On the Generation of Cyber Threat Intelligence: Malware and Network Traffic Analyses

Amine Boukhtouta

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

April 2016

CONCORDIA UNIVERSITY

Devision Of Graduate Studies

This is to certify that the thesis prepared

By: **Amine Boukhtouta**

Entitled: **On the Generation of Cyber Threat Intelligence: Malware and Network Traffic Analyses**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Fariborz Haghighat	Chair
Dr. Indrajit Ray	External Examiner
Dr. Olga Ormandjieva	External to Program
Dr. Ferhat Khendek	Examiner
Dr. Amr Youssef	Examiner
Dr. Mourad Debbabi	Thesis Supervisor

Approved by ______________________________

Chair of Department or Graduate Program Director

______________ ______________________

Dean of Faculty

ABSTRACT

On the Generation of Cyber Threat Intelligence: Malware and Network Traffic Analyses

Amine Boukhtouta, Ph. D.

Concordia University, 2016

In recent years, malware authors drastically changed their course on the subject of threat design and implementation. Malware authors, namely, hackers or cyber-terrorists perpetrate new forms of cyber-crimes involving more innovative hacking techniques. Being motivated by financial or political reasons, attackers target computer systems ranging from personal computers to organizations' networks to collect and steal sensitive data as well as blackmail, scam people, or scupper IT infrastructures. Accordingly, IT security experts face new challenges, as they need to counter cyber-threats proactively. The challenge takes a continuous allure of a fight, where cyber-criminals are obsessed by the idea of outsmarting security defenses. As such, security experts have to elaborate an effective strategy to counter cyber-criminals. The generation of cyber-threat intelligence is of a paramount importance as stated in the following quote: "the field is owned by who owns the intelligence". In this thesis, we address the problem of generating timely and relevant cyber-threat intelligence for the purpose of detection, prevention and mitigation of cyber-attacks. To do so, we initiate a research effort, which falls into: First, we analyze prominent cyber-crime toolkits to grasp the inner-secrets and workings of advanced threats. We dissect prominent malware like Zeus and Mariposa botnets to uncover their underlying techniques used to build a networked army of infected machines. Second, we investigate cyber-crime infrastructures, where we elaborate on the generation of a cyber-threat intelligence for situational awareness. We adapt a graph-theoretic approach to study infrastructures used by malware to perpetrate malicious activities. We build a scoring mechanism based on a page ranking algorithm to measure the badness of infrastructures' elements, i.e., domains, IPs, domain owners, etc. In addition, we use the min-hashing technique to evaluate the level of sharing among cyber-threat infrastructures during a period of one year. Third, we use machine learning techniques to fingerprint malicious IP traffic. By fingerprinting, we mean detecting malicious network flows and their attribution to malware families. This research effort relies on a ground truth collected from the dynamic analysis of malware samples. Finally, we investigate the generation of cyber-threat intelligence from passive DNS streams. To this end, we design and implement a system that generates anomalies from passive DNS traffic. Due to the tremendous nature of DNS data, we build a system on top of a cluster computing framework, namely, Apache Spark [70]. The integrated analytic system has the ability to detect anomalies observed in DNS records, which are potentially generated by widespread cyber-threats.

ACKNOWLEDGEMENTS

To a great degree, all praises to God for giving me the power to accomplish this thesis. I could not finish this work without his blessing and guidance. I would like to express my heartfelt gratitude to my supervisor Prof. Mourad Debbabi, who has highly contributed in the supervision and fulfillment of this thesis.

I have to express my appreciation to the committee members, including Dr. Amr Youssef, Dr. Olga Ormandjieva, Dr. Ferhat Khendek, and Dr. Indrajit Ray for evaluating my thesis and providing me with valuable feedback. I thank the National Cyber-Forensics & Training Alliance (NCFTA) Canada for providing facilities to conduct research, this work would not be possible without their active supports. I thank Farsight Security, Inc. and in particular, Dr. Paul Vixie, for the access to valuable data feeds.

Special thanks to lab-mates, namely, Hamad Binsalleeh, Nour-Eddine Lakhdari, Serguei Mokhov, Djedjiga Mouheb, Farkhund Iqbal, Son Dinh, Claude Fachkha, Elias Bou-Harb, Prosenjit Sinha, Thomas Ormerod and Victor Belarde, Taher Azab, Saed Alrabaee, Perry Jones, Houssam Borjiba, Mouatez Karbab, Anhar Haneef, Paria Shirani, Lina Nouh, Ray Sujoy, André Soeanu for stimulating discussions and good memories shared in the laboratory. Last but not least, I would like to express my profound gratitude to my beloved wife and daughter for their moral support and patience during my studies, without forgetting my parents, sister, brothers, my uncle and his family, and my best friend H. Oualikene for their encouragement and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

A	Internet Protocol v4 Address Record
AAAA	Internet Protocol v6 Address Record
AIS	Artificial Immune Systems
ANY	DNS name-servers reconnaissance record
API	Application Programming Interface
ARIMA	Auto-regressive Integrated Moving Average
ASN	Autonomous System Number
C&C	Commmand and Control
CDN	Content Delivery Network
CNAME	Canonical Name DNS Record
CPU	Central Processing Unit
DCI	Direct Code Injection
DGA	Domain Generation Algorithm
DKIM	Domain Keys Identified Mail
DMARC	Domain-based Message Authentication, Reporting & Conformance
DNS	Domain Name System
DoS	Denial of Service
DDoS	Distributed Denial of Service

DRDoS	Distributed Reflection Denial of Service
DPI	Deep Packet Inspection
EM	Expectation Maximization
EP	Entry Point
FFT	Fast Fourier Transform
FTP	File Transfer Protocol
HDFS	HaDoop File System
HMM	Hidden Markov Model
HTTP	Hyper-Text Transfer Protocol
IDS	Intrusion Detection System
IRC	Internet Relay Chat
ISP	Internet Service Provider
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LDAP	Lightweight Directory Access Protocol
LPC	Linear Prediction Coefficient
MAC	Modification, Access, & Creation
NAT	Network Address Translation
NCFTA	National Cyber-Forensics Training Alliance

NetBIOS	Network Basic Input/Output System
NLP	Natural language Processing
NULL	Experimental null DNS record
OPT	Pseudo DNS record type
PCM	Pulse-Code Modulation
PHP	Personal Home Page script language
PPM	Prediction per Partial Matching Algorithm
RAM	Random Access Memory
RC4	Rivest Cipher 4
RDD	Resilient Distributed Disk
SLD	Second-Level Domain
SMTP	Simple Mail Transfer Protocol
SPF	Sender Policy Framework
SSH	Secure SHell
SIP	Session Initiation Protocol
SQL	Structured Query Language
SRV	Service Locator DNS record
SVM	Support Vector Machine
TLD	Top-Level Domain
TTL	Time-To-Live

TCP	Transmission Control Protocol
TXT	Text DNS record
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VoMM	Variable order Markov Model
XML	eXtensible Markup Language
XOR	eXclusive-OR
XSS	Cross-Site Scripting Attack

Chapter 1

Introduction

1.1 Motivation and Problem Description

With the stupendous expansion of information technology, individuals, corporations and institutions rely mainly on information systems and networks (Internet) to send, receive and store security critical data. Fields like communication, finance, business, research and development use such information system networks. However, these networks face the emergence of innovative cyber-threats and attacks. They represent nests for cyber-crime activities. Cyber-criminals have been showing a keen interest to create cyber-threats and orchestrate featured cyber-attacks such as information theft, email-spams, malware infections, networks of malicious malware robots (botnets), Distributed Denial of Service (DDoS) attacks, etc. Cyber-threats and attacks have the following attributes:

- *Originality*: Cyber-criminals have created sophisticated networks of infected machines that use existing service or protocols (e.g., Domain Name Service (DNS), Internet Relay Chat (IRC) or Peer-to-Peer (P2P)) to steal sensitive information such as users' credentials, credit card numbers and email addresses.

- *Financial motivation*: Observed threats and attacks result in severe lost. For example, FBI reported that ten cyber-criminals managing botnets have stolen $\$850$ million after obtaining personal financial information from infected machines [152].

- *Opportunism*: Cyber-criminals tend to take advantage of existing software and systems vulnerabilities to cause harm in networks. For example, after the disclosure of security bugs in Unix Bash shell, known as Shellshock [166] on 24^{th} September 2014, hackers targeted many web server deployments to execute Bash arbitrary commands.

- *Service unavailability*: Cyber-criminals tend to take down services. For example, in 2013, a well-established IT security organism was a victim of a Distributed Reflection Denial of Service (DRDoS) attack [49]. Cyber-attackers took advantage of DNS protocol to generate a stream of 300 Gbps of data. They redirect such data traffic to spoofed IP addresses belonging to the organization.

- *Sabotage of critical infrastructures*: Hackers design and integrate cyber-weapons to target critical infrastructures. In 2010, a worm known as Stuxnet [237], targeting Programmable Logic Controllers (PLCs) infected Iranian networks. It was designed to sabotage Iranian nuclear program.

In the prevailing of the illustrated cyber-threat and attack attributes, security experts have to design appropriate techniques to extract operational cyber-threat intelligence from malware and network traffic sources. Such intelligence is employed to detect, prevent and mitigate different threats. In this thesis, we focus on the generation of intelligence based on malware feeds, malicious traffic and passive DNS logs. We leverage different classification techniques, graph theory algorithms and patterns identification to generate an intelligence out of data provided by third parties, e.g., ThreatTrack [210]. This thesis tackles four threads of research, which are described in the sequel.

1.2 Objectives

The primary intent of this thesis is to generate relevant and timely cyber threat intelligence for detection, prevention and attribution purposes. We envision to accomplish this through the analysis of malware samples and network traffic. More explicitly, the main objectives of this thesis are to:

- Grasp the inner-workings of cyber threats through the reverse engineering of prominent malware samples
- Analyze cyber threats and the underlying infrastructures together with an assessment of their badness and patterns

- Elaborate, design and implement a technique for the automatic fingerprinting on cyber threats in IP traffic
- Elaborate, design and implement a scalable detection system, which identifies anomalies in passive DNS streams

1.3 Methodology

In order to fulfill the aforementioned objectives, we define two types of cyber-threat intelligence, namely, malware based cyber-threat intelligence and network based cyber-threat intelligence. The former aims at the extraction of intelligence based on malware analysis, whereas the latter targets to harvest intelligence to corroborate the detection and prevention of threats at the network level.

1.3.1 Malware Cyber-Threat Intelligence

To extract such intelligence, we aim at gaining a deep knowledge about innovative threats to understand their modus-operandi as well as unveiling their cyber infrastructures, i.e., domains, IPs, etc. To do so, we use static and dynamic malware analyses to generate intelligence needed by security researchers to detect, prevent and mitigate advanced threats as well as to draw a situational awareness of different infrastructures used to perpetrate attacks.

1.3.2 Network Traffic Cyber-Threat Intelligence

In addition to the former intelligence, we target to act on the network level, where we use network traces collected from the malware dynamic malware analysis and Domain Name System (DNS) logs to detect the maliciousness and indicators of compromise. To do so, we consider the use of machine learning techniques to segregate the malicious IP layer traffic from the benign one and attribute it to threats as well as to look at DNS streams to detect anomalies generated by threats at the application network layer. In the sequel, we provide an overview of each contribution discussed in this dissertation.

1.4 Contributions

1.4.1 Analysis of Prominent Threats

Static and dynamic malware analyses are considered as cornerstone artifacts that boost the learning curve about cyber-criminals underground communities. Valuable information can be gathered by analyzing malicious binaries. The aim of this research effort is to answer the following question: (1) How can we grasp cyber-threats inner-workings? A reply lies in reverse-engineering harmful software, uncovering their dynamics, namely, code obfuscation, infection methods and communication schemes. Such analysis has its own unique value for IT security since it allows identification of malware attack vectors. Therefore, an intelligence can be used to extract patterns or signatures that are useful for the detection of partially or totally shared malicious behaviors.

We analyze two media noisy botnets by reverse engineering techniques. We present a detailed reverse engineering analysis of Zeus and Mariposa crime-ware toolkits to unveil techniques use by malware creators to perpetrate malicious activities. We report in these two reverse engineering many observations that are insightful in terms of de-obfuscation of packed malware as well as communication schemes used by advanced botnets.

1.4.2 Investigation of Cyber-Threat Infrastructures

Nowadays, cyber-criminals use network resources to conduct their malicious activities. They set up networks of compromised machines to perpetrate attacks on both corporations and individuals. Infected machines are instructed to steal sensitive data, conduct reconnaissance, launch DDoS attacks, etc. As such, there is a keen interest to investigate infrastructures used by cyber-criminals. Being inspired by a first effort done by Nadji *et al.* [147], we conduct a research initiative to look thoroughly at cyber-threat infrastructures. In this research effort, we target to answer the following questions: (1) How to characterize infrastructures used by cyber-threats? (2) What are the key players in cyber-threats infrastructures? (3) What are the shared elements between cyber-threats among such infrastructures? To tackle the aforesaid questions, we use a graph-theoretic approach to characterize elements observed in cyber-threat infrastructures. We use influence concept to rank badness of elements constituting cyber-threat infrastructures. Finally, we employ a graph hashing technique to identify patterns shared between different cyber-threat infrastructures.

We apply a graph-theoretic approach to characterize infrastructures used by malware. Based on one-year data collected from dynamic malware daily feeds, we characterize cyber-threat infrastructures as graphs. We use Google's PageRank algorithm [42] to rank badness of key players, i.e., IP addresses, domains, owners, registrars. In addition, we employ min-hashing algorithm [208] to identify recurrent patterns appearing in cyber-threat infrastructures.

1.4.3 Fingerprinting Maliciousness in IP Traffic

Network defense relies on cyber-attacks detection, prevention, analysis, mitigation and attribution. Cyber-criminals leverage malware to perpetrate amplified, large-scale, debilitating, intimidating and disrupting attacks causing severe privacy/economic consequences. The infected machines send or receive suspicious network flows, which can be different compromise indicators like worm propagation, botnet, commands, probing events, DDoS, etc. In such cases, security analysts would like to detect and mitigate such activities. Thus, there is a desideratum to develop maliciousness fingerprinting techniques at the network level. By fingerprinting, we mean the ability of malicious traffic detection, then, its malware family attribution. Thus, we define a research that attempts to identify maliciousness in IP traffic.

In this research effort, we target to answer the following questions: (1) How to use malware analysis downstream outcome to fingerprint maliciousness at the network level? (2) What are the techniques to detect malicious traffic and attribute it to malware? (3) Among potential techniques, is there a technique that is better than others or are they complementary? The goal of this research is to use malicious traffic collected from dynamic malware analysis as an intelligence (ground truth) to classify malicious traffic. To do so, we choose two techniques, packet headers and Deep Packet Inspection (DPI) [56, 111] malicious traffic classification and malware families attribution. These techniques lie in applying machine learning techniques, which are widely used to identify patterns, i.e., maliciousness patterns. Both techniques are compared based on their detection and attribution accuracies as well as their level of complexity.

1.4.4 Near-Real-Time and Scalable Detection of Anomalies in Passive DNS Streams

A part of Internet evolution, DNS protocol plays the phone-book role. It is a masterpiece that allows hosts accessible worldwide through the internet. Despite its benign utility, DNS carries out malicious activities. Hackers abuse its flexibility to create short-lived domains used as botnets control nodes. Harmful programs (e.g., Torbig [202]) employ Domain Generation Algorithms (DGA) [50] to register domain names resolving bot-masters. DNS permits communication between infected machines with queries, perpetrating activities like key-logging, spamming or spreading infections. In addition, C&Cs exploit DNS tunneling for malign payloads distribution. Malware families like Morto [145], Katusha [154] and Feederbot [58] employ this technique to create covert channels for data transport. Accordingly, we investigate passive DNS data. We monitor such real-time stream data for cyber-threats identification.

In this research thread, we focus on answering the following questions: (1) How to monitor DNS traffic cyber-threats misuse? (2) Which artifacts needed to handle real-time streams of DNS data? (3) How to identify potential cyber-threat infrastructures based on DNS protocol misuse? To deal with these research issues, we decide to use a computational clustering solution to capture near real-time data and extract DNS protocol anomalies. We utilize outlier detection algorithm and scoring functions to identify DNS misuses. We design and integrate an online near real-time system to identify DNS anomalies. These anomalies fall into machine generated domains, DNS malware covert channels, fast-flux malicious networks based on DNS-based features. In addition, we corroborate the system with the ability to monitor IP spaces of organizations like universities, governmental and financial organizations. We use a lightning-fast cluster computing framework, namely, Apache Spark [70] to aggregate, map, and reduce DNS logs for the purpose of anomalies identification. The work described in this thesis was published in [32, 38–41, 189].

1.5 Thesis Organization

The remainder of this thesis is structured as follows: In Chapter 2, we present the background literature and related work. In Chapter 3, we describe reverse engineering analysis of the Zeus and Mariposa crime-ware toolkits. Chapter 4 puts forward an investigation on cyber-threat infrastructures. In Chapter 5, we entail the different approaches in the name of header flow-based features classification and, Signal and NLP DPI, to detect maliciousness in IP traffic. Chapter 6 sets forth our passive DNS anomalies identification system along with performance benchmarks. Chapter 7 provides concluding remarks together with a discussion of future works.

Chapter 2

Background and Related Work

2.1 Overview

In this chapter, we provide some definitions related to cyber-threat intelligence. The chapter is organized as follows: Section 2.2 puts forward different concepts related to cyber-threat intelligence, namely, its definition, challenges, model and different sources. Section 2.4 reviews related work tackling the use of graph-theoretic approach for the purpose of characterization and analysis of networks. Section 2.3 introduces different works, where prominent threats have been analyzed. Section 2.5 reviews the different works done on fingerprinting network traffic. Finally, Section 2.6 entails the different works related to DNS monitoring systems.

2.2 Cyber-Threat Intelligence

2.2.1 Definition

Cyber-threat intelligence is meant to be the relevant information and inductive reasoning gathered from tracking, analysis and mitigation of security threats. This intelligence is a mix of physical espionage and information technology [9]. The cyber-threat intelligence efforts target mainly fighting against viruses, hackers and terrorists that consider the Internet as an artifact to perpetrate malicious activities. The protection of governmental institutions, commercial companies and individuals from cyber-threats is the main actor of the cyber-threat intelligence. Cyber-threat intelligence parties combat different forms of threats. Cyber-threat intelligence authorities tend to provide security against threats. Cyber-threat intelligence experts should have a dual background in IT security and espionage. The analysis of terror threats is one of the important aspects in cyber-threat intelligence. It needs the collection of information from third parties in the name of governments, independent companies, ISPs and universities. This data helps to ascertain how threats were perpetrated. It can result in useful reports for future investigations and mitigation.

2.2.2 Cyber-Threat Intelligence Challenges

Nowadays, emerging threats come up with more and targeted attack scenarios. More advanced and prominent malicious programs and activities have been taking place in comparison with threats that appeared ten years ago. Newer attack scenarios follow a kill chain model. Figure 2.1 illustrates this model, which shows the different stages of an attack. This Attack chain model represents the modus-operandi of today's prominent threats. The authors tend to conduct cyber-crimes, industrial espionage, terrorism and hacktivism.

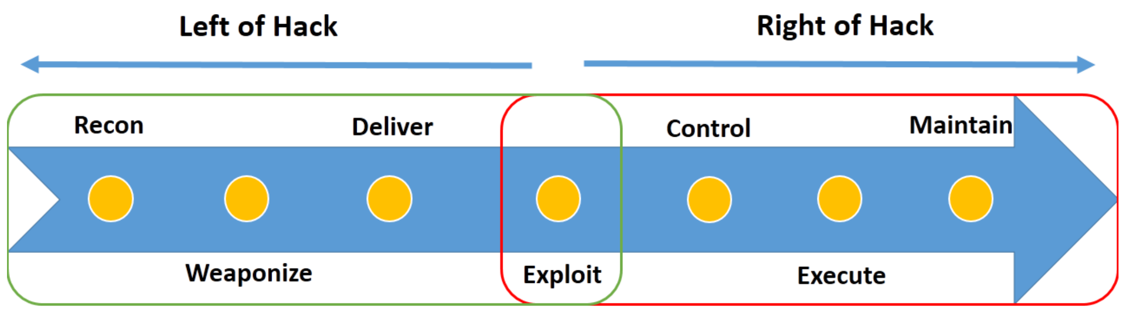


Figure 2.1: Attack Chain Model

New threats driven by hackers aim to persist and cause ongoing damages. This fact motivates the need to move beyond the traditional reactive approach to a more proactive one. In order to be proactive, IT security experts need to change the nature of the defense strategy. The intent is to get left of the hack in order to kill the chain of attack at an early stage. Thus, grasping hackers' capabilities, actions and intent bring a valuable support for security in the cyber-space. The challenges of cyber-threat intelligence are characterized by the following questions:

- How can attacks be detected and recognized?
- How can attacks be mitigated?
- Who is behind perpetuating an attack?

- What are the objectives of attackers?
- What are the tactics, techniques and procedures that are leveraged by attackers?
- What are the vulnerabilities, misconfiguration and weaknesses that are likely targeted by attackers?

2.2.3 Cyber-Threat Intelligence Model

Cyber-threat intelligence is meant to support a set of core use cases involved in cyber-threat management and mitigation. Figure 2.2 illustrates cyber-threat intelligence use cases model. In the sequel, we introduce the different use cases.

