply shows the execution order and does not indicate any data binding (via handle or attribute).

$\texttt{Assign} : \text{Arcs} \rightarrow \text{Expression} \cup \varnothing$ is a variable assignment and guard function such that,

$$\texttt{Assign}(a) = \begin{cases} \text{Assignment expression, } Source(a) \in \text{State} \\ \text{Guard expression, } Source(a) \in \text{Pseudo} \end{cases}, \ \forall a \in \text{Arcs} : Source(a) \qquad \text{(A-8)}$$

$$\text{Assignment expression} = \left\{ "v := out" \middle| v \in \text{Vars}, out \in OutPar\left(Source(a)\right) \right\}, \qquad \text{(A-9)}$$

where $OutPar(x)$ is a list of output parameters of object operation x.

This function defines a variable assignment expression for corresponding arcs having the *State* node as a source. The assignment expression utilizes output parameters of the arc's source operations to assign required local variables. Such parameters may include object descriptors (handle, memory offset, etc) of the source operation. If the source of the arc is a Pseudo state node, this function determines a guard expression as defined in the original UML 2.0 activity diagrams. Note that the Assign function does not define an expression for every arc, but for those where it is necessary.

APPENDIX B - REMOTE IPC ADS

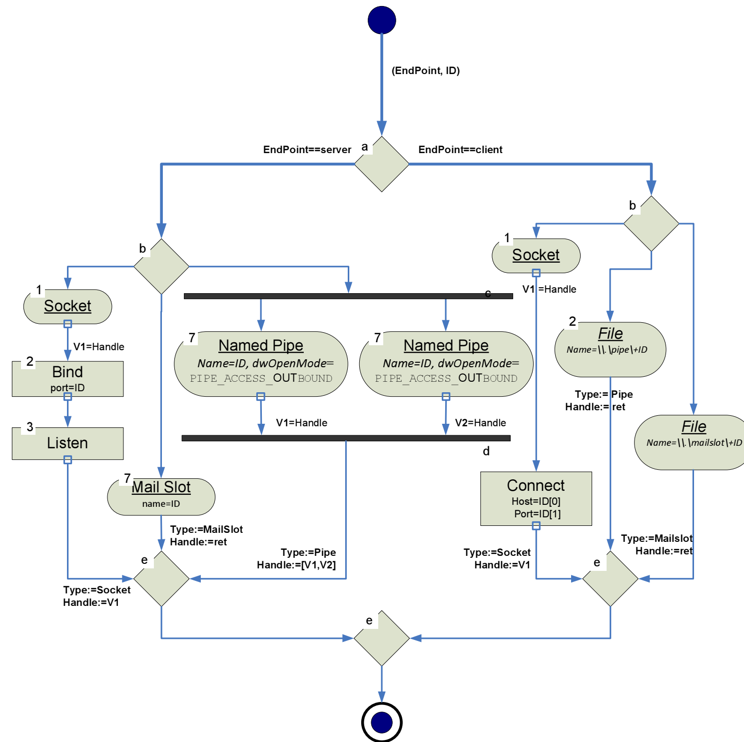

Figure B-1. Remote IPC- Create Operation

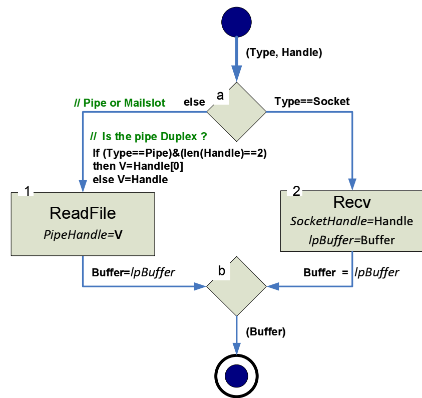

Figure B-2. Remote IPC – Receive Operation

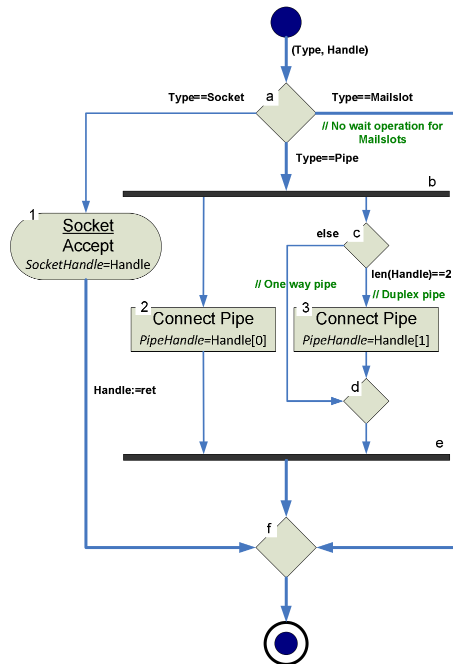

Figure B-3. Remote IPC –Wait Operation

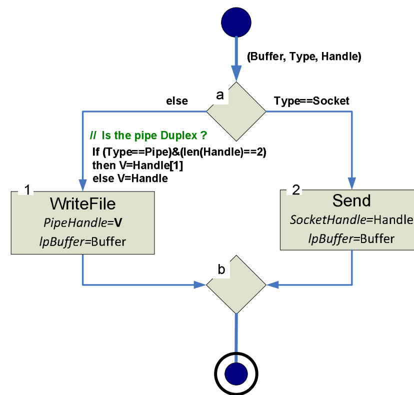

Figure B-4. Remote IPC – Send Operation

APPENDIX C - GENERALIZATION FUNCTIONALITIES AD

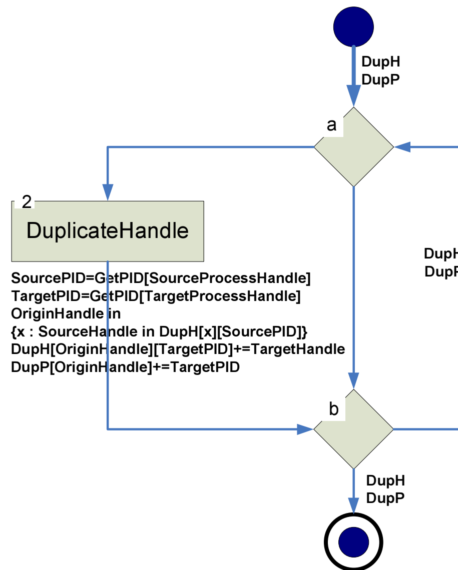

Figure C-1. Handle Duplication Functionality

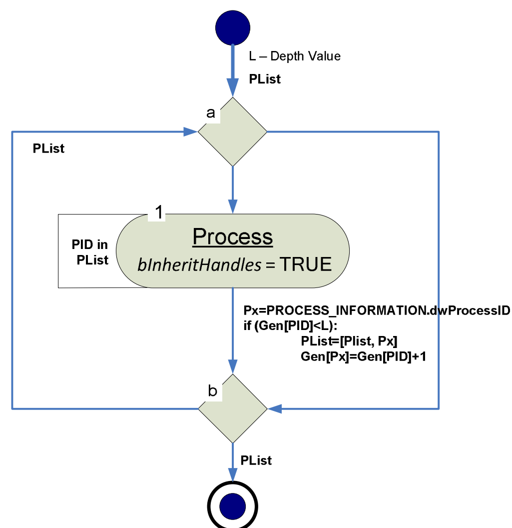

Figure C-2. Process Generation Functionality

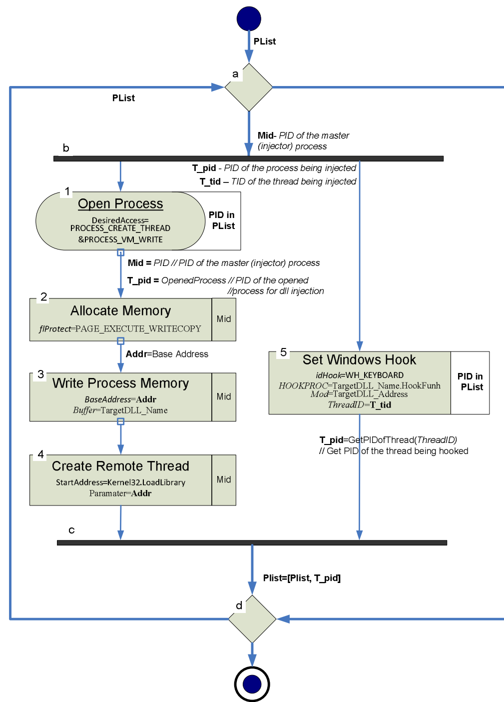

Figure C-3. Code Injection Functionality

APPENDIX D - FUNCTIONS UTILIZED IN GENERALIZATION ALGORITHMS

```
AddParallelFunct(F.AD OriginAD, F.AD NewAD, OriginAD.Nodes Fork,
OriginAD.Nodes Join)
```

It adds AD of NewAD functionality to AD of OriginAD functionality as a parallel flow that starts right after the node Fork and joins to Origin AD just before the node Join. F.AD means set of AD of all functionalities.

```