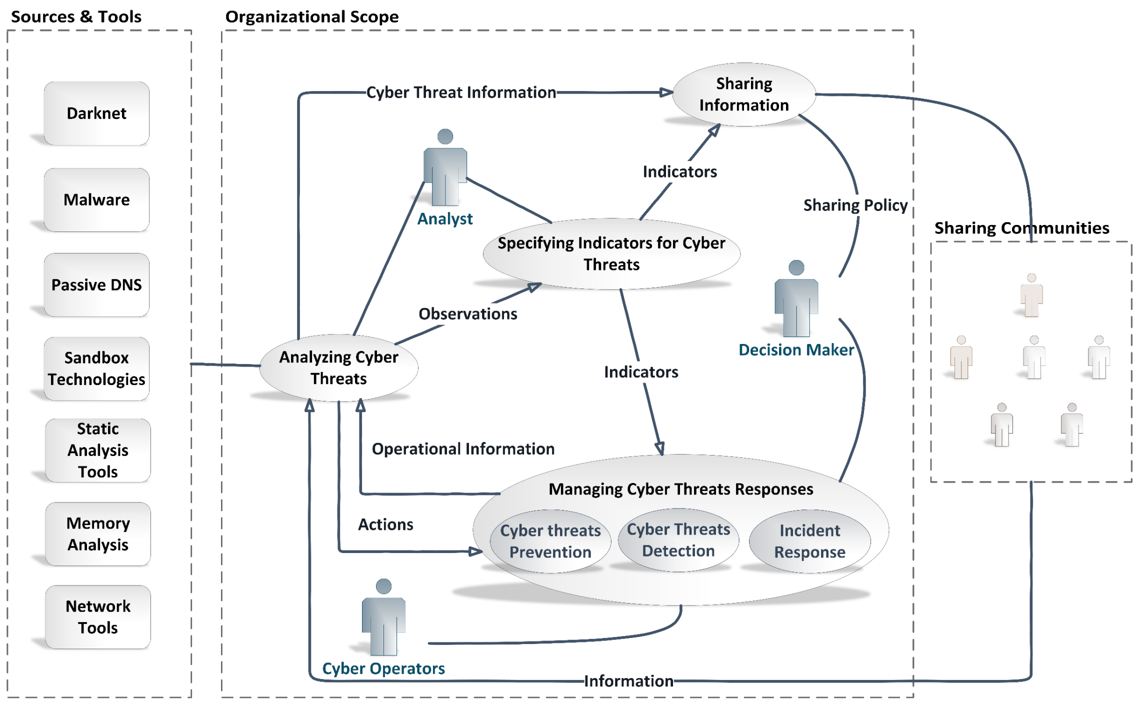


Figure 2.2: Cyber-Threat Intelligence Use Cases Model

Analysis of Cyber-Threats

A security analyst reviews information related to cyber-threat activity from manual or automated sources. The analyst aims to understand the nature of relevant threats, ascertain them and characterize them in order to grasp the inner-secrets of threats, which tend to evolve over time. The knowledge encloses threat behaviors, intents, attribution and capabilities. Thus, the analyst can put forward threat indicators to prevent further threats and suggest courses of actions and mitigation.

Specifying Indicators for Cyber-Threats

An analyst produces patterns representing the observable characteristics of cyber threats. The expert maps the indicators along with threats for the purpose of interpreting, handling and applying patterns to detect them. For example, in the case of a phishing attack, an analyst harvests observables (email addresses, source, subject, embedded URLs, attachments, etc.) from the analysis of the phishing email, identify the relevant tactics, techniques and procedures that are exhibited in the phishing attack. The expert performs a kill chain correlation of the attack by blacklisting emails and post them to sharing communities.

Managing Cyber-Threat Responses

Decision makers and operational personnel aim to prevent or detect cyber-threat activity. In addition, they want to investigate and counter any detected incidents. Preventive courses of action are remedial in nature to mitigate vulnerabilities, which are targeted by exploits. Decision makers and operational personnel work together to understand the effect of attacks in order to assess the cost and efficiency of potential courses of and elaborate an appropriate preventive/detective actions.

Sharing Cyber-Threat Information

Decision makers establish policies in order to share different cyber-threat information. They decide with which other parties the data should be shared and how it should be handled based on agreement frameworks of trust. The sharing policy is implemented to share indicators and cyber-threat information. The relevant information is shared automatically or manually by trusted partners.

Cyber-Threat Intelligence Sources

Malware Analysis A malware is a piece of small software, which completes the harmful intent of cyber-criminals. Terms such as worm, bot, rootkit, spyware are used to categorize malware samples, which mirror common malicious activities. The following paragraph is intended to digest different malware types. In addition, these types are known to not be mutually exclusive. A given malware may be hybrid since it can reflect characteristics of multiple classes. In [191, 205], the authors discussed such malicious hybrid activities. Malware fall into the following categories:

- *Virus*: Spafford defined viruses as follows: "A virus is a piece of code that adds itself to other programs, including operating systems. It cannot run independently and it requires that its host program be run to activate it". Viruses propagate by infecting vulnerable hosts and local files.

- *Worm*: This kind of malware is known in networked systems. In [196], the author gave a worm the following definition "a program that can run independently and can propagate a fully working version of itself to other machines". The Morris Worm [196] is the first worm, which came onto scene. In the last decade, the Code Red worm [143] infected thousands of machines during the first days of its appearance. More recently, the Storm worm was used to create botnets to send spams and perpetrate denial of service attacks [101].

- *Trojan*: It is the software that seems to be legitimate but performs malicious background activities. This software may mirror useful activities, frequently, screensavers, games or browser plug-in objects. It can launch malicious activities once installed in the system. It is used mainly to download other pieces of malware.

- *Spyware*: This malicious software steals sensitive information from a user and sends this information to third parties. The information can be passwords, number of debit and credit cards, emails and visited websites.

- *Bot*: A bot is malware, which is controlled remotely by a bot-master. It uses network protocols such as P2P, HTTP or IRC to communicate with controllers. For instance, we can cite Zeus [32] and Mariposa [189]. These malware appeared mainly in 2009 and 2010, and were behind the perpetuation of malicious activities such as stealing information to bot-masters, which were located in many countries.

- *Rootkit*: It is a malware that has the ability to hide its presence by applying some advanced techniques. These techniques can be applied at different levels such as the instrumentation of API calls or interfering with system structures like kernel modules or drivers. The rootkits are common since they are installed by other malware, specifically bots and spyware.

Before generating a signature for a given malware, security analysts tend to check whether it is a real threat or not. Different techniques permit IT security experts to unveil the risk and intention hidden in potential malware. Resulted insight allows the analyst to find new trends in malware development and mitigate different threats. The main intent of IT security experts is to overlook the behavior of a sample. Since analysis tools and techniques are more elaborated, malware authors move to the integration of new innovative evasion techniques in order to avoid their malware being examined. These techniques fall mainly into self-modifying binary and the detection of the presence of an analysis tool. In the sequel, we provide an overview of static and dynamic malware analysis.

- *Static Malware Analysis*: Static malware analysis tends to dissect malware samples to find out the different functions that are hidden in malware binaries. Static analysis can be applied on the source representation of a program. The static analysis tools are used to harvest relevant information about malware. For instance, a call graph gives an analyst an insight about malware structure and which function may be invoked in the code. In the majority of situations, the source code of malware is not readable. It is dissembled to machine language. By analyzing binaries, an analyst can be confronted to binary self-modifying techniques. In addition, malware based on values that cannot be found such as system date or indirect jump instructions, may make the analysis more difficult for analysts. Moreover, some malware authors may fingerprint static analysis methods, so they can detect instances to prevent running of malware.

- *Dynamic Malware Analysis*: It consists of analyzing a program's actions while it is being run in the system. There are techniques that are related to such analysis and they span over:

  1. *Function Call Monitoring*: A function call relies on the analysis of code that performs actions intended for different tasks. These functions tend to be reusable in different versions of malware. One possible way to analyze the malware behavior consists of intercepting functions. Such method is called hooking. It allows to log function invocations and analyze input/output parameters. The implementation of hooking function has many approaches. It depends on the availability of programs source code. If the code is available, hooks can be inserted into appropriate places. Another technique is to use binary rewriting if a malware is available in a binary form. Binary rewriting falls into two techniques: either rewriting monitored functions to call hook functions instead or modifying all call locations to invoke the hook. The hook function

can access the original arguments on the stack and monitor them. Moreover, if a function is invoked through a DLL function pointer, the value of a pointer can be changed to point to the hook function. Hunt and Brubacher [91] introduced the Detours library to apply function rewriting in order to implement hooking function. Their technique consists of creating a trampoline function that contains overwritten instructions. This function contains an unconditional jump to the original function after overwritten instructions. These instructions perform the code analysis. The code may contain any pre-processing and control the execution flow. The Detours library provides two alternatives to apply modifications to programs. It has the ability to either modify the binary before execution or manipulate the in-memory images of loaded binaries.

2. *Function Call Traces*: This is another technique, which tends to monitor function calls inputs and outputs. The trace lies in the set of functions that were invoked by the program with passed arguments. These traces are used to create abstract representations of malware behavior. In [48], the authors represented call traces with graph representations. Such representations permit them to compare behavior of malicious programs with legitimate software. Thus, analysts can find out malicious instances of the same malware families within unknown samples. In [226], authors use traces of known malware to detect polymorphic variants of unknown samples. The authors applied sequence alignment technique to compute function traces similarities. Such a

technique can take a considerable running time in order to calculate differences and similarities between traces.

Darknet The term darknet emerged at the beginning of this millennium. First, Microsoft researchers defined it as “a collection of networks and technologies used to share digital content” [28]. Later, darknet has been associated with other meanings. Currently, no single definition has been globally accepted. Thus, *darknet*, refers to but not limited to the following definitions:

- *Darknet as Dark Address Space*: Usually refers to routable public IP addresses that are not publicized or advertised to the Internet community. These IP addresses have neither assigned hosts nor DNS entries or search engines’ indexing. Therefore, noticing the online existence of these elements is not simple without prior knowledge. This address space can be used either for malicious activities or for benign traffic monitoring.

- *Darknet as an Anonymity Environment*: An environment that provides communication anonymity. This is related to the task of achieving private communication between users.

- *Darknet as Dark Web*: Also known as *Invisible Web* or *Deep Web*. It refers to digital content, which exists in the public cyberspace. It is known to be untraceable and inaccessible by regular search engines. Such cyberspace content remains concealed because there are neither registration records among domain name servers nor direct link pointing to it.

- *Darknet as Private P2P Communication*: It refers to any type of closed, private, and concealed communication between groups of people. It represents a mixture of cordoned-off encrypted peer-to-peer networks that overlay the existing Internet design. Such Darknets, often consisting of a tight-knit group of people, are conceived based on trust and common interests. Moreover, joining such networks require an invitation from trusted members.

In this dissertation, we refer to Darknets with the dark address space, since these network addresses correspond to illegitimate hosts or devices, any observed traffic originating or targeting the dark address space, is suspicious and hence needs to be investigated. The intent of monitoring Darknet is to look for pandemic and epidemic cyber incidents through the unused (dark) address space. Darknets known also as Network Telescopes are assimilated to astronomical telescopes since large and sensitive telescopes have a high probability to observe new cyber phenomena. Darknets have been initiated to passively observe attacks that are perpetrated to target different pools of IP addresses. A brief review of Darknets' literature show their usefulness since they are used in:

- *Analysis of back-scattered packets*: This analysis aims at characterizing responses to spoofed Denial of Service (DoS) attacks. In the darknet, we can notice that the most common responses to a SYN flood packets are TCP packets with ACK/RST. We can observe common spikes of SYN/ACK and SYN/ACK/RST responses during a short duration.

- *Investigation of unique and multiple periodic probes*: We observe the time periodicity on collected data for the purpose of intrusion detection. For instance, we can isolate TCP flows for scanning services running through ports 139 (Server Message Block protocol over NetBIOS) and 445 (Direct Server Message Block protocol).

- *SMTP hot-spot analysis*: We can identify SMTP hot-spot. For instance in [157], the authors discovered the existence of an IP address, which attracted a large number of SMTP (Simple Mail Transfer Protocol) scans. This IP is bound to 14.000 IPs, which results in 4.5 million scans.

- *Detection of worms*: For instance, in iSink darknet deployment [229], the authors detected worms such as Sasser, which uses *lsarpc* exploit. Moreover, the authors managed through iSink to observe different Sasser variants and other malware propagation, namely, Agobot and RRBOT.CC.

Passive DNS Passive DNS or passive DNS replication is a technique invented by Florian Weimer in 2004 to store a partial view of the data available in the global Domain Name System into a centralized database where it can be queried and updated. Passive DNS databases are extremely useful for a variety of purposes. Malware and cyber-crime rely massively on DNS, and the so-called “fast flux botnets” abuse the DNS with frequent updates and low Time To Live (TTL). Passive DNS provides relevant insights and analytics upon DNS queries that users and/or malware may be performing. It has the ability to provide the following information:

- Pool of IP addresses associated with host-names.
- Tertiary name bound to specific domains.
- When a specific domain was resolved on the network.
- How many times a domain name has been resolved.
- Domains with short time to live (TTL's) may infer malicious activities.
- Non-approved DNS servers.
- Detection of fast flux and double flux of domains.

Spam Traps Spam-traps are a bench of e-mail addresses that are created not for communication, but rather to harvest spamming and fishing emails. In order to prevent legitimate email from being invited, the e-mail address will typically only be published in a location hidden from view such that an automated e-mail address harvester can find the email address, but no sender would be encouraged to send messages to the email address for any legitimate purpose. Since no e-mail is solicited by the owner of this spam-trap e-mail address, any e-mail messages sent to this address are immediately considered unsolicited. The term is a compound of the words "spam" and "trap", because a spam analyst will lay out spam-traps to catch spam in the same way that a fur trapper lays out traps to catch wild animals. The provenance of this term is unknown, but several competing anti-spam organizations claim trademark over it.

2.3 Prominent Cyber-Threat Analysis

The analysis of prominent threats is of a rewarding importance. It is considered as one of the cornerstones that IT community uses to gain knowledge about the inner-workings of different cyber-threats. Thus, beneficial information can be obtained by the analysis of malware binaries, their network traces, and the change in infected systems behavior. The analysis of prominent cyber-threats is an intelligence that helps security researchers to detect, prevent and eradicate such threats. As such, some security research efforts put an emphasis on the analysis of famous variants of cyber-threats. In the sequel, we discuss the different works done to unveil secrets of different cyber-threats.

In [149], the authors presented the analysis of an HTTP botnet, namely, *BlackEnergy*. The analysis provided a detailed information about the botnet architecture, commands and communication patterns. *BlackEnergy* is a web-based tool that allows to build bot binaries. The main threat of this botnet is Distributed Denial of Service (*DDoS*). Chiang and Lloyd studied the *Rustock* rootkit in [46]. This rootkit contains a spam bot module. The authors studied the network traces and noticed that the traffic is encrypted by *RC4* algorithm. The *Rustock* rootkit has multiple levels of obfuscation, which makes it hard to be detected. The main usage of this tool resides in mail spamming. In addition to the network analysis, the authors were able to extract the encryption key of the communication. Daswani *et al.* [52] put forward a detailed case study of *Clickbot.A*. This bot is responsible of click fraud attacks. Their analysis covered the main components of this botnet as well as the commands and configuration. Porras *et al.* reverse-engineered the Storm botnet in [167]. They detailed the techniques used to hide the binary and how it has been obfuscated. Storm botnet uses the *Overnet* protocol for the communication. This botnet is used to send email spams and DDoS attacks. In [89], the authors investigated the Storm botnet by studying the encryption key generation algorithm that is used for communication between different peers. In [60], the authors reported their analysis of the *Nugache* instance. They analyzed the communication pattern between different principals. The communication is based on a key exchange protocol. In *Nugache* botnets, the bot herder instructs bots to listen to a specific *IRC* channel in order to initiate a *DDoS* attack. The authors addressed extra aspects of their initial analysis and estimated the size of the *Nugache* botnet by using a bot client crawler. In [200], Stock *et al.* investigated the successor of *Storm* botnet, namely, *Waledac* botnet. Instead of using common reverse engineering to grasp the modus-operandi of *Waledac* botnet, they created a clone bot named *Walowdac*, which implements the same communication features of *Waledac* without causing any harm. The authors managed to observe that there have been $390,000$ infected machines throughout the world. They succeed to gather information about the success rates of corresponding spam campaigns and the credentials theft from infected machines.

2.4 Network Analysis: Graph Theoretic Approach

Various research efforts use graph theory for the purpose of studying social media networks. Java *et al*. [96] investigate micro-blogging phenomena through studying topological and geographical properties of Twitter's social network. They analyze people's intentions associated with different communities and show that users with similar intentions tend to connect with one another. Ugander *et al*. [99] study the structure of the Facebook social graph using different network features such as degree distribution, path length, clustering, and mixing patterns. The study concludes three key observations: (1) Facebook social network is nearly fully connected. (2) The graph neighborhoods of users have a dense structure. (3) The graph shows assortativity patterns related to users' friendships as well as age and nationality. Other researchers focus on the use of complex network analysis for the purpose of studying phenomena related to the Internet. In [158], the authors aim to derive a network model that is capable of explaining common structural characteristics of Internet Autonomous Systems (AS). They propose a framework, HyperMap, which replicates the geometric growth of complex networks on AS topology and identifies communities of AS belonging to the same geographic region. The authors show that their framework also has the ability to predict, with high precision, any missing links in the topology. Deri *et al*. [124] represent collected DNS with ".it" suffix data through complex graphs. They found that the Italian DNS ecosystem, represented through domain and resolver degree frequencies, follows power law distributions, and acknowledged the nature of DNS large scale evolution. In another work [55], the authors aim to rank Internet domains based on their popularity across resolvers. The authors validate their approach on Italian Internet domains. The ranking is based on node degree and Eigen-vector centrality metrics. Regarding threat network analysis, Nadji *et al*. [147] conduct an outstanding effort to unveil the structure of criminal networks. They use DNS history of known C&Cs, IP addresses found in blacklists, and spam URLs to build graphs. They develop a method based on the Eigen-vector metric to identify general structural trends and determine which strategy should be adopted for an effective remediation through take-down. The authors show that in many cases, by de-registering five domain names, many criminal networks can be taken down. Moreover, in one highlighted case, disabling 20% of criminal network hosts reduces the volume of successful DNS look-ups by 70%. Despite the interesting results shown by Nadji *et al*, we aim to provide more insightful information related to cyber-threat infrastructures by including new actors such as malware families, second-level domains, organizations, owners, etc. We also focus on the study of the evolution of cyber-threat infrastructures to understand their scale and forecast their potential evolution in the near future.

2.5 Traffic Fingerprinting and Malware Analysis

Regarding fingerprinting malicious traffic at the network level based on machine learning, we have done a literature review that encompasses two research threads, namely, (1) Network Traffic Analysis and (2) Malware Analysis and Classification. The former helps to look at the different techniques used to analyze traffic for the purpose of applications protocols fingerprinting, intrusion detection and identification of zero-day attacks, whereas the latter exposes the different works that dealt with malware classification and traffic analysis.

2.5.1 Network Traffic Analysis

Data mining techniques have been used in the analysis of network traffic for many purposes, i.e., application protocols fingerprinting, anomaly detection for intrusion and zero-day attacks identification. In protocols fingerprinting, many research efforts have been proposed. For instance, *Density Based Spatial Clustering of Application with Noise* [225] was proposed in 2008 to use clustering algorithms to identify various FTP clients, VLC media player, and UltraVNC traffic over encrypted channels. Li *et al.* [116] used wavelet transforms and k-means classification to identify communicating applications on a network. Alshammari *et al.* [18,20] put forward research efforts to identify `ssh` and `Skype` encrypted traffic (without looking at payload, port numbers, and IP addresses). Additionally, comparison of algorithms and approaches for network traffic classification were proposed separately by Alshammari *et al.* [19] in 2008 and Okada *et al.* [153] in 2011, surveying and comparing various machine learning algorithms for encrypted traffic analysis.

In addition to application protocols fingerprinting, many research efforts have been introduced to identify anomalies in traffic for the purpose of intrusion and malicious traffic detection. In 2000, Lee *et al.* [115] introduced a data mining approach for the purpose of intrusion detection. They described a data mining framework, which leverages system audit data as well as relevant system features to build classifiers that recognize anomalies and known intrusions. Bloedorn *et al.* [33] in 2001 described data mining techniques needed to detect intrusions along with needed expertise and infrastructure. Fan *et al.* [67] proposed an algorithm to generate artificial anomalies to force the inductive learner to segregate between known classes (normal traffic and intrusions) and anomalies. In [201], the authors provided an overview of *Columbia IDS Project*, where they presented the different techniques used to build intrusion detection systems. In [114], Lee reported on mining patterns from system and network audit data, and constructing features for the purpose of intrusion events identification. This work provided an open discussion about research problems that can be tackled with data mining techniques. Locasto *et al.* [122, 123] brought the use of collaborative security at the level of intrusion detection systems. They proposed a system that distributes alerts to collaborative peers. They integrated a component that extracts information from alerts and encodes it in Bloom filters. Another component is used to schedule correlation relationships between peers.

In [217], Wang *et al.* integrated a tool, namely, *PAYL*, which models the normal application payload of network traffic. The authors used a profile byte distribution and standard deviation for hosts and ports to train the detection model. They took advantage of *Mahalanobis* distance to compute the similarity of testing data against pre-computed profiles. If the distance exceeds a certain threshold, the alert is generated. Zanero *et* *al*. [235] presented a hybrid approach, which lies in: (1) an unsupervised clustering algorithm to reduce network packets payload to a tractable size, and (2) an anomaly detection algorithm, to identify malformed and suspicious payloads in packets and flow of packets. similarly, Zanero showed explicitly in [234] how *Self Organizing Map* algorithm (SOM) is used to identify outliers on the payload of TCP network packets. In [236], Zanero *et al*. extended their work by introducing approximate techniques to speed up the SOM algorithm at runtime. They provided more elaborated results and compared their work with existing systems. In [209], the authors introduced *Payload Content-based Network Anomaly Detection (PCNAD)*, which is a corroboration to *PAYL* system. They used *Content-based Payload Partitioning (CPP)* to divide the payload into different partitions. The subsequent anomaly analysis is performed on partitions of packet payloads. They showed that *PCNAD* has a high accuracy in terms of anomaly detection on port 80 by using only 62.64% of packet payload length. Perdisci *et al*. [163] presented the multiple classifier payload-based anomaly detector (*McPAD*). Like *PAYL* system, the authors use n-grams but with features reduction to avoid the curse of the dimensionality problem [62]. They applied a feature clustering algorithm proposed in [57] for text classification to reduce features. *McPAD* detects network attacks having shell-code in the malicious payload as well as some advanced polymorphic attacks.

Song *et al*. [193] introduced *Spectrogram* to detect attacks against web-layer code-injection (e.g., PHP file inclusion, SQL-injection, XSS attacks, and memory-layer exploits). They built a sensor that builds dynamically packets to construct content flows and learns to recognize legitimate inputs in web-layer scripts. They used the Mixture-of-Markov-Chains to train a model that detect anomalies in web-content traffic. Golovko *et al.* [78] discussed the use of neural networks and *Artificial Immune Systems* (AIS) to detect malicious behavior. The authors studied the integration and the combination of neural networks in modular neural systems to detect malware and intrusions. They proposed a multi-neural network approaches to detect probing, DoS, *user-to-root* attacks, and *remote-to-user* attacks. In [35], Boggs *et al.* elaborated on a system that detects zero-day attacks. The authors correlated web requests containing user submitted data considered abnormal by Content Anomaly Detection (CAD) sensors. Boggs *et al.* filtered the requests with high entropy to reduce data processing overhead and time. They evaluated their correlation working prototype with data collected during eleven weeks from production web servers. Whalen *et al.* [219] adapted outlier detection to cloud computing. The authors proposed an aggregation method where they used random forest, logistic regression, and bloom filter-based classifiers. They showed the scalability of their proposed aggregation content anomaly detection with indistinguishable detection performance in comparison with content anomaly detection classical methods. In [187], Shirani *et al.* proposed an intrusion detection in web-services based on the auto-regressive integrated moving average (ARIMA) model [127]. The model detects malicious behaviors within web-services using the predictive model, any behavior that falls out of the model confidence level is considered as an outlier (malicious).

As being the first step of an attack's vector, network scanning (reconnaissance) has been the target of many research efforts. For instance, Simon *et al.* [188] formalized the scanning detection as a data-mining problem. They converted collected datasets as a set of features to run off-the-shelf classifiers, like *Ripper* classifier. They showed that the data-mining models encapsulate expert knowledge that outperform in terms of coverage and precision in scanning identification. The emergence of botnets and malicious content delivery networks has pushed researchers to investigate the identification and detection of such networks. For example, in [31, 121], the authors put forward methods to detect IRC botnets. In [31], Binkley *et al.* presented an anomaly-based algorithm to detect IRC-based botnet meshes. The algorithm uses a TCP scan detection heuristic (TCP work weight) and other collected statistics gathered on individual IRC hosts. The algorithm sorts the channels by the number of scanners producing a list of potential botnets. The authors deployed a prototype in a DMZ and managed to reduce the number of botnet clients. In [121], Livadas *et al.* presented machine learning-based classification techniques to identify the command-and-control (C&C) traffic of IRC-based botnets. They proposed two-stages detection system. The first stage consists of distinguishing between IRC and non-IRC traffic, whereas the second lies in segregating botnet and real IRC traffic. In [103], Karasaridis *et al.* put forward an approach to identify botnet C&Cs by combining heuristics characterizing IRC flows, scanning activities, and botnet communications. They used non-intrusive algorithms that analyze transport layer data and do not rely on application layer information.

In [81], Gu *et al*. introduced *BotHunter*, which models all bot attacks as a vector enclosing scanning activities, infection exploits, binary download and execution, and C&Cs communication. The tool was coupled with *Snort* [195] IDS with malware extensions to raise alerts when different bot activities are detected. Based on statistical payload anomaly detection, statistical scan anomaly detection engines and rule-based detection, *BotHunter* correlates payload anomalies, inbound malware scans, outbound scans, exploits, downloads and C&C traffic and produces bot infection profiles. In [82], Gu *et al*. used aggregation technique to detect botnets. They explained how bot infected hosts have spatial-temporal similarity. They introduced *BotSniffer*, which is a system that pinpoints suspicious hosts that have malicious activities such as sending emails, scanning, and shared communication payloads in IRC and HTTP botnets by using shared bi-grams technique. In [80], Gu *et al*. exposed *BotMiner*, which aims to identify and cluster hosts that share common characteristics. It consists of two traffic monitors (C-plane and A-plane monitors) deployed at the edge of network. The C-plane monitor logs network flows in a format suitable for storage and analysis. The A-plane monitor detects scanning, spamming, and exploit attempts. The clustering components (C-plane clustering and A-plane clustering components) process the logs generated by the monitors to group machines that show very similar communication patterns and activity. The cross-plane correlator combines the results and produces a final decision on machines that belong to botnets.

Another noticeable research using aggregation technique was introduced in [230], where Yen *et al*. presented *TAMD*, an enterprise network monitoring prototype that identifies groups of infected machines by finding new communication flows that share common characteristics (communication “aggregates”) involving multiple network internal hosts. Their characteristics span over flows that communicate with the same external network, flows that share similar payload, and flows that involve internal hosts with similar software platforms. *TAMD* has an aggregation function, which takes as input a collection of flow records and outputs groups of internal hosts having a similarity value based on the input flow record collections. To reduce the dimensionality of vectors representing hosts, the authors used *Principal Component Analysis (PCA)*. To cluster different hosts, authors used k-means algorithm on reduced vectors. In [44], Chang *et al*. proposed a technique that detects P2P botnets C&C channels. They considered a clustering approach (agglomerative clustering with Jaccard Similarity criterion function) to capture nodes’ behavior on the network, then, they used statistical tests to detect C&C behavior by comparing it with normal behavior clusters. In [151], Noh *et al.* also defined a method to detect P2P botnets. They focused on the fact that a peer bot generates multiple traffic traces to communicate with a large number of remote peers. They considered that botnet flows have similar patterns, which take place at irregular intervals. They used a flows grouping technique, where a probability-based matrix is used to construct a transition model. The features representing a flow state are protocol, port, and traffic. A likelihood ratio is used to detect potential misbehavior-based transition information in state values. In [207], the authors introduced a novel system, *BotFinder*, which detects infected hosts in a network by considering high-level properties of the botnet network traffic. It uses machine learning to identify key features of C&C communication based on bots traffic produced in a controlled environment. Our approach has the same flavor of *BotFinder*; however, we create a detection model based on machine learning techniques by considering not only bots, but any malware type. In [59], Dietrich *et al.* introduced *CoCoSpot*, which recognizes botnet C&Cs channels based on carrier protocol distinction, message length sequences and encoding differences. The authors used average-linkage hierarchical clustering to build clusters of C&C flows. These clusters are then used as knowledge base to recognize potentially unknown C&C flows.

2.5.2 Malware Traffic Analysis and Classification

In addition to network analysis for the purpose of malicious and intrusion traffic detection described earlier, many research efforts have emerged to tackle malware classification. Part of our methodology shares some similarities with the related work on automatic classification of new, unknown malware and malware in general, such as viruses, web malware, worms, spyware, and others where pattern recognition and expert system techniques are successfully used for automatic classification [138]. Malware classification falls into system-based classification and network-based classification. Regarding the first strand, Schultz *et al.* [180] proposed a data-mining framework that automatically detects malicious executables based on patterns observed on some malware samples. The authors considered a set of system-based features to train classifiers, such as inductive rule-based learner (*Ripper*), which generates Boolean rules, and a probabilistic method that computes class probabilities based on a set of features. A multi-classifier system combines the outputs from several classifiers to generate a prediction score. In [25], Bailey *et al*. proposed a behavioral classification of malware binaries based on system state changes. They devised a method to automatically categorize malware profiles into groups that have similar behaviors. They demonstrated how their clustering technique helps to classify and analyze Internet malware in an effective way. Rieck *et al*. [174] aimed to exploit shared behavioral patterns to classify malware families. The authors monitored malware samples in a sandbox environment to build a corpus of malware labeled by an anti-virus. The corpus is used to train a malware behavior classifier. The authors ranked discriminative features to segregate between malware families. In [212], Trinius *et al*. introduced *Malware Instruction Set* (MIST), which is a representation of malware behavior. This representation is optimized to ease and scale the use of machine learning techniques to classify malware families based on their behavior. Bayer *et al*. [26] put forward a scalable clustering approach to group malware exhibiting similar system behavior. They performed dynamic malware analysis to collect malware execution traces. These traces are transformed to profiles (features set). The authors used *Locality Sensitive Hashing* (LSH) to hash feature values and improved scalability of profiles hierarchical clustering.

Wicherski [221] introduced a scalable hashing non-cryptographic method to represent binaries using a portable executable format. The hashing function has the ability to group malware having multiple instances of the same polymorphic specimen into clusters. Hu *et al*. [90] implemented and evaluated a scalable framework, namely, *MutantX-S*, that clusters malware samples into malware families based on programs' static features. The program is represented as set of opcode sequences easing the extraction of n-gram features. The dimensionality of vectors representing the features is reduced through a hashing function. Regarding malware network-based profiling and classification, Rossow *et al*. [176] provided a comprehensive overview about malware network behavior obtained through the use of *Sandnet* tool. The authors conducted an in-depth analysis of the most popular protocols that are used by malware, such as DNS and HTTP. Nari and Ghorbani [148] classified malware samples based on network behavior of malware. Their approach transforms pcap files representing malware families into a protocol based behavioral graph. The features (graph size, root out-degree, average out-degree, maximum out-degree, number of specific nodes) are extracted from these graphs and a J48 classifier was used to classify malware families. In [105], Kheir *et al*. presented *WebVisor*, a tool that derives patterns from Hypertext Transfer Protocol (HTTP) C&C channels. The tool builds clusters based on statistical features extracted from URLs obtained from malware analysis. The approach is a fine-grained, noise-agnostic clustering process, which groups URLs for the purpose of malware families' attribution.

2.6 Passive DNS Analysis Systems

Many techniques have been proposed for detecting malicious activities and distinguish them from legitimate domains using passive DNS traffic. Some techniques are used to detect malicious domains that linked to specific types such as fast flux and spam. In [164], Perdisci *et al*. introduced an approach to detect malicious fast flux services through passive analysis of recursive DNS traces, unlike the other works that are limited to extract the malicious fast flux domain names from spam emails [88, 109, 150, 160]. Perdisci's approach has the ability to distinguish malicious fast flux domain names from legitimate domains by characterizing features that pinpoint fast fluxing IP addresses. In [165], Perdisci *et al*, extended the previous work, where they detect passively flux networks from above local recursive DNS servers in contrast with the first work, where they used from below recursive DNS servers. In [22], Antonakakis *et al*. introduced Notos, which is a DNS dynamic reputation system. It differentiates the malicious activities from benign ones using many features. It assigns reputation scores for the new domains based on models of known benign and malicious domains. The score shows if the domain is malicious or benign. It has been deployed in a large ISP's network and has been able to find domains before the public blacklist. In another work [29], Bilge *et al*. introduced a system to detect malicious domains, namely, EXPOSURE. They characterize passive DNS logs to segregate between malicious and benign domains. The segregation is based on 15 features. They conducted experiments on 100 billion DNS requests and deployed their solution during two weeks in an ISP. They managed to identify malicious domains used in botnet command and control, spamming, and phishing. In another work [30], same authors extended their initial work by deploying EXPOSURE for 17 months. In [197], the authors put forward a technique to identify botnet using DNS queries. The system uses Naïve Bayesian classifier to segregate between malicious and benign domain names. The classifier achieves a detection rate of 82% and false positive rate of 8.30%. In [227], the authors introduced a technique, which correlates successful and failed DNS queries for the purpose of detecting DGA-based botnets based on the entropy of domain names. In [228], Yadav *et al.* presented a technique to detect DGA-based botnets by using distribution of uni-grams and bi-grams for all domains associated with the same IP address, TLD or SLD. In [47], the authors presented a system, namely, BotGAD (Botnet Group Activities Detection). They used an unsupervised approach (X-means clustering algorithm) to group domain names into clusters. Each cluster has a binary matrix, where rows are hosts sending DNS queries and columns represent time periods. This matrix is used to compute a cosine similarity score to decide if the cluster represents a botnet group or not. In [23], Antonakakis *et al.* proposed another system called Kopis, which is a detection system for malicious domains using upper DNS hierarchy. Kopis distinguishes between legitimate and malware domains using the global DNS query resolution patterns. In addition, it has the ability to detect malware domains in the absence of IP reputation information. The eight months experiment shows that Kopis identified new malware domains before the blacklist. Antonakakis *et al.* [24], presented a system that detects DGA-generated domains by analyzing Non-Existent Domain (NXDomain) responses without the need to reverse engineering. It uses two algorithms, which are clustering and classification algorithms. It clusters similar domain names in terms of structure. The classification algorithm refers the clusters to known DGAs models. If there is no model to assign, it generates a new DGA model. The system has been deployed on real time data. It was able to find new DGA families. In [185], Sharifnya *et al.* proposed a reputation system to detect DGA-based botnets based on the symmetric Kullback-Leibler divergence score to compute reputation of hosts mapping to a large number of suspicious domain names. Their approach marked domain names as dynamically generated if their distribution of uni-grams or bi-grams do not fit the normal distribution. In [186], the same authors proposed a negative reputation system that detects domain flux botnets. Unlike previously cited works, it relies on the history of the large number of malicious activities to a specific IP address beside the suspicious failures. It assigns a high negative score to the suspicious domains. In [102], Kara *et al.* proposed a detection mechanism for malicious payload distribution channels in DNS. The authors used a significant amount of DNS traffic to identify covert channels that abuse DNS protocol. They proposed a technique that counts the usage of resource records to detect payload distribution channels despite the fact that they have been rarely exploited.

2.7 Conclusion

In this chapter, we expose the different related works for the purpose of studying cyber-threat analysis. To this quest, we initially entail different definitions related to cyber-threat analysis such as the use case model, definition of different malware and sources of cyber-threat intelligence. Then, we introduce some works that describe some prominent threats found in the wild like *Blackenergy*, *Rustock*, *Nugache*, etc. These works helped us to gain a learning curve about malware analysis. In addition, we describe some works, where graph theory was used to characterize and study different networks. These works allow us to grasp how graph theory can be used to investigate cyber-threat infrastructures. Moreover, we put an emphasis on works using machine learning techniques to analyze the network traffic. These works shed the light on different techniques used to fingerprint applications on traffic or to detect intrusions and anomalies. Finally, we study different works that use DNS traffic as a source of cyber-threat intelligence. These works have been used to design and integrate a system to identify anomalies in DNS ecosystem.

Chapter 3

Prominent Cyber-Threats

3.1 Overview

In this chapter, we present two case studies on the analysis of prominent threats. We report details about Mariposa botnet and Zeus crime-ware toolkit respectively in Sections 3.2 and 3.3. The analysis of these threats is done with reverse-engineering analysis tools. We introduce a brief description of each threat as well as the different reverse engineering steps done to unveil their inner-workings. A discussion about these research efforts is entailed in Section 3.4.

3.2 Analysis of Mariposa Botnet

In this section, we analyze one of the most popular and prominent botnets, namely Mariposa [189], which infected more than 13 million computers located in more than 190 countries. We describe the botnet architecture, components, commands and communication. We detail the obfuscation and anti-debugging techniques it uses. Moreover, we detail the infection and code-injection techniques into legitimate processes. In addition, we explain the spreading mechanisms that are employed in Mariposa as well as the communication protocols. Furthermore, we analyze the injected bot code. This is accomplished by a reverse-engineering exercise that uses both network analysis together with reverse-engineering analysis. The insights from this work are meant to illustrate the know-how used in current botnet technologies and enable the elaboration of analysis, detection and prevention techniques.

3.2.1 Mariposa Botnet Description

In this section, we provide an overview of the Mariposa Botnet. We describe how the botnet works as well as the various features of the bot. Different variants that constitute Mariposa botnet mainly evolved from the so-called butterfly bot. The authors of Mariposa variants enhance the capabilities of the butterfly bot to make it more robust, resilient, stealthy and threatening. The botnet architecture consists of a set of clients, a server and a master. The architecture is connectionless because it is based on the UDP protocol (no guarantee to the upper layer protocols of message delivery). The server plays the role of the relay between the master and the clients. The UDP protocol is used due to its covertness: The UDP connections are not generally logged in firewalls and gateways, which is not the case with TCP connections. In order to check the presence of bot clients, the server pings clients periodically in a predefined time gap. If it does not receive any reply from the bot, the server marks it as a time-out bot. Further details about the communication protocol are described in the next section that reports on the network analysis of the botnet. We summarize Mariposa's features as follows:

- *Bot client*: The bot has innovative capabilities comparing to the majority of bots that exist in the wild. It has the ability to make direct code injection into remote processes. This injected code corresponds to the entry point of all activities that are done by the bot. Mariposa is capable of downloading any extra modules like the Zeus botnet and execute them on the fly. Besides, it is capable of performing UDP and TCP flooding, and can tune the flood strength by acting on the data and packet size, and send random data to the victim host. In addition, the bot has mechanisms to spread through the infection of USB keys or using MSN messenger and P2P applications. Moreover, the Mariposa bot contains a module that tracks the visited websites and a grabber that catches all the posted data that are sent from Internet Explorer 6, 7, and 8. On the other hand, the bot is endowed with two downloaders: The first one can download via HTTP, HTTPS and FTP protocols whereas the second downloads files via the ButterFly Network Protocol. Additionally, it has a built-in cookie stuffer for IE and Mozilla Firefox. Recently, Mariposa authors added new features like a flooder and a reverse proxy module, which can turn all bots into proxy servers.

- *Server*: The server is a mediator between the master and the bot clients. As such, it allows to control the traffic between them by setting the number of frames per second to diminish the CPU usage and the communication latency ratio. We can also set up the maximum upload on the server. The latter localizes the bots using IP geo-localization.

- *Master*: The master represents the core of all operations. It can get multiple server connections and has the ability to enable and disable servers and clients. The master sends commands to bot clients through servers. These commands are various and can be used to customize the operations that are done by clients. The next section reports on the results of our network analysis.

3.2.2 Network Analysis

Before digging into the inner details of the static analysis of the bot code, we analyze the network behaviors of Mariposa in a controlled environment to grasp the botnet behaviors. First, let us explain the experimental setup for the network analysis. The controlled environment is based on `VMware Server 2.0.3` [216] running on a Windows XP system. This software allows running multiple virtual machines in an isolated environment and gives a certain flexibility to create different types of network architectures. The network consists of a default virtual network, which behaves as a stub network. In our analysis, we use four hosts to build a virtual network. These hosts are used to set up the botnet. We installed a master, a C&C server and a host, which is infected by a Mariposa bot. The fourth host is used as sniffing box. It runs a live-CD for network security analysts [182]. The utility of this live-CD resides in logging all communications promiscuously in order to correlate events and monitor the network activities of the botnet. It also allows to verify whether backdoors are set or not. In addition, it can bind to any DNS server. As a result, network records can be created to simulate an Internet-connected network. For this intent, we used $c:\backslash windows\backslash system32\backslash drivers\backslash etc\backslash hosts$ file as a source of a domain name resolution. The communications within the botnet breaks into three phases: initialization phase, bot aliveness phase and action phase. All the phases involve the participation of the master, server and the bot client.

The initialization phase takes place after an infection. Once a bot infects a machine, it sends a *join server command*. This command allows a bot to register the IP address of the bot within the server. The latter acknowledges the registration by sending a *join acknowledgment* packet. By receiving this command, the bot sends an acknowledgment to the server and *command-response* packet. The latest message contains the bot information like system information and the country code. The server sends an acknowledgment to the bot and forwards the *command-response* to the master, which acknowledges the reception of this message to the server.

The second phase aims at checking the aliveness of bot clients. The server keeps sending *command-response* packets to the bot client in a frequency of four minutes. If a given bot is alive, it replies with an *acknowledgment* packet.

The action phase aims to instruct the bots to make actions at the infected hosts. The master sends *command-response* packet to the server. The server forwards this packet

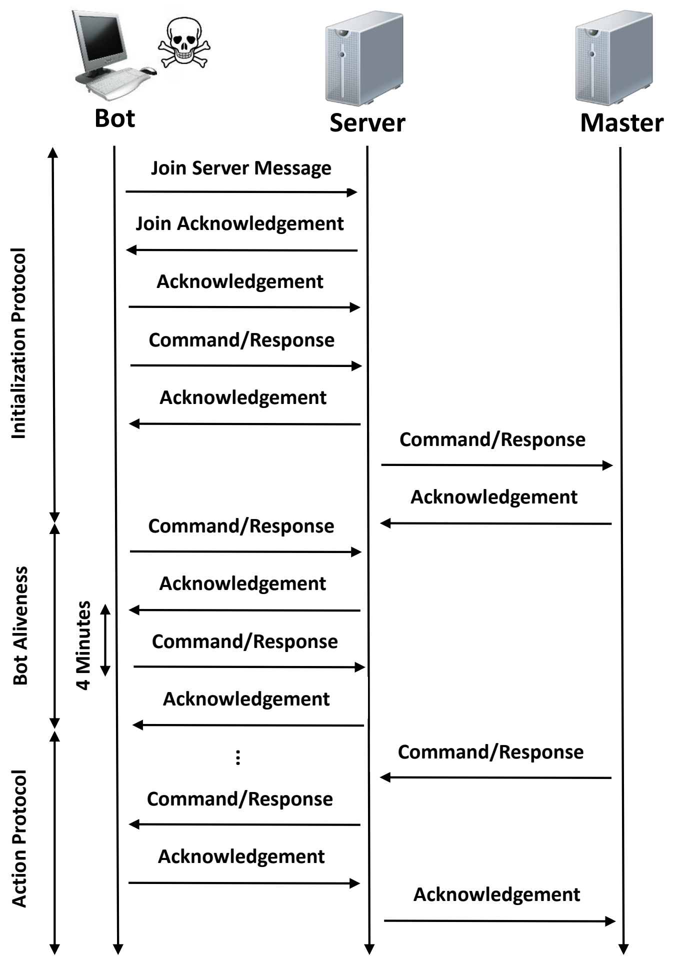


Figure 3.1: Mariposa Botnet Protocol

to the bot. By receiving the packet, the bot performs the action that is mentioned in the packet. It acknowledges its action by sending an *acknowledgment* packet. The server sends an *acknowledgment* packet to the master. Figure 3.1 depicts three phases of the Mariposa botnet communication. The next section is devoted to the results of the static analysis.

3.2.3 Static Analysis

The static analysis constitutes a must when it comes to reverse-engineer malware. Actually, it allows digging into the inner-secrets of the malware code. In our analysis, we used `IDA pro` [53] disassembler and de-compiler to analyze the Mariposa bot client. The MD5 hash of the malware variant is *3E3F7D8873985DE888CE320092ED99C5*. Before digging into the details of the static analysis, we used `SysAnalyzer` [94] to get an initial insight about the client. After running this tool, we noticed that Mariposa infects *explorer.exe* process. This process opens the UDP port 1055. Moreover, `SysAnalyzer` reveals the registry keys and external references that are accessed by the Mariposa bot.

The static analysis consists of getting over the obfuscation and anti-debugging techniques that are employed by Mariposa as well as reaching the susceptible parts of the code that execute Mariposa bot features that we previously described in Section 3.2.1. The Mariposa binary has a metamorphic code [204], comprised of various obfuscation and anti-debugging techniques. Figure 3.2 depicts the different phases of Mariposa bot metamorphose. The execution of the bot client has three phases: the obfuscation phase, the decryption phase and the injection phase.

In the sequel, we introduce the different phases that are related to the de-obfuscation, anti-debugging traps and different decryption layers.