NewNode=AddParallelNode(F.AD OriginAD, O∪M NewOperation, OriginAD.Nodes
Fork, OriginAD.Nodes Join)
```

Creates node representing an input operation (NewOperation) and adds it to OriginAD functionality as a parallel flow that starts right after the node Fork and joins to Origin AD just before the node Join. This function returns added node NewNode. $O \cup M$ is a set of objects and manipulations.

```
NewNode=AddNextNode(F.AD OriginAD, O∪M NewOperation, OriginAD.Nodes
ParentNode)
```

Creates node representing an input operation (NewOperation) and adds it to OriginAD functionality right after the node ParentNode.

```
AttValue=GetAttributeValue(AD.Node.State Node, AttList(Node) Attr)
```

Returns value of the attribute Attr of the node Node.

```
SetAttributeValueExpression(AD.Node.State Node, AttList(Node) Attr, String
Expression)
```

Sets attribute expression for Attr attribute of the state node Node of the current AD.

```
SetNodePIDExpression(AD.Node.State Node, String Expression)
```

Sets an expression assigning PID of the node Node.

```
NewVarName=CreateNewVar(InputAD.Arcs Arc, String Expression)
```

Introduces a new variable to the current input AD arc (Arc) that is defined with the assignment expression (Expression). The assignment expression may use output attributes of the parent of the arc and other global variables. This function returns name of the newly created variable. By current AD we mean AD being input of the algorithm.

```
InputAD.Nodes.State Node=GetAssignNode(InputAD.Vars Var)
```

Searches for and returns the node in current AD which output arc assigns variable Var.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ActiveX – framework for defining reusable software
AD – Activity Diagram
ASLR – Address Space Layout Randomization
BBIDS – Behavior Based Intrusion Detection System
CPN – Colored Petri Net
CPU – Central Processing Unit or Processor
C&C – Command and Control (botnet C&C)
DEP – Data Execution Prevention
DNS – Domain Name System
FTP – File Transfer Protocol
GSR – Gene of Self Replication
ICMP – Internet Control Message Protocol
ICQ – Messaging protocol
IDS – Intrusion Detection System
IRC – Internet Relay Chat
MIME – Multipurpose Internet Mail Extensions
MS – Microsoft Corporation
NIDS – Network based Intrusion Detection System
OS – Operating System
PLC – Programmable Logic Controller
TCP – Transmission Control Protocol