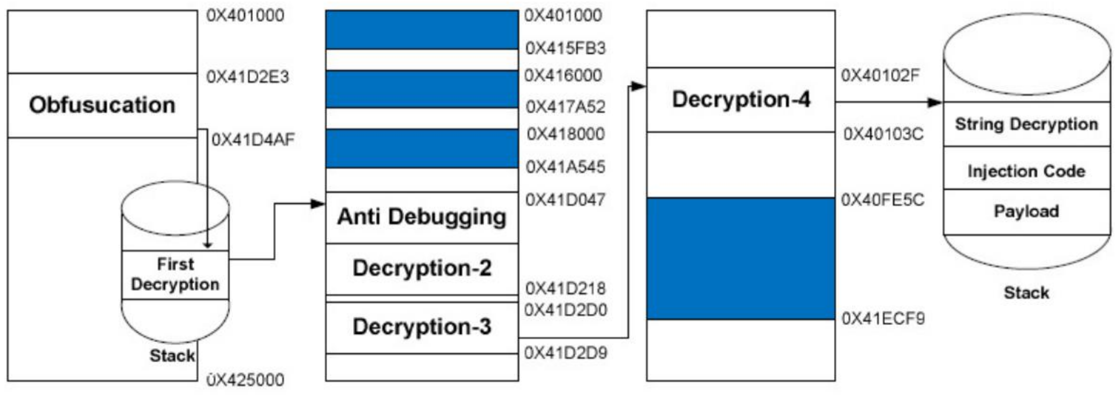


Figure 3.2: Overview of Mariposa Bot

De-Obfuscation and First Decryption Layer

Code obfuscation is nowadays a standard practice within Malware. It constitutes the concealment of the intended meaning of an integrated malicious code. It makes the code confusing and intentionally ambiguous and more difficult to interpret. In the Mariposa bot, the obfuscation starts with useless computations. These computations are done within a loop that iterates 889,976,605 times. At the end of this loop, a jump to an address is loaded into *EAX* register. As a consequence, a jump is initiated to start a routine that *XOR*s the range of data that is located between the addresses*0x41D000* and *0x41D4C0* with the constant *0x0CA1A51E5*. Afterwards, the address *0x41D047* is pushed into the stack. As a result, the control flow is transferred to this address. The latter corresponds to an entry point of the anti-debugging traps.

Anti-Debugging Traps

Anti-Debugging techniques detect if a program runs within a controlled environment or a debugger. They are used by commercial executable protectors, packers and malicious software to prevent or slow-down the process of reverse-engineering [181]. The Mariposa bot client uses several anti-debugging techniques. These techniques make the reverse-engineering tasks as strenuous and difficult as possible. They increase the time that is required for a full analysis of the bot binary. The address *0x41D047* constitutes the entry point of the code that employs anti-debugging traps. The most important anti-debugging techniques that have been encountered in the analyzed variant of the Mariposa bot are:

- *ICE Breakpoint* (In Circuit Emulator): It is one of Intel's undocumented instructions with opcode *0xF1*. The execution of this instruction generates a single step exception. This instruction pushes a debugger to think that a normal exception is generated by the program. It sets the single step bit in the flag register. Thus, the associated exception handler is not executed.

- *QueryPerformanceCounter Function*: It is used to compute the hardware performance. It reads the values of performance counters that are stored in some processor registers[1]. Mariposa uses this function to compare hardware activities with a threshold value and checks if a process is running under debugging mode or not.

[1]Contemporary processors use registers that act like performance counters. They count performance of hardware activities within the processor.

- *GetTickCount Function*: It is located in *kernel32.dll*. It returns the number of milliseconds that the system has elapsed since its last reboot. The highest return value is *49.7* days. Malware calls the *GetTickCount* function consecutively to calculate the difference between two function calls. It allows the malware to detect the presence of a debugger.

- *OutputDebugString Function*: It is generally used by encryption programs. The function receives a string as a parameter. If a program runs under a debugger, then, the returned value of this function (value of EAX register) corresponds to the address of the string that is passed as parameter. Otherwise, it returns the value 1.

- *Stack Segment Register*: This technique consists of pushing and popping the content of the stack segment in order to mislead the debugger. It forces the debugger to not break on a *PUSH* instruction and stop on *NOP* instruction.

Second, Third and Fourth Decryption Layers

After unveiling and removing the obfuscation and the anti-debugging routines, we reach the part of code that contains the decryption routines. The second layer of the decryption corresponds to an iteration of a *XOR* operation with a 32 bytes key. Each byte within the data is *XOR*ed with a byte from the key. This byte corresponds to a modulo result of data byte position with the size of the key (32 bytes). This algorithm is iterated three times for three different chunks of data. The first location of data corresponds to the range [*0x401000*, *0x415FB3*], the second location of data resides in the range [*0x416000*, *0x417A52*] and the third location of data is within the range [*0x418000*, *0x41D21E*]. There exist three 32 bytes keys; each one is used in the algorithm for each chunk of data. These keys are located at the following addresses: *0x41D015*, *0x41D155* and *0x41D1B4*. Figure 3.3 illustrates the pseudo code of the second decryption layer. The value *x* corresponds to the key location, whereas *r1* and *r2* are the start and the end addresses of data respectively.

```
Second_Decryption_Layer()
{
  Key_size=32 byte;
  Key_location = x;
  Key[]=getKey(x);//For first decryption layer.
  Start_address=r1;
  End_address=r2;
  Enc_data[]=getData(Start_address, End_address);
  for(i=0;i<Enc_data.size();i++){
    Dec_data=Enc_data[i] XOR Key[i % 32];
  }
}
```

Figure 3.3: Pseudo Code of the Second Decryption Layer

After executing the second layer decryption, the control flow reaches the part that is responsible of loading the imported functions. The next step consists of running another decryption routine (third layer decryption). This decryption takes place after the first layer decryption. It *XOR*s each byte of data in the range [*0x41D000*, *0x41D21E*] with a constant key *0x39*.

After executing the third layer decryption, the program loads its process and thread identifiers by calling *GetCurrentProcessID* and *GetCurrentThreadID* functions. It uses some anti-debugging traps using the *QueryPerformaceCounter* and *GetTickCount* functions. The intent behind this is to check again whether the current process runs under a debugger or not. In order to check whether it runs in a sandbox technology, it verifies the presence of *sbiedll.dll* in the system. By getting over these traps, we notice that the program allocates $60,925$ bytes of space from the stack. It decrypts the data in the range [*0x40FE5C*, *0x41EC59*] by utilizing the algorithm that is illustrated in Figure 3.4, and loads into the allocated space of the stack. Afterwards, Mariposa transfers its control to the stack.

```
Fourth_Decryption_Layer()
{
  Key1=getByte(0x418CA2);
  Key2=getByte(0x418CA3);
  Key1=((! Key1) + Key2) / 2;
  Source_address= 40FE5C;
  Enc_data[0xEDFD] = getData(Source_address, Source_address +0xEDFD );
  Dec_data[0xEDFD]=null;
  Dest_address = 0xXXXX;//in the stack.
  for(i=o; i<Enc_data.length ; i++){
    Dec_data[i]= (Enc_data[i] + Key1) XOR Key2;
    If(Key1==0xFF){
      Key2= (Key2+1) % 0xFF;
    }
    Key1= (Key1+1) %0xFF;
  }
}
```

Figure 3.4: Pseudo Code of the Fourth Decryption Layer

At this point, Mariposa code passes several phases of decryption. However, all the strings are encrypted. These strings represent API functions and magic words that will be used by the injected process. Once the fourth layer decryption is executed, the program runs a decryption routine three times. This routine decrypts all the strings that are located in *.data* section. Figure 3.5 illustrates the pseudo code of the string decryption.

```
Decrypt_Strings()
{
  Start_add=0x4197E0;
  Size=0xD65;
  Enc_data[]=Get_data(Start_add,Start_add+Size);
  Key1=Get_byte(0x418CA2);
  Key2=Get_byte(0x418CA3);
  Key=(Key2+ ~Key1) >> 1;
  for(i=Size; i >= 0; --i){
      Dec_data[i]=(Enc_data[i]+Key) XOR Key2;
    Key=(Key++)%255;
  }
}
```

Figure 3.5: Pseudo Code of String Decryption Algorithm

Code Injection

Despite substantial improvement in host-based security, the code injection technique sustains as the favorite method to compromise operating systems. The method of code injection is used to conceal evil processes inside legitimate processes. The execution of a process inside another address space can be achieved in several ways. We can enumerate windows hooks [93], dll injection and direct code injection [223]. The Mariposa bot uses the Direct Code Injection (DCI) technique to inject malicious code inside the address space of *explorer.exe*. Instead of writing a separate DLL, the DCI technique copies the malicious code to the remote process directly via *WriteProcessMemory* function and starts its execution with an invocation of the *createRemoteThread* function. The direct code injection (DCI) technique can be summarized into the following steps:

- Retrieval of the handle of the remote process by calling the *OpenProcess* function
- Allocation of memory in the remote process address space in order to inject code.

This is achieved by calling the *VirtualAllocEx* function.

- Writing a copy of the initialized *INJDATA* structure to the allocated memory by invoking the *WriteProcessMemory* function
- Execution of the injected code via the *CreateRemoteThread* function

Before code injection, Mariposa creates some directories and files. The created directories and files are:

- Directory Path: $C:\backslash Recycler\backslash s-1-5-21$.
- Directory Path: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454$. The directory access control is set to read, write and execution permissions.
- File Name: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash Desktop.ini$.
- File Name: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$.

Then, the program calls the *GetVersion* function to get the version of the operating system. The reason behind this call resides in checking whether the operating system is a Windows NT or not. If so, it uses the *CreateRomoteThread* function [2]. At the beginning of the injection process, the program calls the *CreateToolhelp32Snapshot* function to take

[2] *CreateRemoteThread* function works only in Windows NT versions.

a snapshot of the running processes in the system. It enumerates the existing processes by calling *Process32First* and *Process32Next* functions. Once *explorer.exe* process is found, it retrieves its process identifier (process ID).

After getting the process ID, the program calls *OpenProcess* function to open *explorer.exe* process. Then, it calls *VirtualAllocEX* function to allocate memory within the targeted process and *NtWriteVirtualMemory* function to write into *explorer.exe* process. Once the code is written in a virtual memory location, the program calls the *CreateRemoteThread* function in order to run the injected code.

Injected Thread Activity

The code that is injected into *explorer.exe* is the pivotal part of Mariposa bot. In the following, we discuss the behaviors of the injected code. To this end, we attached the process *explorer.exe* to `IDA pro` debugger and set a breakpoint at the entry point of the newly created thread to get full control of the execution. The thread creates a *mutex* object namely *c__kdjcpeoij*. The *mutex* object is used to ensure singular execution of the bot. The intent is to avoid a possible running of multiple bot instances, which can crash the system, or at best slow down the machine. It uses the *WaitForSingleObject* function with a predefined waiting time to ensure singular execution. Once the single instance checking is ensured, it creates two files: $C:\backslash Recycler\backslash S{-}1{-}5{-}21{-}7344526690{-}8558129233{-}739613093{-}1787\backslash windll.exe$ and $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash Desktop.ini$. After the file creation, the thread copies the whole bot code to $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$. At this point of execution, Mariposa uses the *WsaStartup* function to initiate the use of *Winsock DLL*, which is responsible for the socket communication. It also opens the registry key $software\backslash Microsoft\backslash WindowsNT\backslash CurrentVersion\backslash Winlogon$, and creates a new entry, namely, *Taskman*. It sets the value of this entry to $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$ in order to make a direct injection of code when the machine reboots. It also creates another entry named *shell* with the value $C:\backslash Recycler\backslash S-1-5-21-7344526690-8558129233-739613093-1787\backslash windll.exe$.

At this stage, the bot creates two pipes. The first one is $\backslash\backslash.\backslash pipe\backslash cdcpr55$ whereas the second is an anonymous pipe. The first pipe is created in *pipe_access_inbound* mode, which supports client to server transfer only. Once the pipes are set, the program calls the *InternetOpen* function in order to use the *WinInet* library functions. Mariposa bot uses three hard-coded domain names to resolve the IP address of the C&C server. It picks the first domain name and sends the encrypted magic word to the resolved IP address, and waits for the reply from the server. If the server does not respond, it picks the second or third domain name and tries to connect to the server using the resolved IP address. The domain names that are used for this Mariposa variant are:

- Shv4.no-ip.biz
- Shv4b.getmyip.com

- Booster.estr.es

The sequence of actions that are taken by the Mariposa bot to reach the server and receive commands are:

- The function *Inet_addr* is called to convert the domain names into a proper address.
- The bot retrieves the host information from the corresponding host name using the *gethostbyname* function.
- The bot calls the *htons* function, which converts an unsigned short number from a host to a TCP/IP network byte order [3].
- The bot encrypts the magic word (*bpr1* is the magic word in this variant of Mariposa). The encryption/decryption algorithm is detailed in [95].
- The bot sends the magic word using the *sendto* function.
- The bot receives a reply from the server using the *recvfrom* function.
- The bot decrypts and decodes the received command. The bot can then trigger appropriate actions that are instructed by the master.

[3]Network byte order defines the bit-order of network addresses as they pass through the network. The TCP/IP standard network byte order is big-endian. In order to participate in a TCP/IP network, little-endian systems usually bear the burden of conversion to network byte order [92].

3.2.4 Modules

Spreader Module

The Mariposa bot comes with a spreader module. This module breaks into three different components, namely, USB spreader, MSN spreader, P2P spreader. In the Mariposa botnet, the master can send commands to enable and disable the spreaders. In the sequel, we introduce these different components:

- *USB spreader*: At the beginning, the program creates a new top-level window by executing *CreateWindowEx* function. The returned handle is used by the *Register-DeviceNotification* function in order to receive notification from the system when a flash drive is inserted. Once a user inserts a USB key, it locks the *autorun.inf* file and modifies the file accordingly. As a result, no software or malware can launch an auto-run. The file stays locked until a user decides to remove the USB key. Mariposa makes a copy of itself into the USB key.

- *MSN spreader*: The Mariposa bot infects MSN messenger by hooking sending and receiving functions. The MSN spreader is activated if a bot receives an enabling command. This command contains a custom link, which is used to download a bot in the user's machine.

  After receiving the MSN spreader activation command, the bot looks for the *msn-msgr.exe* process. This operation is done periodically if the process is not running in the system. Once the *msnmsgr.exe* process is found, the Mariposa bot retrieves its process identifier. Then, it calls the *OpenProcess* function to get the handle of this process. Afterwards, it creates a duplicate handle of the current process by calling *GetCurrentProcess* and *DuplicateHandle* functions. At this point, the Mariposa bot starts a new routine, which is responsible for injecting code inside the virtual address space of *msnmsgr.exe* process. This routine is called twice. In the first call, it allocates 256 bytes of space by calling *VirtualAllocEX* function and injects code using *NtWriteVirtualMemory* function. In the second call, it injects string utility functions and the custom link that is sent by the master. It creates a thread by calling *CreateRemoteThread* function. After the injection process, the bot hooks *ws2_32_send* function in order to make the injected code executed for each message that is sent from a user to a recipient. This is done by calling the *VirtualProtectEx* function to allows writing in the virtual memory. At the end, it calls the *NtWritevirtualMemory* function to overwrite with the address of injected code.

- *P2P spreader*: When the bot receives a command that enables the P2P spreader, the program calls the *GetEnvironmentVariable* function in order to get the registry entry for the current user. The intent behind this resides in checking if P2P applications are installed or not. The Mariposa bot looks for the following P2P applications in the system: `Ares`, `BearShare`, `iMesh`, `Shareaza`, `Kazaa`, `DC++`, `eMule` and `LimeWire`. Once, it detects the presence of a P2P application, it copies itself into the shared folder with a fake name that is issued by the master.

Uploader and Downloader Modules

During the analysis of the main thread, we noticed that when the bot receives update/-download commands, it triggers two new threads. To debug these threads in `IDA pro`, we set a breakpoint at the beginning of each thread. When Mariposa bot transfer its control to one of these threads, we suspended the original thread in `IDA pro` and continued debugging with the new thread.

Thread 1 Mariposa starts this thread when the bot receives a download command. After receiving this command, the bot checks the command. If the latter corresponds to *descargar*[4], the thread launches the following activities:

- It targets the temporary location in the system to download a new executable.
- It calls the *InternetOpenUrl* function with the supplied *url*.
- If the *InternetOpenUrl* function succeeds, the bot creates a file in the temporary location by calling the *CreateFile* function.
- It downloads the file using the *InternetReadFile* function.
- It writes the file onto the disk by invoking the *WriteFile* function.
- It uses the *CreateFile* function again to create the file.

[4]Descargar is a Spanish word, which means download

After downloading the file, the bot checks the first two bytes to ensure whether the downloaded file is an executable or not. If so, it runs the file by calling the *CreateProcess* function and exits the thread by calling the *ExitThread* function.

Thread 2 This thread starts when the bot receives an upload command. After receiving the command, the bot checks the command and compares it with *subir*[5]. If the comparison is successful, the thread executes the following activities:

- It calls the *InternetCrackUrl* function to read different *url* components.
- By getting the *url* components, it calls the *InterConnect* function to set a connection with the *url*.
- It uses the *HttpOpenRequest* function to create an *HTTP* request.
- It invokes the *InternetReadFile* function to read data to be sent.
- It sends the data using the *HttpSendRequest* function.
- Finally, it closes the connection handle using the *InternetCloseHandle* function.

After uploading the file, the thread calls the *exitthread* function to close the thread.

[5]Subir is a Spanish word, which means upload

Components Diagram

By conducting a thorough reverse-engineering task, we noticed that Mariposa bot has complex interactions between its functional components. Figure 3.6 illustrates the different interactions between the different functional components.

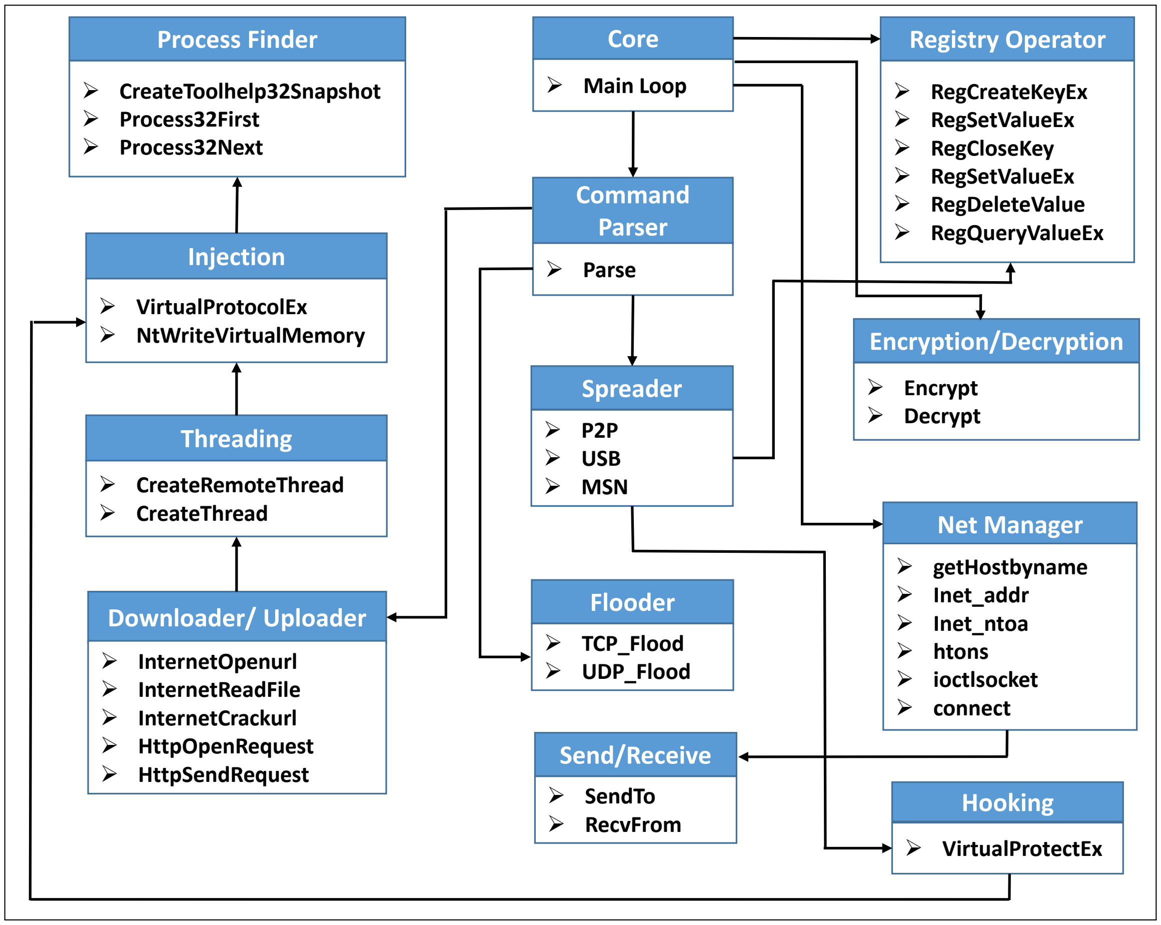


Figure 3.6: Component Diagram

3.3 Analysis of Zeus Botnet Crime-ware Toolkit

In this section, we present a reverse engineering effort done on the Zeus crime-ware toolkit [32]. The latter is one of the powerful crime-ware tools that emerged in the Internet underground community to control botnets. Zeus has infected over 3.6 million computers in the United States. Our analysis aims to uncover the obfuscation levels from packed Zeus code. In the prevailing of this thought, we describe the bot building and infection processes. In addition, we put forward a method to extract the encryption key from the malware binary and use it to decrypt the communications of the botnet and its configuration information. We combine reverse engineering and network analysis to understand behaviors of this new generation crime-ware toolkit. After performing the reverse engineering exercise, we found out that C&C communications indicates that the authors used RC4 algorithm. We uncovered the format of messages that are sent through a network.

3.3.1 Zeus Botnet Description

The Zeus crime-ware toolkit is a set of programs which have been designed to setup a botnet over a high-scaled networked infrastructure. Generally, the Zeus botnet aims to make machines behave as spying agents with the intent of getting financial benefits. The Zeus malware has the ability to log inputs that are entered by the user as well as to capture and alter data that is displayed into web-pages [87]. Stolen data can contain email addresses, passwords, online banking accounts, credit card numbers, and transaction authentication numbers. In our analysis, we examine the Zeus crime-ware toolkit v.1.2.4.2, which is considered as the latest stable publicly available version in the underground community. The overall structure of the Zeus crime-ware toolkit consists of five components:

- A control panel, which contains a set of PHP scripts that are used to monitor the botnet and collect the stolen information into `MySQL` database and then display it to the bot-master. It also allows the bot-master to monitor, control, and manage bots that are registered within the botnet.
- Configuration files that are used to customize the botnet parameters. It involves two files: the configuration file `config.txt` that lists the basic information, and the web injects file `webinjects.txt` that identifies the targeted websites and defines the content injection rules.
- A generated encrypted configuration file `config.bin`, which holds an encrypted version of the configuration parameters of the botnet.
- A generated malware binary file `bot.exe`, which is considered as the bot binary file that infects the victims' machines.
- A builder program that generate two files: the encrypted configuration file `con-fig.bin` and the malware (actual bot) binary file `bot.exe`.

On the C&C side, the crime-ware toolkit has an easy way to setup the C&C server through an installation script that configures the database and the control panel. The database is used to store related information about the botnet and any updated reports from the bots. These updates contain stolen information that are gathered by the bots from the infected machines. The control panel provides a user friendly interface to display the content of the database as well as to communicate with the rest of the botnet using PHP scripts. The botnet configuration information is composed of two parts: a static part and a dynamic part. In addition, each Zeus instance keeps a set of targeted URLs that are fed by the web injects file `webinject.txt`. Instantly, Zeus targets these URLs to steal information and to modify the content of specific web pages before they get displayed on the user screen. The attacker can define rules that are used to harvest a web form data. When a victim visits a targeted site, the bot steals the credentials that are entered by the victim. Then, it posts the encrypted information to a drop location that is meant to store the bot update reports. This server decrypts the stolen information and stores it into a database.

3.3.2 Network Analysis

In this section, we explain the network communication that occurs between the C&C server (the server containing the control panel) and an infected machine. Such analysis can be used to write IDS rules and anti-virus detection routines. In order to perform the network analysis, we built a sandbox environment to collect and analyze the network traces that are generated from the communication between the C&C server and one of the bot instances. We configured a web server, which acts as the C&C server and the drop location. This server hosts all resources that are required to operate the botnet (`config.bin` file, `PHP` scripts and the `MySQL` database). To customize the malware, we used the builder program to generate the malware binary file, which is configured to communicate with a C&C server. Within our environment, fake websites are generated to reflect real scenarios of botnet attacks. All necessary entries of the configuration file as well as the web injects scripts are modified to target the fake website. After infecting a machine with the bot binary file, we collected network traces for one day. During this session, the user of the infected machine visited the targeted website and then used login credentials, personal information, and credit card information for testing purposes.

By analyzing the bot network communications, we can learn the overall behavior of the Zeus botnet. The network behavior of the Zeus botnet constitutes a starting point, where we can dig into the crime-ware toolkit functions. Since the Zeus botnet is based on HTTP protocol, it uses a pull-method to synchronize the botnet communications. From the collected network traces between a bot and a C&C server, we observe that the bot periodically checks specific servers for an up-to-date configuration and bot binary files. Moreover, HTTP communication messages between the two entities are encrypted. By observing the network trace, we managed to determine the following communication pattern between the C&C server and the infected machine:

- The infected client starts the communication by sending a request message `GET /config.bin` to the C&C server. This message is a request to fetch the configuration file for the botnet.

- The C&C server replies with the encrypted configuration file `config.bin`.
- The client receives the encrypted configuration file and decrypts its content by using an encryption key, which is embedded inside the bot binary file.
- In situation where the bot-master wants to involve the infected machine to manage the botnet, the infected machine has to provide its external IP address and report any use of Network Address Translation (NAT). In order to know the external IP address that is seen by the botnet servers, the infected machine makes a request to a specific server. Afterwards, this server informs the infected machine about their externally facing IP address. The server's URL is provided in the static configuration file.
- The bot posts the stolen information and its update status reports to the C&C server `POST/gate.php`.

Figure 3.7 illustrates the communication pattern between the C&C server and the infected machine. The communication pattern is repeated frequently depending on a timing variable, which is defined in the botnet configuration file.

3.3.3 Static Analysis

The increasing usage of malicious software has pushed security experts to try finding the secrets related to the development of malware design. A common technique to detect the existence of a given malware is by tracking system modifications. The changes include what an operating system runs at start-up, changes of default web pages, generated traffic,

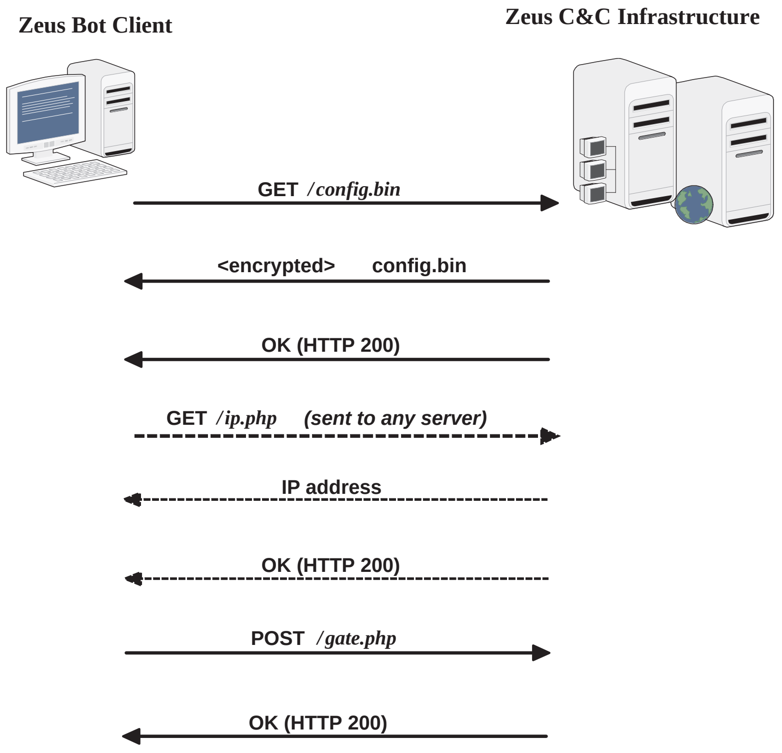


Figure 3.7: Communications Pattern of Zeus

infection of processes, packing/unpacking of binaries, and changes to the registry keys. One way to look for these changes is to reverse engineer the malware and try to reveal what is hidden behind the assembled code. In our case, this kind of analysis provides an invaluable insight into the inner-working of the crime-ware toolkit in general and about the malware binary in particular. In the stream of this thinking, we investigate the builder program and malware binary file. To this end, we mainly employ `IDA Pro` to disassemble the binaries and debug them to understand their business logic. The analysis is two folds: First, the analysis that is related to the builder program. Second, the analysis that is linked to the malware binary file.

Analysis of the Zeus Builder Program

The builder is one of the components of the Zeus crime-ware toolkit. It uses the configuration files as an input to generate the bot binary file and the encrypted configuration file. We analyze the builder program first because it uses a known obfuscation technique that can be easily removed. In addition, the GUI allows us to categorize different subroutines, which make up the builder program functionalities. Using the `PaiMei` reverse engineering framework [5] (which provides many reverse engineering tasks such as fuzzer assistance, code coverage tracking, and data flow tracking), we were able to see exactly what functions of the builder program are invoked by a specific action. This immensely aids in simplifying the reverse engineering efforts as it allows us to focus on a few key subroutines. In the following, we summarize the reverse engineering analysis of the functions of the builder program.

- *Building the Configuration File Functionality*: This function is responsible for encoding the clear text of the configuration files of the botnet into a specific structure. Afterwards, it encrypts the whole structure with the RC4 encryption algorithm using the configured encryption key.
- *Building the Malware Binary File Functionality*: The main function of the builder program resides within this functionality, which is responsible for building the customized malware binary files. In general, it builds the malware executable file into

a portable executable (PE) standard format. Moreover, it sets some parameters according to the current configuration file and then produces the malware binary file.

- *Malware Infection Removal Functionality*: The builder has a functionality that ascertains the presence of Zeus bot and removes it. When this functionality runs, it performs a detection routine by checking the existence of special registry keys that are inserted during the bot infection process. Also, it detects the presence of some files in the system. If these files are detected, the builder program cleans some registry keys and instructs the bot to shutdown itself and then deletes the stored Zeus binary file from the system. The expected behavior of the bot when it receives the shutdown command is to disinfect itself from the currently running processes. The analysis reveals the file names that the builder checks their presence in the system. Table 3.1 represents these file names with their description.

File	**Description**
C:/WINDOWS/system32/sdra64.exe	A copy of a bot which has infected “system32” folder.
C:/WINDOWS/system32/lowsec/local.ds	A data storage file which is used to store the configuration file that is used by a given bot locally in the system.
C:/WINDOWS/system32/lowsec/user.ds	A data storage file which is used to log the users’ activities that have been recorded by the bot.

Table 3.1: Files Created During the Bot Infection

Zeus Bot Binary Analysis

As depicted in Figure 3.8, the bot binary file contains four segments: A “text/code” segment, an “imports” segment, a “resources” segment, and a “data” segment. Therefore, we begin our analysis at the malware Entry Point (EP) that resides in the “text/code” segment.

The initial analysis of the disassembly reveals that only a small part of the “text/code” block is valid computer instructions. The rest of the binary is highly obfuscated, which means that the computer cannot use these segments directly unless it is de-obfuscated at some stage.

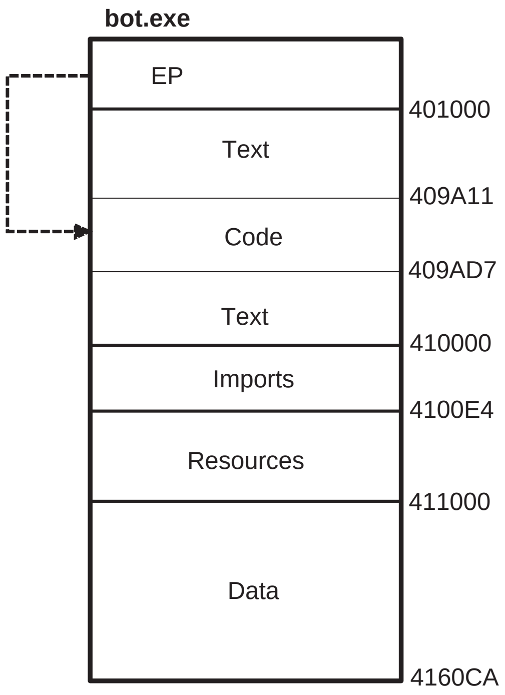


Figure 3.8: Segments of the `bot.exe` Binary File

- *De-obfuscation Process*: By using the `IDA Pro` debugger, we were able to debug the malware and step through the instructions to analyze and understand the logic of the de-obfuscation routines. Each routine reveals some information which is used by the other routines until all obfuscation layers are removed. The first de-obfuscation routine contains a 4-byte long decryption key and a one-byte long seed value. These two values are used to decrypt a block of data from the “text/code” segment and then write the decrypted data in the virtual memory. The result of the first de-obfuscation routine revealed some new code segments. These segments contain three de-obfuscation routine, as shown in Figure 3.9. During our analysis, the initial memory offset address for the code segments was `0x390000`. After the address space of the second de-obfuscation routine, there was an 8-byte key that `IDA Pro` incorrectly identified as code instructions. Figure 3.10 illustrates the location of the 8-byte key. In the following, we explain the main logic of the second de-obfuscation routine.

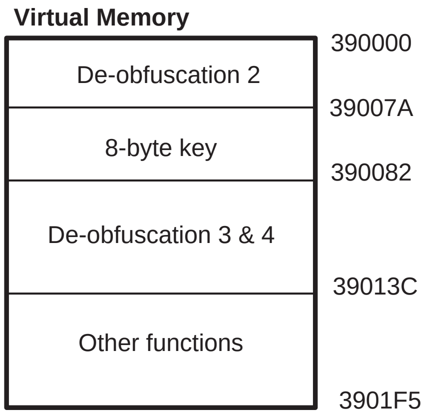
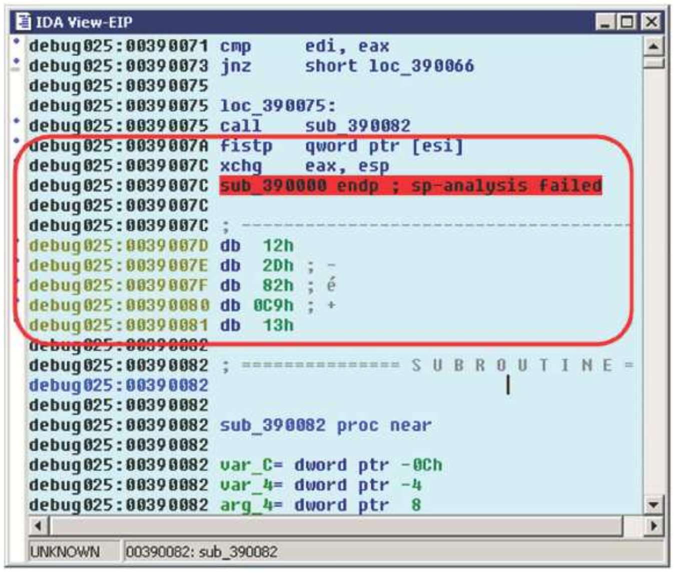


Figure 3.9: De-Obfuscated Code in the Virtual Memory

1. First, the routine copies two binary blocks from the “text/code” segment, concatenates them together, and then writes them into the virtual memory. The first text block contains data with many zero value bytes that will be filled by the next text block, as shown in Figure 3.11.
2. The routine scans every byte on the first text block and when it encounters a “hole” (zero byte), it overwrites the zero byte with the next available byte in



```
debug025:00390071 cmp     edi, eax
debug025:00390073 jnz     short loc_390066
debug025:00390075
debug025:00390075 loc_390075:
debug025:00390075 call    sub_390082
debug025:0039007A fistp   qword ptr [esi]
debug025:0039007C xchg    eax, esp
debug025:0039007C sub_390000 endp ; sp-analysis failed
debug025:0039007C
debug025:0039007C ; ---------------------------------------
debug025:0039007D db  12h
debug025:0039007E db  2Dh ; -
debug025:0039007F db  82h ; é
debug025:00390080 db 0C9h ; +
debug025:00390081 db  13h
debug025:00390082
debug025:00390082 ; =============== S U B R O U T I N E =
debug025:00390082
debug025:00390082
debug025:00390082 sub_390082 proc near
debug025:00390082
debug025:00390082 var_C= dword ptr -0Ch
debug025:00390082 var_4= dword ptr -4
debug025:00390082 arg_4= dword ptr  8
```

Figure 3.10: Eight-byte Key

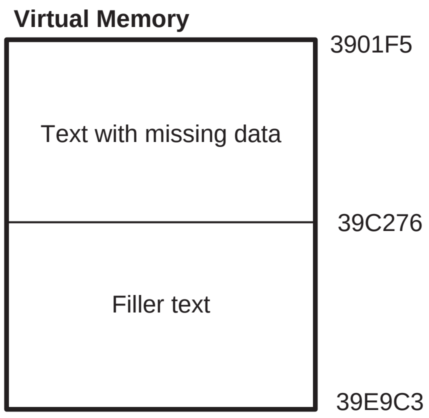


Figure 3.11: Virtual Memory Used by the Second De-Obfuscation Routine

the “filler” text block. This is repeated until all “holes” are filled (see Figure 3.12).

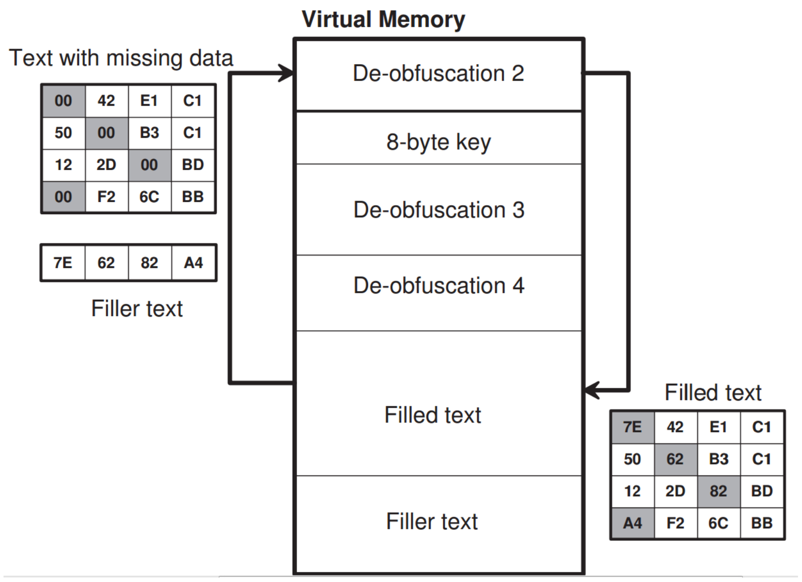


Figure 3.12: Result from the Second De-Obfuscation Routine

The filled text segment turns to be the main outcome of the second de-obfuscation routine. However, this text segment is still not readable and not considered as computer instructions. By utilizing the 8-byte key, the third de-obfuscation routine starts by decrypting the output of the second de-obfuscation. Similar to the first de-obfuscation routine, this routine utilizes the 8-byte key and performs an exclusive-OR (XOR) operation instead of an addition operation. Finally, the fourth de-obfuscation layer contains heavy computations to initialize and prepare some parameters for the rest of the malware operations. It uses the decrypted bytes revealed by the previous routines to modify the rest of the “text/code” segment. After this routine completes, we can observe the real starting point of the Zeus malware.

Even though the “text/code” segment is now valid, the Zeus bot binary employs two additional layers of obfuscation. These two layers are de-obfuscated during the installation procedure. They consist of logical loops that transform arbitrarily long strings into a readable text. The first layer is performed on a set of strings that the malware uses to load the DLL libraries, retrieve function names, and for other purposes during the installation process. Similarly, the second layer is used to decrypt URLs in the static configuration of the configuration file. The main logic of these two routines are described in Algorithm 1 and Algorithm 2.

Algorithm 1 First Routine

```
Input: $seed = 0xBA$,$enc\_string$
Output: $new\_string$
  $new\_string = String(enc\_string.length())$
  for $inti \in Range(0, enc\_string.length())$ do
      $new\_string[i] = (enc\_string[i] + seed)\%256$
      $seed = (seed + 2)$
  end for
  return $new\_string$
```

Algorithm 2 Second Routine

```
Input: $enc\_url$
Output: $new\_url$
  $new\_url = String(enc\_url.length())$
  for $inti \in Range(0, enc\_url.length())$ do
      if $i\%2 == 0$ then
          $new\_url[i] = (enc\_url[i] + 0xF6 - i * 2)\%256$
      else
          $new\_url[i] = (enc\_url[i] + 0x7 + i * 2)\%256$
      end if
  end for
  return $new\_url$
```

- *Bot Installation Process*: After the first four de-obfuscation routines are executed, the malware begins the installation process. The installation process aims at preparing and then launching the malicious activities of the malware. In the following, we explain the main procedure of the installation process.

1. The Zeus malware dynamically loads the `LoadLibrary` and the `GetProcAddress` methods from `Kernel32.dll` library.
2. It decrypts the set of strings, which become DLL methods names, into the virtual memory according to Algorithm 1.
3. The `LoadLibrary` and the `GetProcAddress` methods are then used to load the further methods, as decrypted in step 2, from the Windows DLLs.
4. The Zeus malware enumerates the current process table looking for targeted processes such as the main process name for the Outpost personal firewall application from Agnitum Security `outpost.exe` and the main process name for the personal firewall of the ZoneLabs Internet security `zlclient.exe`. If any of these processes is found, then the Zeus malware aborts the installation process.
5. The Zeus malware appends the path `C:/Windows/System32/sdra64.exe` to `HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows-NT/CurrentVersion/Winlogon/Userinit` registry key. This entry enables the Zeus malware to initiate its installation process again during Windows start-up.

6. Finally, it injects its entire Zeus binary file from the memory address `0x4000-00` to `0x417000` into the virtual memory of `winlogon.exe` process. After that, Zeus passes the control to this process by creating a new user thread, which is immediately executed.

Similarly, the bot uses these steps when the infected machine is restarted. However, there are few steps that are performed only during the initial Zeus installation process. These steps are related to the creation of a local copy of the malware in the infected system for further activities. In the following, we list the main process of creating a local copy of the malware.

(a) The Zeus malware searches for any existing copies of previous Zeus infection files `sdra64.exe`, and then erases them from the infected machine. This behavior would occur when the Zeus binary file is being updated with a newer version of the malware.

(b) It makes an exact copy of itself and then saves it to `C:/Windows/System-32/sdra64.exe`. To evade signature-based detection systems, it appends some randomly generated bytes to the end of the file.

(c) In order to hide itself, the bot duplicates the Modification, Access, and Creation times (MAC times) information from `Ntdll.dll` library, and applies them to the `sdra64.exe`. The intent of this is to make `sdra64.exe` appears to be a system file that has been around since Windows was first installed.

(d) In another level of hiding the created file, it sets the `sdra64.exe` file attributes to system and hidden, so that the user cannot see the file using the standard file explorer.

At this stage, the malware is already injected within the `winlogon.exe` running process. On the other hand, the currently running bot exits and leaves the control to the injected process. However, the installation procedure is continued by the user thread that was started in the `winlogon.exe` process, as described in step 6. From the injection process, we infer that the entire Zeus binary file is copied into the `winlogon.exe` process. Therefore, the injected Zeus instance starts by removing the remaining two layers of the obfuscation by applying Algorithm 1 and Algorithm 2. When the injected malware decrypts all the strings, the Zeus instance employs the piggyback thread technique (to control the infected system through legitimate process) within the `winlogon.exe` process. However, Zeus instances only perform few tasks before they create another thread and exit themselves. This is another attempt by the designers of the Zeus malware to evade detection. Afterwards, the Zeus instance starts injecting itself into another process, namely the `svchost.exe` process. This injected process initiates a communication channel with the C&C server to download the latest updates on the configuration file and the malware itself. Later, the targeted processes get injected with the latest malware payload and then activate the process of stealing information through API hooking techniques. During the malware update process, the following changes were observed on the file system:

1. A new folder is created at the path `C:/Windows/System32/lowsec`. Hiding techniques similar to those that are applied to the file `sdra64.exe` are also applied to the created folder.
2. Two new files, `local.ds` and `user.ds`, are created and placed in the new created folder. The file `user.ds` stores the dynamic configuration file, and the file `local.ds` logs the stolen information until the Zeus malware is ready to send it to the drop location.

The malware that resides on the `winlogon.exe` process acts as the brain for the Zeus malware activities. It communicates and coordinates all the infected processes using the named pipe `_AVIRA_2109`. Table 3.2 shows the list of the commands that are supported by the Zeus malware.

- *Key Extraction*: Zeus botnet uses a configuration file that contains a static information. Specifically, this part of the configuration is stored inside the malware binary file in a specific structure. During the de-obfuscation processes, this structure is recovered and placed in the virtual memory (In our analysis, starting at `0x416000`). All information in the structure is completely de-obfuscated except for two URLs:

Command	Purpose	Return Value
1	Retrieve Zeus version number	4 bytes in a buffer
2	Retrieve name of the botnet	Ascii string in buffer
3	Uninstall Bot	n/a
4	Open the `local.ds` file or create it if it does not exist	n/a
5	Close the `local.ds` file	n/a
6	Open the `user.ds` or create it if it does not exist	n/a
7	Close the `user.ds`	n/a
8	Close the `sdra64.exe`	n/a
9	Open the `sdra64.ex`	n/a
10	Retrieve loader file path	Wide character string
11	Retrieve configuration file path	Wide character string
12	Retrieve log file path	Wide character string
13	Crash the `winlogon` process intentionally	n/a

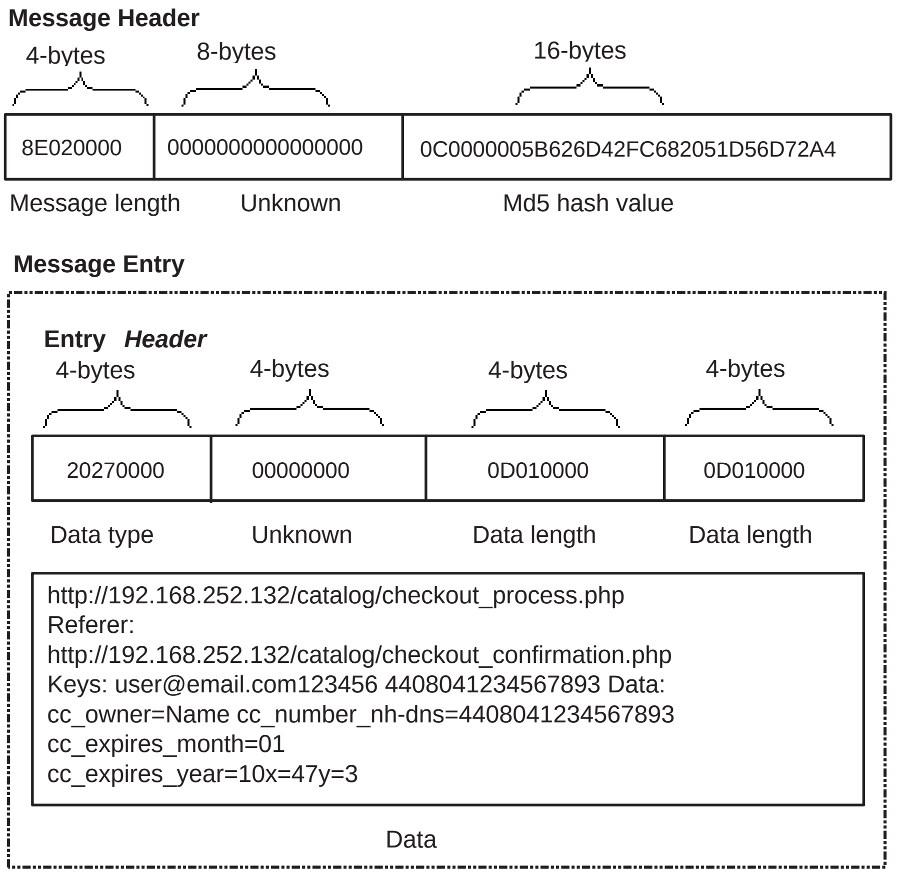
Table 3.2: List of the Zeus malware commands

`url_compip` and `url_config`. These URLs can be de-obfuscated using Algorithm 2. The URL `url_compip` is the web location to determine the IP address of the infected host, and the `url_config` is the web location to download the configuration file for the botnet. The static configuration structure also contains an RC4 substitution table that is generated by the encryption key specified in the configuration file. Throughout our analysis, we noticed that the substitution table was generated by the RC4s key-scheduling algorithm and then we verified that the encryption employed by Zeus is done by the RC4 algorithm. The recovered static configuration can be used in different ways to gain some control over the botnet. The most valuable piece of information is the substitution table, which can be used to decrypt all the communications of the Zeus botnet. Moreover, it can be used to decrypt the configuration file as well as the stolen information. In order to recover the static configuration structure described above, we have to go through all the de-obfuscation phases. This requires executing the malware until it finishes all the de-obfuscation layers. Emulation techniques are considered as a safe and fast procedures to achieve our goals. Using Python scripting language along with the `IDAPython` plugin [6], we were able to emulate all the de-obfuscation routines and extract the substitution table from the static configuration structure. These extracted keys allows decrypting the botnet communication traffic and all the encrypted files. Similarly, it allows us to extract any information from the static configuration structure, such as the URLs for any future updates, which point to the C&C servers. Our experimental results show that any sub-version of Zeus (v.1.2.x.x) can be fully analyzed using our methodology because it holds the same logical blocks.

Packet Decryption

After extracting the RC4 encryption key, we used it to decrypt the botnet communications. By decrypting the transmitted HTTP payload, we are able to uncover the structure of the messages between the bot and the C&C server. We analyzed the structure of the HTTP POST messages (`POST /gate.php`) which carries all the updates and reports from the bots to the C&C server. Each bot posts a variable number of encrypted bytes based on the sent data to the C&C server in a specific structure. The payload is encrypted using an RC4 encryption algorithm only. As depicted in Figure 3.13, we restore the structure of the messages as follows:

1. Each message starts with a header that consists of 28-bytes. This header contains an MD5 hash value for the rest of the message.



```
http://192.168.252.132/catalog/checkout_process.php
Referer:
http://192.168.252.132/catalog/checkout_confirmation.php
Keys: user@email.com123456 4408041234567893 Data:
cc_owner=Name cc_number_nh-dns=4408041234567893
cc_expires_month=01
cc_expires_year=10x=47y=3
```

Figure 3.13: A Decrypted Sample Message

2. As shown in Figure 3.13, the rest of the message follows in the form of repeated data blocks where each block consists of:

   (a) An entry header with 16-bytes that contains information about the current data entry. The first 4-bytes serve as the type of the reported information, which can be recognized by the bot and the control panel. The third 4-bytes determine the length of the carried information.

   (b) A variable number of bytes that is specified in the entry header. These bytes represent one piece of the information that is transmitted within this packet.

It should be noted that the encrypted communication of the Zeus botnet is vulnerable to the RC4 key-stream re-use attack because there is no Initialization Vector (IV) setup in every session, i.e., the same RC4 key-stream is reused to encrypt all messages.

3.4 Conclusion

The aforementioned case studies have been highly considered by malware researchers at the National Cyber-Forensics and Training Alliance (NCFTA) Canada as well as academic and industry researchers worldwide. The analysis done in both case studies set a knowledge to a malware research initiative leading to other case studies done by Rahimian *et al.* [171] and Andriesse *et al.* [21]. The former has shown that Citadel botnet is an advanced version of Zeus botnet, the latter addressed a P2P variant of Zeus, namely, GameOver Zeus. In addition, other works have been conducted at Concordia University, e.g., works done on malware authorship attribution [16], fingerprinting re-usability of malware functions [17] and the identification of malware binaries creation process [170]. Based on the different findings illustrated in this chapter, showing how threats like Mariposa and Zeus can be innovative, it is of paramount importance to investigate who potentially pulls the strings behind elaborated threats. As such, we decide in the next chapter, to look at the cyber-threat infrastructures used to orchestrate botnets and advanced threats activities. We discuss an investigation done on one year malware dynamic analysis reports. We will also describe the different findings related to cyber-threat infrastructures.

Chapter 4

Cyber-Threat Infrastructures

4.1 Overview

In this chapter, we present a study to investigate cyber-threats and the underlying infrastructures. More precisely, we detect and analyze cyber-threat infrastructures for the purpose of unveiling key players (owners, domains, IP addresses, organizations, malware families, etc.) and the relationships between these players. To this end, we propose metrics to measure the badness of different infrastructure elements using graph theoretic concepts such as centrality concepts and Google PageRank. In addition, we quantify the sharing of infrastructure elements among different malware samples and families to unveil potential groups that are behind specific attacks. Moreover, we study the evolution of cyber-threat infrastructures over time to infer patterns of cyber-criminal activities. The proposed study provides the capability to derive insights and intelligence about cyber-threat infrastructures. Using one year dataset, we generate important results regarding emerging threats and campaigns, important players behind threats, linkages between cyber-threat infrastructure elements, patterns of cyber-crimes, etc. The remainder of this chapter is organized as follows. In Section 4.2, we describe our approach to investigate cyber-threat infrastructures. In Section 4.3, we provide statistics and insights generated from the analysis of cyber-threat infrastructures. Finally, we conclude with a discussion and future works in Section 4.4.

4.2 Approach

In this section, we present a framework to collect insights and intelligence out of dynamic malware analysis. Malware samples tend to exhibit a cooperative strategy with remote malicious domains and IPs to perpetrate malicious activities, e.g., stealing credentials, spams propagation, advanced DDoS attacks, etc. In the light of these facts, we aim to design and integrate an approach to generate cyber-threat intelligence for the purpose of identifying the infrastructures used by malware to threaten the cyber-space. Our approach to generate cyber-threat intelligence is depicted in Figure 4.1. The approach falls into: (1) data collection, (2) cyber-threat graph generation, (3) descriptive statistics, (4) badness scoring and (5) patterns inference.

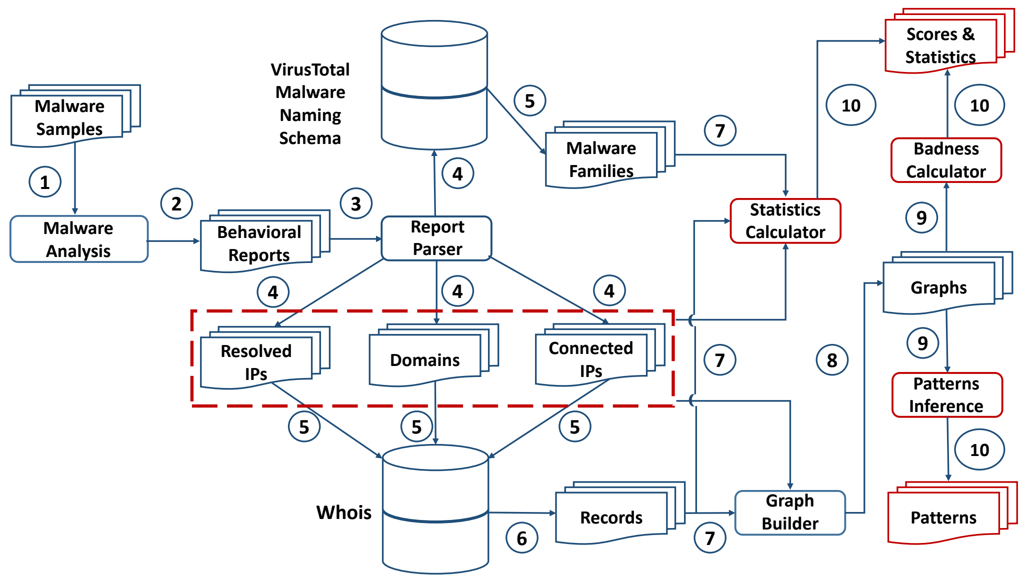


Figure 4.1: Approach Overview

4.2.1 Data Collection

We collect malware samples on a daily basis from a trusted third party. These malware samples are analyzed through a sandbox technology to monitor malware behavior on either physical or virtual machines. The malware behavior is stored in XML reports. We usually manage to get an average of $45,000$ malware reports per day. For each report, we extract the domains visited by malware samples, IPs resolving to these domains and IPs directly connected by malware through FTP, SMTP, IRC servers as well as plain UDP and TCP connections. Meanwhile, we use VirusTotal [211] malware naming schema to get malware family information out of 54 anti-virus engines. In addition, we use Whois database [220] to get domains and IP records. The intent is to gather domains' owners, administrative and technical support people, organizations, registrars, physical addresses, network names and name-servers.

4.2.2 Cyber-Threat Graph Generation

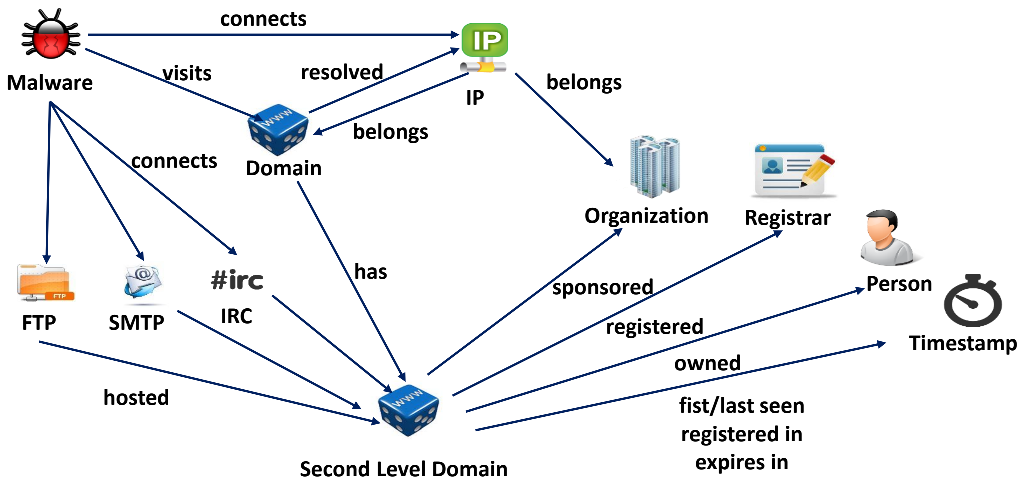


Figure 4.2: Cyber-threat Infrastructure Schema

Based on the data collected from dynamic malware analysis and Whois database, we define a cyber-threat infrastructure as the set of entities involved in malware activities (Figure 4.2). The different components that constitute a cyber-threat infrastructure are: malware, domains, IP addresses, FTP servers, SMTP servers, IRC channels, timestamps, organizations, registrars, technical people, administrative people and domain owners. The interaction between the infrastructure components is as follows: A malware tends to visit domains, which can be command and control servers (C&Cs) or re-directions of legitimate domains to malicious proxies or C&Cs. These domains are resolved to IP addresses. They also usually have second-level domains. On another hand, malware can connect to FTP servers to upload stolen information or download other malware binaries. Malware can also connect to SMTP servers to conduct spamming activities or IRC channels to interact with IRC botnets. They can also connect directly through non-conventional TCP and UDP protocols for the purpose of cooperating with infected machines or C&Cs. FT-P/SMTP servers and IRC channels can be hosted within a second-level domain server. The latter is registered within a registrar and sponsored by an organization. It can have an administrative contact, a technical support contact and an owner. Each domain or a second-level domain has a creation timestamp, expiration timestamp and passive DNS first/last seen timestamps. An organization can have more than one IP block and be located in different countries.

We represent cyber-threat infrastructures as a complex network of *directed graphs*. The vertices of the graph represent components of cyber-threat infrastructures, i.e., malware, domains, IPs, FTP/SMTP servers, IRC channels, organizations, registrars, technical/administrative people, domain owners and physical addresses. Collection timestamps are properties of malware nodes, whereas first/last seen timestamps, creation and expiration timestamps are properties of second-level domains.

Figure 4.3 illustrates a directed graph representing a cyber-threat infrastructure. The red vertices represent malware samples connecting to domains (blue vertices). Both of these domains resolve to the same IP address (yellow vertex). Owners, organizations and registrars are represented by green vertices.

The increasing number of vertices appearing in cyber-threat infrastructures make their analysis a complex task. Figure 4.4 depicts the evolution of five days cyber-threat

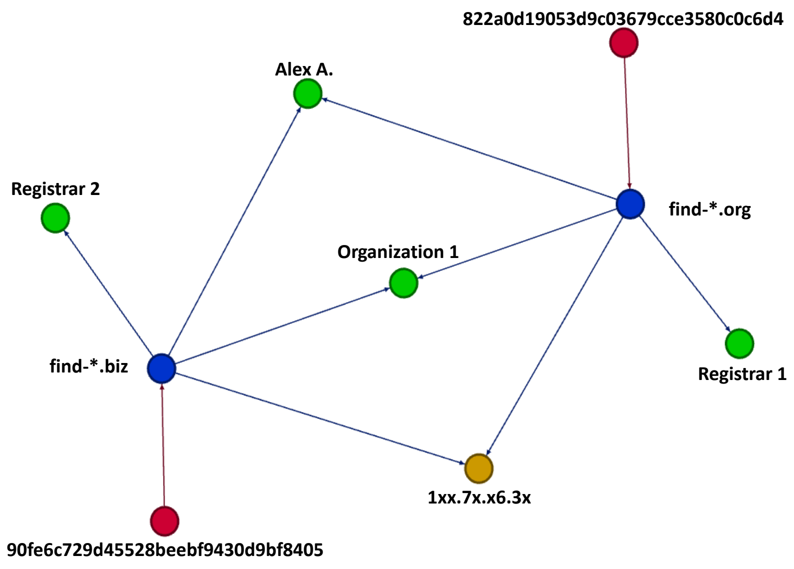


Figure 4.3: Example of a Cyber-threat Infrastructure

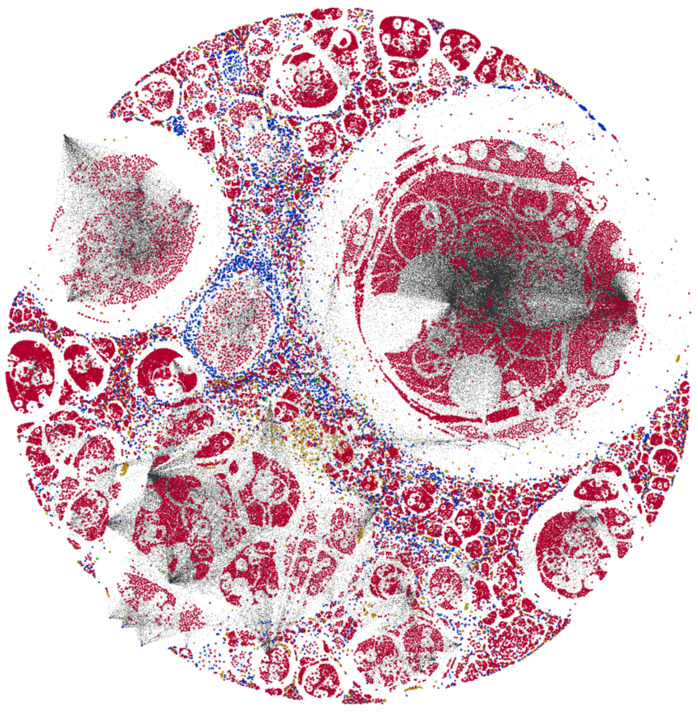


Figure 4.4: Components representing Cyber-threat Infrastructures

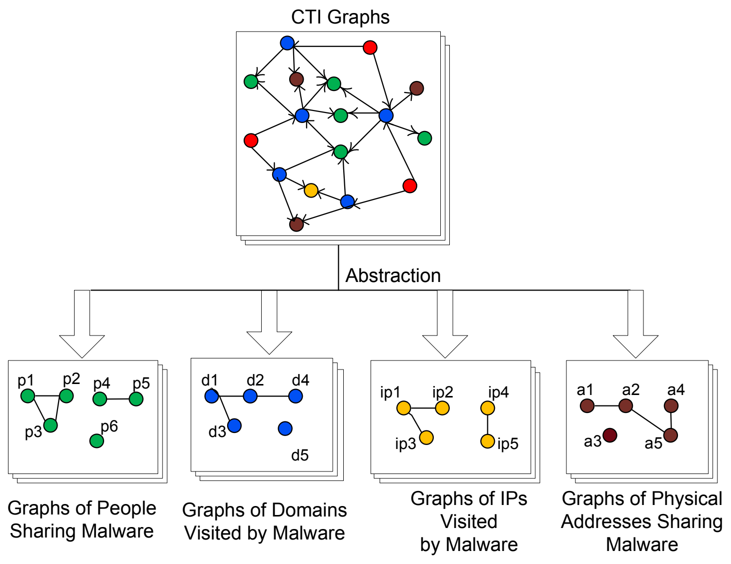

CTI Graphs
Abstraction
p1
p2
p4
p5
p3
p6
d1
d2
d4
d3
d5
ip1
ip2
ip4
ip3
ip5
a1
a2
a4
a3
a5
Graphs of People Sharing Malware
Graphs of Domains Visited by Malware
Graphs of IPs Visited by Malware
Graphs of Physical Addresses Sharing Malware

Figure 4.5: Abstraction

infrastructures. To overcome the complexity of cyber-threat graphs, we use a graph abstraction technique, where we decompose heterogeneous directed graphs (vertices representing many types) into homogenous weighted graphs. To illustrate the abstraction, we consider the case of malware samples sharing two domains. Initially, each vertex v_i, representing a malware sample, is linked to two vertices, v_{d1} and v_{d2}, representing visited domains. This sub-graph is abstracted to two linked vertices v_{d1} and v_{d2}, representing domains. The edge between v_{d1} and v_{d2} is labeled with the number of malware shared by these domains. Figure 4.5 depicts the abstraction of cyber-threat infrastructure graphs. By performing abstraction, we create the following sub-graphs: (1) *Domain-Malware graph*: Domains are linked if they are visited by shared malware samples. (2) *Domain-IP graph*: Domains are linked if they resolve to shared IPs. (3) *IP-Malware graph*: IPs are linked if shared malware samples connect to. (4) *Owner-Malware graph*: Owners are linked if they own domains that are visited by shared malware samples. (5) *Owner-Physical address graph*: Owners are linked if they register different domains with the same physical addresses. (6) *Organization-Malware graph*: Organizations are linked if they have IPs connected by shared malware samples.

4.2.3 Badness Scoring

In this research effort, we put an emphasis on finding *what are the key players in cyber-threat infrastructures*. The importance of vertices in a network graph is known as *vertex's centrality*. The latter represents a real-valued function produced to provide a ranking, which identifies the most important nodes [37]. Despite the fact that different centralities, namely, degree centrality [177], closeness centrality [51, 199], betweenness centrality [75], and Eigen-vector centrality [36], are widely used in the analysis of different social networks, we are mostly interested in evaluating the importance or influence of different characters in cyber-threat infrastructures. For this purpose, some algorithms have been defined, such as, Hypertext Induced Topic Search (HITS) algorithm [106] and Google's PageRank algorithm [42]. In our approach, we adopt Google's PageRank algorithm due to its efficiency, feasibility, less query time cost, and less susceptibility to localized links [79]. In the sequel, we briefly introduce the PageRank algorithm and the random-surfer model.

Definition 4.2.1 *(PageRank). Let $I(v_i)$ be the set of vertices that link to a vertex v_i and let $deg_{out}(v_i)$ be the out-degree centrality of a vertex v_i. The PageRank of a vertex v_i,* *denoted by $PR(v_i)$, is provided in Equation 5.1:*

$$PR(v_i) = d\left[\sum_{v_j \in I(v_i)} \frac{PR(v_j)}{deg_{out}(v_i)}\right] + (1-d)\frac{1}{|D|} \tag{4.1}$$

In the aforementioned formula, the constant d is called *damping factor*. Its value is generally assumed to be set to 0.85 [42]. From Equation 5.1, we can have one equation per vertex v_i with an equal number of unknown $PR(v_i)$ values. Assuming that the PageRank values $PR(v_i)$ sum up to 1 ($sum_{i=1}^{n} PR(v_i) = 1$), then the PageRank algorithm tries to find out iteratively different PageRank values. This algorithm has been developed intuitively considering a user surfing the Web, starting from a web page and randomly visiting another web page through a link. If the user is on page v_j with a probability d (damping factor), then the probability for this user to visit another page v_i is equal to $\frac{1}{deg_{out}(v_j)}$. With a probability of $1-d$, the user will stop following links and pick another random page in V. Since the web-surfing process shows randomness, the authors of the PageRank algorithm claim that the PageRank values can be computed through a stochastic process. Thus, a stochastic transition matrix W is defined. The vertices ranking values are computed as expressed in Equation 5.2:

$$\vec{PR} = d\left[W.\vec{PR}\right] + (1-d)\frac{1}{|D|}\vec{1} \tag{4.2}$$

The stochastic matrix W is defined as follows:

$w_{ij} = \frac{1}{deg_{out}(v_j)}$ if a vertex v_j is linked to v_i

$w_{ij} = 0$ otherwise

The notation $\vec{R}$ stands for a vector where its i_{th} element is $PR(v_i)$ (PageRank of v_i). The notation $\vec{1}$ stands for a vector having all elements equal to 1. The computation of PageRank values is done iteratively by defining a convergence stopping criterion ϵ. At each computation step t, a new vector $(\vec{PR}, t)$ is generated based on previous vector values $(\vec{PR}, t-1)$. The algorithm stops computing values when the condition $|(\vec{PR}, t) - (\vec{PR}, t-1)| < \epsilon$ is satisfied. In our case, since graphs are abstracted to weighted undirected graphs, the out-degree centrality of a vertex v_i is similar to the degree centrality. However, the weights of edges for each vertex are normalized with values between 0 and 1. The definition of the stochastic matrix W is slightly changed to:

$w_{ij} = e_{ij} \times \frac{1}{deg_{out}(v_j)}$ if a vertex v_j is linked to v_i

$w_{ij} = 0$ otherwise

e_{ij}: edge (v_i,v_j) (normalized weight value)

The reason behind using PageRank algorithm to compute badness of vertices, lies in: (1) Scores are computed through a stochastic approach, which reflects randomness in the evolution of a model. With respect to cyber-threat infrastructures, we assume that there exists a random evolution, on a daily basis, in the appearance of malware samples, domains, IPs, servers, organizations, owners and registrars. Such appearance of new vertices impacts the evolution of badness scores. (2) The random web-surfer model illustrates how web pages can be accessed with a probability value (damping factor). In analogy with cyber-threat infrastructures, the probabilistic approach is interesting since it reflects potential actions done through infected machines: A malicious domain can be visited through an infected machine, an IP address can be connected by infected machines or resolved to a malicious domain or a server, an FTP server can be used to upload stolen information, an SMTP server can be used to launch spam or phishing campaigns, an IRC channel can be used to instruct bots to launch DDoS attacks, malware propagation or other malicious activities.

4.2.4 Patterns Inference

Here, we aim to closely study *how cyber-threat infrastructures evolve over time*. To this end, we target the identification of discernible regularities and irregularities in such infrastructures by isolating observable patterns in the generated graphs. In cyber-threat infrastructures, a pattern is associated with possible relationships between domains, IPs, owners and organizations. To infer patterns, we compute similarities between graphs collected on a daily basis.

Computation of similarity between graphs is a challenging task especially when dealing with large-scale evolving graphs. To overcome this challenge, we resort to the so-called graph kernels [77, 172, 215]. A graph kernel is a function that computes the similarity between graphs using linear methods. However, for large-scale graphs, graph

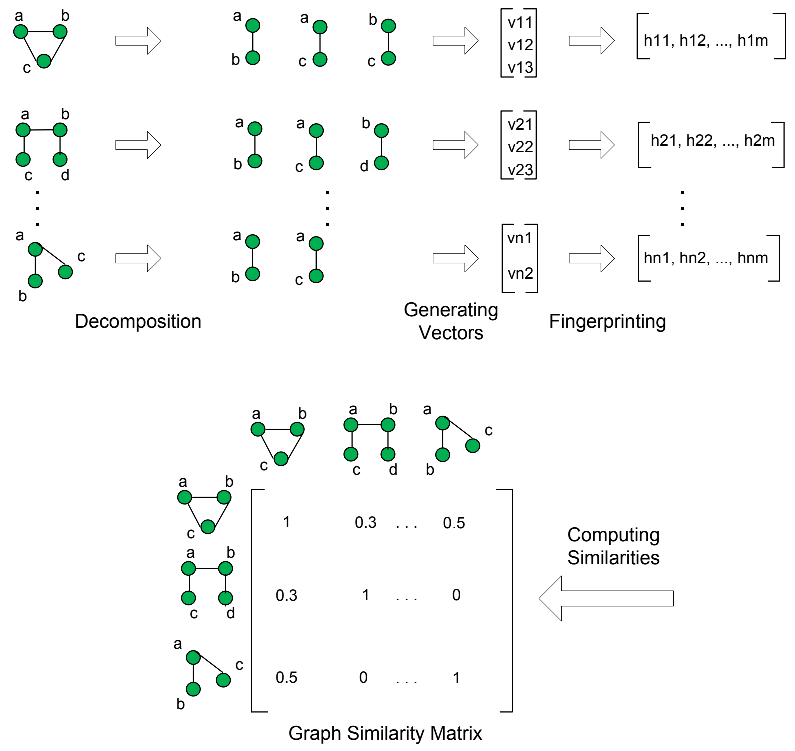


Figure 4.6: Fingerprinting Approach

kernel methods generate vectors with high dimensions that are not easy to handle. To address this issue, graph kernel methods require an important process known as *fingerprinting* [172]. The latter consists of producing compact representations, known as signatures (or fingerprints), for graph structures based on the generated vectors. Graph similarities are then computed using these fingerprints. To generate graph fingerprints, we use the *min-hashing* technique. Our approach of computing graph similarities is inspired by the work done by Teixeira *et al.* [208] who introduced a fingerprinting technique for graph kernels based on min-hashing. In the sequel, we introduce our methodology used to observe the presence of patterns and how they are inferred. First, we present the different steps to compute similarities between graphs, as illustrated in Figure 4.6. These steps fall into: (1) decomposition, (2) vector generation, and (3) fingerprinting.

Decomposition

During this step, we decompose each graph, obtained from the abstraction process, into a set of substructures. These substructures may be obtained based on paths, cycles, trees, etc. In our approach, we decompose graphs based on paths between graph vertices. In other words, two vertices v_1 and v_2 form a substructure if there is an edge between v_1 and v_2.

Vector Generation

This step consists of mapping graph substructures into vectors. Algorithm 3 implements the vectors' generation process. It takes, as input, a set of substructures $S(G)$, obtained from decomposing a graph G. For each substructure, we convert it into an integer value using the formula in line 3, where α is a random number, P is a big prime number such that $0 < \alpha < P$, and $\mathcal{L}$ is the labeling function that returns the number of shared malware between v_i and v_j.

Algorithm 3 Vector Generation

```
Input: Graph $G$,Set of substructures $S(G)$
Output: $Vector\ V$
  $i := 1$
  for $Substructure(v_i, v_j) \in S(G)$ do
      $x := (\alpha\ \mathcal{L}(v_i, v_j))(mod\ P)$
      $V[i] := x$
      $i++$
  end for
  return $V$
```

Fingerprinting

The vectors generated from the graph substructures might have high dimensionality especially for large-scale graphs. To reduce the dimensionality of a vector, we use a technique, known as fingerprinting, to produce a compact representation that is easy to handle. To this end, we adopt the ideas presented in [208] to fingerprint graphs based on min-hashing. The fingerprinting method is presented in Algorithm 4. It takes, as input, a vector generated from the substructures of a graph G and a set of m hash functions $h_1, h_2, ..., h_m$. It produces, as output, a fingerprint of graph G, which is a compact representation of the input vector. The fingerprint consists of a vector of m min-hash values, where m is the number of hash functions.

Algorithm 4 Fingerprinting

```
Input: Vector $V$ of a graph $G$
Input: Set of hash functions $H$: $h_1, h_2, ..., h_m$
Output: Graph fingerprint $F$
  for $i := 1; i <= |H|; i++$ do
      $F[i] := min(h_i(V))$
  end for
  return $F$
```

Graph Similarities Computation

Graph similarities are represented through a matrix, where each value is proportional to the number of values, in the min-hash vectors, that are shared between graph pairs (Figure 4.6). This matrix is important to group graphs with proportional similarities into groups that can be good candidates to detect patterns between corresponding min-hash values. The graph similarity matrix provides a big picture of the evolution of cyber-threat infrastructures over time. However, it needs to be leveraged to extract patterns effectively. To this end, we propose an algorithm to infer patterns as explained hereafter.

Pattern Time-Based Inference

To extract patterns, we elaborate a time-based inference algorithm (Algorithm 5). This algorithm takes, as inputs, a similarity matrix, an analysis period (in terms of days), a time window (usually one day), and a density threshold to filter days where we have low similarities in the matrix. The algorithm collects common patterns, by sliding the time window through the analysis period, and checks if the similarity value is higher or equal to the density threshold. If so, we compute the intersection between patterns found on days representing the row and column index in the similarity matrix. The collected patterns are stored in a list structure that we sort at the end to collect the most or less occurring patterns.

Algorithm 5 Pattern Inference

```
Input: Simlarity Matrix $M$,Analysis Period $t$,Time Window $w$,Threshold $th$
Output: List of patterns $P$
  for $i := 1; i <= t; i$++ do
      for $j := i + 1; j <= i + w; j + +$ do
          if $M[i][j] >= th$ then
              $I := Patterns[i] \cap Patterns[j]$
              $P$.append($I$)
              clear($I$)
          end if
      end for
      sort($P$)
  end for
  return $P$
```

4.3 Experimental Results

In the sequel, we present the results of our analysis of cyber-threat infrastructures. The results include descriptive statistics, badness ranking and patterns inference. It is important to mention that due to the sensitivity of the collected data, we have anonymized domains, IPs, organizations and owners.

4.3.1 Dataset Description

Our dataset consists of one year malware data collected from 25^{th} August 2013 to 25^{th} August 2014. Table 4.1 presents some statistics regarding the collected data. It is important to mention that the domains are filtered and do not contain legitimate domains that are listed in the top one million Alexa white-list domains [1].

Collected Data	Statistics
Malware samples	$4,717,628$
Domains	$9,303,378$
Second-level domains	$151,757$
IPs that domains resolve to	$240,174$
IPs that malware connect to	$118,270$
Domain Whois records	$110,414$
IP Whois records	$287,005$

Table 4.1: Statistics of the Dataset

4.3.2 Descriptive Statistics

In this section, we present some descriptive statistics that we generated from our analysis of cyber-threat infrastructures.

Domains & Resolving IPs

2^{nd} **Level Domain**	**# Malware**
*il.ru	$252,358$
*entre.ru	$194,749$
*soft.com	$190,327$
*update.com	$166,995$
*admr.com	$160,123$
cloud*.net	$137,883$
*lytics.com	$119,233$
*box.net	$113,619$
*host2.com	$110,373$
*tal.com	$106,817$

Table 4.2: Domains vs. Number of Malware

Table 4.2 lists the top-10 most visited second-level domains by malware. We notice that five out of ten of these domains are legitimate. This can be explained by the fact that malware samples tend to test connectivity by visiting legitimate domains or redirecting access to legitimate domains to fake web-pages. Malware also connect to legitimate domains to download vulnerable patches of operating systems or software to exploit vulnerabilities and perpetrate malicious activities.

Domain	**# IPs**
j.nb*.com	$23,021$
ip*.33*.org	$10,779$
f.nb*.com	$10,313$
i.nb*.com	$7,130$
g.nb*.com	$5,825$
*sopuli.*to.org	$4,300$
h.nb*.com	$4,232$
e.nb*.com	$3,963$
router.bi*.com	$3,573$
*lytics.com	$3,342$

Table 4.3: Domain vs. Number of Resolving IPs

Table 4.3 lists the top-10 most fast-fluxing domains (domains that resolve to many IPs). We observe the presence of domains that have the same second-level domain, *nb*.com*, and resolve to many IP addresses. One main observation from the aforementioned table is that fast flux and dynamic generated domain names are still widely used by malware.

IP	**# Domains**
184.1xx.xxx.x6	$171,388$
199.xxx.xx.xx0	$125,454$
184.x7x.xxx.xx5	$90,766$
184.x7x.xxx.xx0	$84,296$
184.x7x.xxx.xx8	$82,104$
216.2xx.xxx.x5	$21,402$
216.2xx.xxx.x1	$20,521$
46.xx.xxx.x0	$13,606$
162.xxx.x.xx4	$7,410$
204.xx.xxx.x7	$7,178$

Table 4.4: Resolving IPs vs. Number of Domains

Table 4.4 illustrates the most shared resolving IPs between domains. We observe the presence of resolving IPs belonging to the same IP space. There are two IP spaces containing many resolving IP addresses. These IPs are listed in blue color. The presence of common IP spaces implicitly infers that cyber-criminals are prone to use an IP infrastructure to perpetrate malicious activities, or infect vulnerable IP spaces to let them be part of their botnets. The IP address listed in red color represents the most resolved IP. By tracking back associated domains, we observe that it resolves to domains dynamically generated and belonging to the same second-level domain. The domain generator associated with this IP generates a set of letters and digits. The length of the chars sequence is 30. The second-level domain is **eker.com*. Malware families associated with the top listed IP are: *antiav*, *barys*, *graftor*, *injector*, *ramnit*, *sality*, *slugin*, *swisyn*, *symmi*, *vbinject*, *virut* and *zusy*.

Network	**# IPs**
ORG1(CHINA)-GD	12,880
ORG1(CHINA)-JS	7,245
ORG2(CHINA)-SD	5,113
ORG2(CHINA)-HA	3,745
ORG2(CHINA)-HE	3,597
ORG1(USA)-2011L	3,255
ORG1(CHINA)-SC	3,232
ORG1(CHINA)-FJ	3,001
ORG1(CHINA)-HB	2,844
ORG2(CHINA)-LN	2,815

Table 4.5: Network Name vs. Number of Resolving IPs

Table 4.5 lists the number of resolving IPs per network name belonging to organizations. We observe that China has the highest number of resolving IPs since 9 out of 10 network names spread throughout different Chinese regions. There is only one American network name that is present in the top-10 of network names containing resolving IPs.

Connected IPs

Table 4.6 lists the number of malware that connect to IPs. Connected IPs are directly accessed by malware through conventional protocols (e.g., FTP, IRC, and SMTP) and unconventional TCP and UDP ports. In contrast to resolving IPs, connected IPs are spread through many IP spaces. We observe that all the listed IPs are from different IP spaces.

Domain	**# Malware**
93.xxx.xx.xx0	$11,553$
65.xx.xx.xx7	$4,497$
219.xxx.x.xx7	$4,097$
113.xx.xxx.xx6	$3,223$
95.xxx.xx.xx3	$2,719$
124.xxx.xxx.6	$2,609$
147.xxx.xxx.x7	$2,498$
69.xxx.xx.x0	$2,429$
89.xxx.xx.xx4	$2,253$
125.xx.xxx.x4	$2,139$

Table 4.6: Connected IP vs. Number of Malware

This can be explained by the absence of fast-fluxing. IPs are dedicated to be a depot of stolen information, a spamming server, IRC channel, or a nest of other malware samples ready to be downloaded. The top listed connected IP has been associated with a VPN anonymity service. We observe a high interaction with this IP and the following malware families:*antiav*, *barys*, *esfury*, *hype*, *injector*, *navipromo*, *pirminay*, *ramnit*, *slugin*, *swisyn*, *symmi*, *vbinject*, *virut*, *vundo*, and *zbot*.

Network	**# IPs**
ORG1(MALAYSIA)-HSDPA	$10,093$
ORG1(INDIA)-SouthZone	$2,176$
ORG1(CHINA)-GD	$1,089$
ORG1(KOREA)	837
ORG1(USA)-2011L	739
ORG3(CHINA)	681
ORG1(CHINA)-JS	604
ORG2(KOREA)	472
ORG4(CHINA)	358
ORG2(USA)	309

Table 4.7: Network Name vs. Number of Connected IPs

Table 4.7 illustrates the number of connected IPs within top-10 network names. We observe that 8 out of 10 network names are located in Asian countries. The number of organizations is 9, among which 3 are Chinese organizations, 2 are Korean and American organizations. However, top ranking network names belong to a Malaysian and an Indian organization respectively. The Malaysian network name is associated with 3 malware samples, namely, a variant of *zlob*, a variant of *zbot*, and a variant of *proxyTroj*. The latter is a proxy Trojan, which infects computers to play the role of a Command & Control and bot at the same time. Usually, such malware variants tend to communicate and cooperate through infected machines.

Whois Information

Registrant	# Domains
Registration private	2,983
Whoisguard protected	744
Domain administrator	632
Domain admin	451
Whois Agent	378
Perfect Privacy LLC	274
E*l Y.	187
Whois Privacy Protection Service	184
Private Registrant	163
Oneandone Private Registration	123
Spy Eye	120
This domain for sale toll free: *-822-*	104
DNS Admin	92
Reactivation Period	92
Domain Manager	75

Table 4.8: Registrant vs. Number of Domains

Table 4.8 lists the different registrants and the corresponding number of domains. We notice that people behind suspicious domains use privacy services to protect their identities. Thus, there exists a big number of domains that share the same private registrants. In the aforementioned table, 12 out of 15 registrants are protected by privacy services and one registrant has a regular name (*E*l Y.*). We observe also the presence of a registrant with the name of a well-known malware family, namely, *Spy Eye* [194]. The domains registered with *Spy Eye* have been visited by 830 malware samples, mainly belonging to the following malware families: *conjar*, *fareit*, *nebuler*, *zbot* and *zusy* (*zbot* variant). All these families are Trojan bots involved in password stealing, downloading other malware samples, modifying system files and registry, adding startup items to systems, etc. We also notice that there are 104 domains registered with a message (*This domain for sale toll free: *-822-**). This phenomenon is known as domains parking, where blackhat Search Engine Optimization (SEO) people are used to infect machines with malware samples to contact these domains and make them visible for different search engines. The more a domain is visible, it is easier to sell it.

Address	**# Domains**
P.O. Box ∗ ∗ ∗ ∗ − ∗ ∗ ∗ ∗ ∗∗ Panama	708
∗ ∗ ∗ ∗ ∗ Northsight blvd PMB**9 USA	379
∗ ∗ ∗ ∗ ∗ Gran bay parkway w. USA	272
∗ ∗ ∗ ∗ ∗ P.O. Box ∗∗ Beach Australia	228
∗ ∗ ∗ ∗ ∗ Memorial Dr. #935 USA	186
Ilyinka Street ∗∗ Russia	120
∗ ∗ ∗ ∗ ∗ 24^{th} Street USA	115
∗ ∗ ∗ Lee Road Suite ∗ ∗ 0 USA	108
∗ ∗ ∗ Main street #∗ ∗ 9 USA	108
∗ ∗ − ∗ ∗ Boulevard Massena France	87

Table 4.9: Physical Address vs. Number of Registered Domains

Table 4.9 lists the different physical addresses associated with registered second-level domains. We notice that 708 second-level domains are registered with an address located in Panama. This address corresponds to a privacy service that hides relevant registrants' information. This service is prone to suspicious activities since it serves spamming domains, websites of companies involved in robot-calls and scam abuses.

4.3.3 Badness Ranking

One of the goals of this research effort is to quantify the badness of cyber-threat infrastructures' elements. We use the badness metric on a daily basis to monitor the aliveness of badness for different IPs, domains, owners or organizations. In the sequel, we provide a description of different observations found on the computation of badness scores.

Domain	**Avg Score** $\times 10^3$
*entre.ru	5.1194617836
*box.net	3.50800756525
*spectr.ru	2.89366998268
*file.ru	2.33333105403
*sung.ru	2.17069956058
*express.ru	2.0137619806
*ldr.ru	1.6519902951
*.elb.*aws.com	1.43752913358
d1sx0cjuasqkw9.*ront.net	1.41139045489
*pro.ru	1.40483436327

Table 4.10: Top-10 Domains Badness Scores

Table 4.10 illustrates the top-10 average badness scores for domains observed on one year. We notice that 7 out of 10 domains have "*.ru*" extension. One of the obtained domains is dynamically generated. An interesting fact is that 5 out of 7 "*.ru*" domains are registered with the same information (registrant is known as “Private Person” and the same name servers). The number of associated malware is 830. In addition, these domains share a lot of malware families, spanning over bot Trojans and bitcoiners, mainly *badur*, *bitcoinminer*, *graftor*, *kryptik* (cryptolocker), *loadmoney*, *minggy*, *strictor*, *symmi*, and *zusy* (zbot variant). We suspect that people belonging to the same criminal group use the same registrant information and are behind bitcoining campaigns and botnet activities.

IP	**Avg Score** $\times 10^3$
93.xxx.xx.xx0	6.90947359832
46.xxx.xxx.xx9	5.76183602875
113.xx.xxx.xx6	5.1836928519
220.xxx.xxx.7	4.53174275775
125.xx.xxx.x4	4.52513888983
124.xxx.xxx.x1	4.51910871828
221.xxx.xxx.x8	4.31768525507
91.xxx.xx.x0	3.06311160603
239.255.255.250	2.88498391036
89.xxx.xx.xx4	2.5103999827

Table 4.11: Top-10 Connected IPs Badness Scores

Table 4.11 illustrates the top-10 average badness scores for connected IPs. The top ranked badness IP is the same leading IP “93.xxx.xx.xx0”, observed in Table 4.6. The same observation can be done on IPs “113.xx.xxx.xx6” and “125.xx.xxx.x4”, which are present in both Tables. The reason is that these IPs have maintained a badness score throughout the whole year, whereas other IPs, listed in Table 4.6, have not maintained their badness score as much as IPs listed in Table 4.11. It is important to notice the presence of the IP “239.255.255.250”, which is SSDP multicast reserved IP. This IP is mainly used by what are called Universal Plug and Play (UP&P) malware families, e.g., *conficker*, *downadup* and their variants. These malware infect other machines through vulnerabilities found in Windows server services (e.g., RPC Handling Remote Code Execution Vulnerability). In [14, 133], the authors illustrate how UP&P devices can be used as an infection vector through SSDP protocol. Such vector of infection is still active since we observe a lot of new variants connecting to “239.255.255.250” IP address.

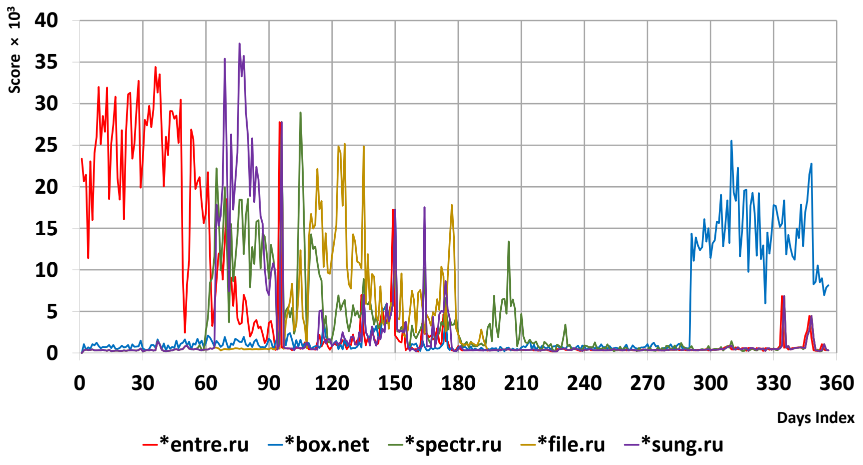


Figure 4.7: Top-5 Domains Score Trend

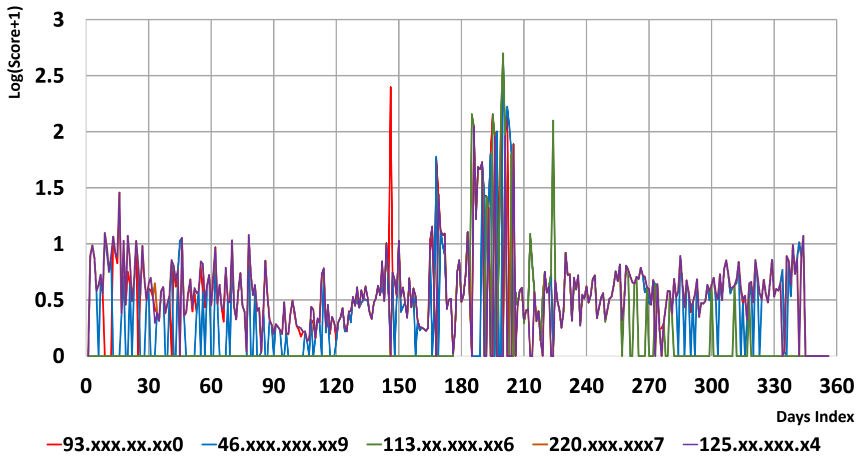


Figure 4.8: Top-5 IPs Score Trend

In Figures 4.7 and 4.8, we present the evolution of the badness scores for the top-5 domains and connected IPs. We observe that the badness of domains has a *periodic badness persistence*. For instance, the domain "**entre.ru*" had high badness scores during the first 60 days and the domain "**box.net*" had high badness scores during the last 70 days. Similarly, the domain "**spectr.ru*" had high badness scores during a period of 40 days, the domain "**file.ru*" had high badness scores during a period of 75 days, and the domain "**sung.ru*" had high badness scores during 30 days. However, we can observe some sporadic changes in scores for all the domains (spikes after observing low badness ranking). For instance, the domain "**entre.ru*" had some changes of scores at days 95, 149, 334 and 345. Similarly, the domain "**spectr.ru*" had some changes of scores between day 195 and day 210 and the domain **sung.ru*" had changes of scores at days 96, 150, 335 and 346. In contrast to domains, connected IPs show more sporadic patterns (abrupt changes of scores). All the observed IPs had idle time period, where their badness scores were equal to 0. For instance, the IP "93.xxx.xx.xx0" had an abrupt change in day 147 and the IP "113.xx.xxx.xx6" had an abrupt change in day 200.

Figure 4.9 illustrates different owners sharing malware samples in July 2014. The colored graph network contains different communities, obtained by applying a fast unfolding community detection algorithm [34], where each color represents a community. We managed to obtain 23 communities. The graph nodes have been anonymized. Notice also that the bigger is a node, the higher is its badness score. For instance, the person "A.Di.M" has the highest badness score (0.029), followed by the person "D.VAN.A" (score of 0.028). The third place is shared between three persons, namely, “M.K.S.M”, “A.K”, and “A.D.de.M.S”, with a score of 0.026.

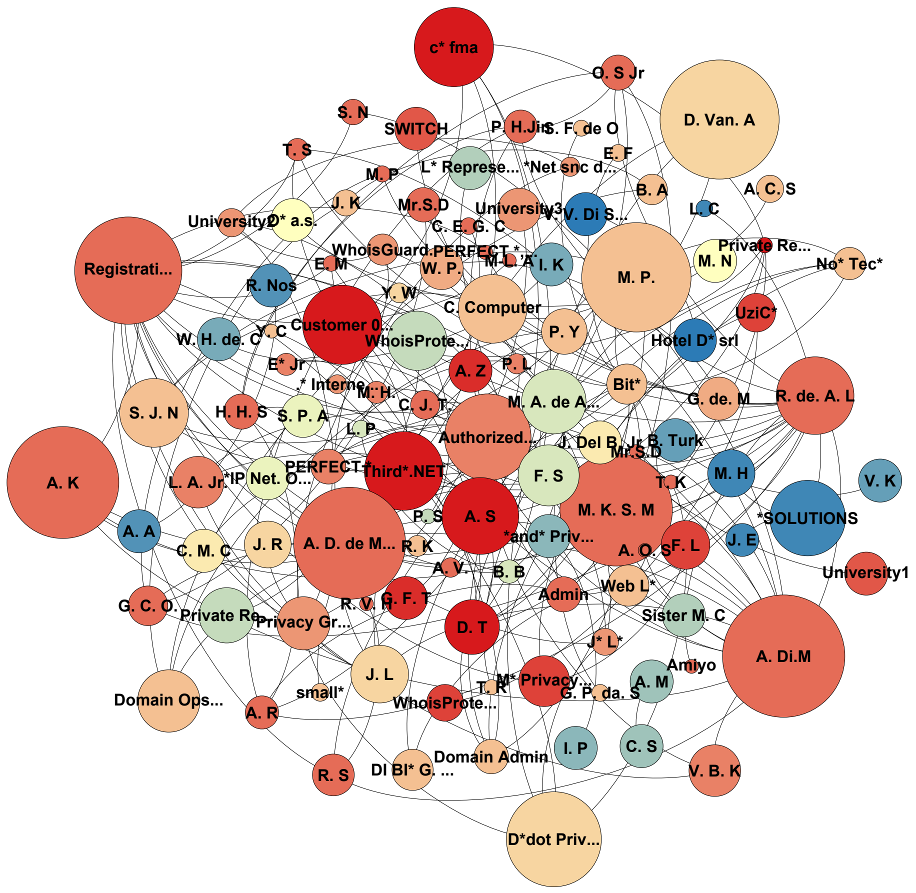


Figure 4.9: Registrants Communities and Badness Scores

4.3.4 Patterns Inference

Figures 4.10 and 4.11 illustrate the similarity matrix obtained during one year period. A major observation is that domains have more patterns than connected IPs. Similarly, to badness ranking, patterns in domains are more persistent and periodic. We observe high density in the first and last semesters in domain patterns similarity matrix. Connected IPs

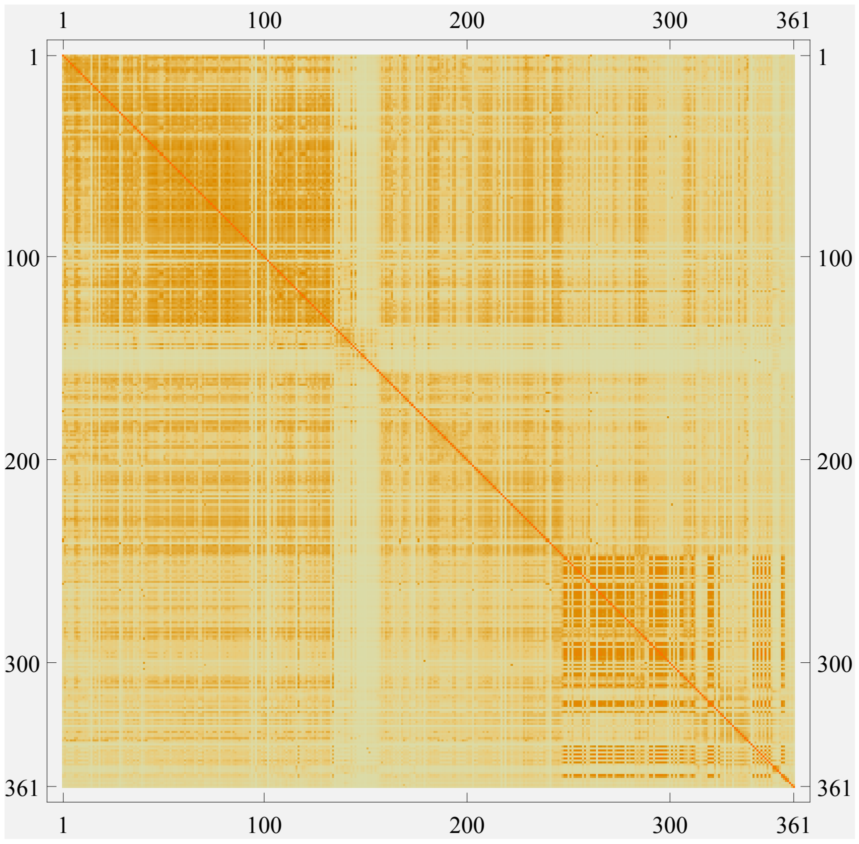


Figure 4.10: Domain Patterns Similarity Matrix

show less periodicity than domains. The presence of patterns tend to be ephemeral and the maximum period is commonly in the order of 1 to 60 days. However, there exists some IPs that have some persistence. Such case is illustrated hereafter in the pattern use cases.

Patterns	**Days**	**Mal.**	**Owners**
f*[dd]75.com;a*[dd]75.com	332	6,045	Registration Private
f*[dd]75.com;w*[dd]88	329	6,046	Registration Private
w*[dd]88.com;a*[dd]75.com	317	5,966	Registration Private

Table 4.12: Domain Patterns Use Case

Table 4.12 shows patterns involving dynamic domains generated in the same way.

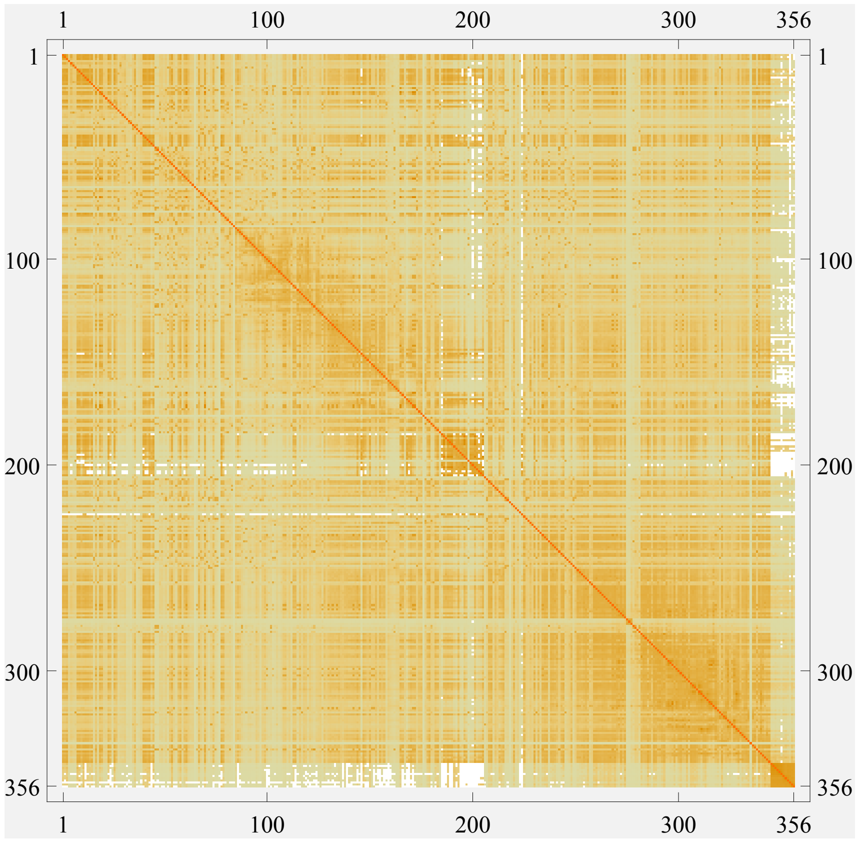


Figure 4.11: IP Patterns Similarity Matrix

In this case, we have three domain names with two letters followed by two digits. All these domains share a big number of malware samples (in order of $6,000$ malware samples). In addition, they have the same owner protected by the same privacy service. Such use case acknowledges the observation found in Table 4.8, i.e., the hosting companies with a privacy service are nests for suspicious domains.

Patterns	**Days**	**Mal.**	**Network Names**
220.xxx.xxx.7;125.xx.xxx.x4	317	2123	ORG1-BJ;ORG5
220.xxx.xxx.7;124.xxx.xxx.x1	311	2068	ORG1-BJ;ORG1-HE
221.xxx.xxx.x8;125.xx.xxx.x4	289	1938	ORG6;ORG5
220.xxx.xxx.7;221.xxx.xxx.x8	289	1948	ORG1-BJ;ORG6
124.xxx.xxx.x1;125.xx.xxx.x4	278	1925	ORG1-HE;ORG5
124.xxx.xxx.x1;221.xxx.xxx.x8	123	1836	ORG1-HE;ORG6

Table 4.13: IP Patterns Use Case

Table 4.13 shows IP patterns connected by thousands of malware samples. All these IPs are located in China, where we observe 4 organizations with 5 network names. All the patterns have appeared during long time periods: more than 300 days for the two first patterns, 289 days for the third and fourth patterns and 123 days for the last pattern. This use case indicates that there is a cluster of IP patterns that represents a collaborative malware activity in China.

4.4 Conclusion

In this chapter, we have presented an approach to investigate cyber-threats and the underlying infrastructures. To this end, we have used graph-theory concepts to rank the badness of different infrastructure elements. This allowed us to identify key players and quantify the sharing among these players. This is of paramount importance as it helps to unveil potential criminal groups. Moreover, we have presented a methodology to track the evolution of cyber-threat infrastructures over time and infer patterns of cyber-criminal activities. Using one year dataset collected from dynamic malware analysis, we have derived important insights about cyber-threats. In the next chapter, we rely on another data extracted from dynamic malware analysis to generate network cyber-threat intelligence. We use network traces to design and integrate techniques to fingerprint maliciousness in IP traffic. The fingerprinting is two-fold: a packet header flow-based approach and deep packet inspection using signal and NLP techniques.

Chapter 5

Malicious Traffic Fingerprinting

5.1 Overview

In this chapter, we address the problem of fingerprinting maliciousness of traffic for the purpose of detection and classification. We aim first at fingerprinting maliciousness by using two approaches: Deep Packet Inspection (DPI) and IP packet headers classification. To this end, we consider malicious traffic generated from dynamic malware analysis as traffic maliciousness ground truth. In light of this assumption, we present how these two approaches are used to detect and attribute maliciousness to different threats. In this chapter, we elaborate a comparative study between two traffic maliciousness fingerprinting techniques, Deep Packet Inspection (DPI) and IP packet headers classification. We evaluate each approach based on its detection and attribution accuracy as well as its level of complexity. The outcomes of both approaches have shown promising results in terms of detection; they are good candidates to constitute a synergy to elaborate or corroborate detection systems in terms of run-time speed and classification precision. In Section 5.2, we describe how we collect malicious traffic. In Section 5.3, we expose how we fingerprint maliciousness based on packet headers flows. Section 5.4 entails how we fingerprint maliciousness of packets based on DPI. We entail different results of both approaches in Section 5.5. We discuss the advantages and drawbacks of proposed fingerprinting approaches in Section 5.6. Finally, we conclude the chapter with few remarks in Section 5.7.

5.2 Traffic Maliciousness Ground Truth

We execute collected malware samples in a controlled environment (sandbox) to generate representative malicious traffic. This is used as a ground truth for maliciousness fingerprinting. The sandbox is based on a client-server architecture, where the server sends malware to clients. It is important to mention that the dynamic analysis setup allows malware to connect to the Internet to generate inbound/outbound malicious traffic. Figure 5.1 illustrates the dynamic malware analysis topology. We receive an average of $4,560$ malware samples on a daily basis from a third party. We execute the malware samples in the sandbox for three minutes. We chose this running period to make sure that we can handle up to $14,400$ malware runs per day. The period gives the ability to run all malware samples with a re-submission. The latter is important in case where malware does not generate network traffic during the initial runs. For each run, a client monitors the behavior of each malware and records it into report files. These files contain malware activities such as file activities, hooking activities, network activities, process activities, and registry activities. The setup of dynamic malware analysis lies in a network, which is composed of a server and 30 client machines. The server runs with an Intel(R) Core^{TM} i7 $920@2.67$ GHz, Ubuntu 11 64 bit operating system and 12.00 GB of physical memory (RAM). Each client runs with an Intel(R) Core^{TM} 2 $6600@2.40$ GHz, Microsoft Windows XP Professional 32-bit operating system and 1.00 GB of physical memory. Such physical machines are used for the reason that some malware samples cannot run in virtual machines. As a downstream outcome of the aforementioned dynamic analysis, we gathered the underlying traffic pcap files that were generated. The dynamic malware analysis has generated approximately $100,000$ pcap files labeled with the hashes of malware, which corresponds to a size of 3.6 GB. In our work, we considered inbound and outbound traffic generated by malware labeled by Kaspersky malware naming schema [3]. The reasons behind using this naming schema are as follows: (1) We noticed that it manages to cover the naming of the majority of malware samples considered in experiments. (2) The malware naming provided by Kaspersky follows the malware convention name (*Type.Platform.Family.Variant*). The number of bidirectional flows is $96,235$ and the number of unidirectional flows is $115,000$.

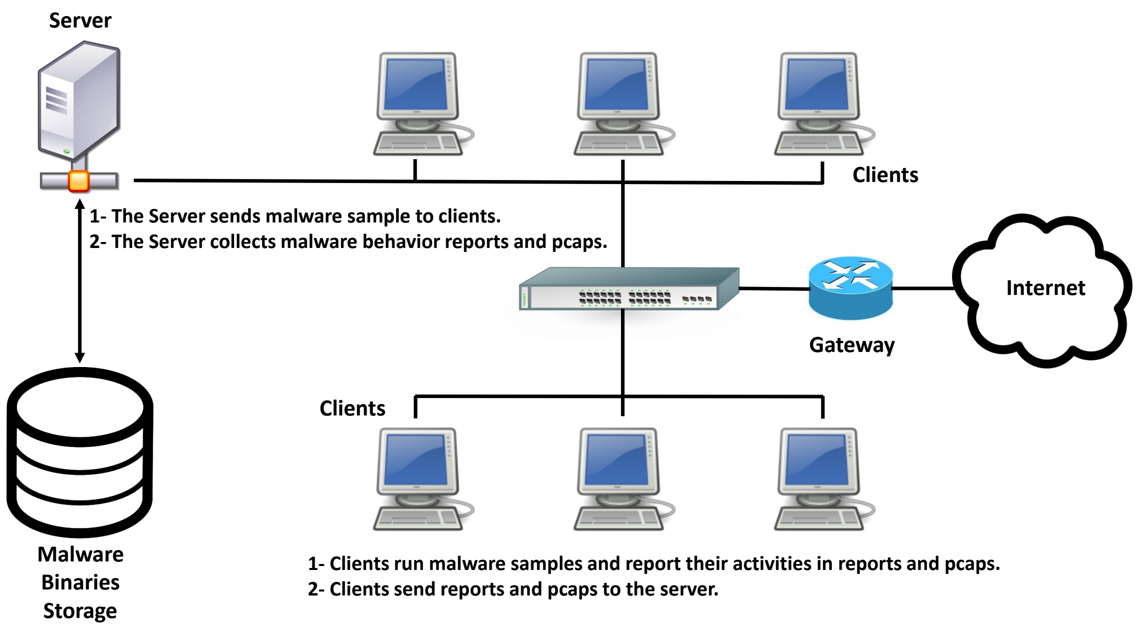

Server
Clients
1- The Server sends malware sample to clients.
2- The Server collects malware behavior reports and pcaps.
Internet
Gateway
Clients
Malware Binaries Storage
1- Clients run malware samples and report their activities in reports and pcaps.
2- Clients send reports and pcaps to the server.

Figure 5.1: Dynamic Malware Analysis Topology

5.3 Packet Headers Flow-Based Fingerprinting

In this section, we describe how packet headers flow fingerprinting is done. By fingerprinting, we mean (1) malicious traffic detection and (2) malware family attribution. First, for detection, we extract bidirectional flow features from malicious traces generated from dynamic malware analysis, together with benign traces collected from trusted third parties. These features are used by classification algorithms to create models that segregate malicious from benign traffic (see Section 5.3.1). Regarding attribution, we elaborate non-deterministic malware family attribution based on Hidden Markov Models (HMMs) [65]. The attribution is done through probabilistic scores for different sequences of malicious labeled unidirectional flows. The obtained models act as probabilistic signatures characterizing malware families.

5.3.1 Malicious Traffic Detection

Malicious traffic detection's goal is to isolate malicious communication sessions. These sessions include flows used to perform various malicious activities (e.g., malware payload delivery, DDoS, credentials theft). These flows are usually intermingled with a large portion of IP traffic that corresponds to benign activities over computer networks. As such, maliciousness detection amounts to the segregation of malicious from benign flows. To this end, we represent flows through a set of attributes (features) that capture their network behaviors. By leveraging these features, we create classifiers that automatically generate models to detect malicious traffic. With this in mind, we define four phases to infer maliciousness at the network level: selecting and extracting the bidirectional flow features, labeling of the traffic (malicious and benign), training machine learning algorithms, and evaluating the classifiers produced by these algorithms. Figure 5.2 illustrates how detection of maliciousness is performed.

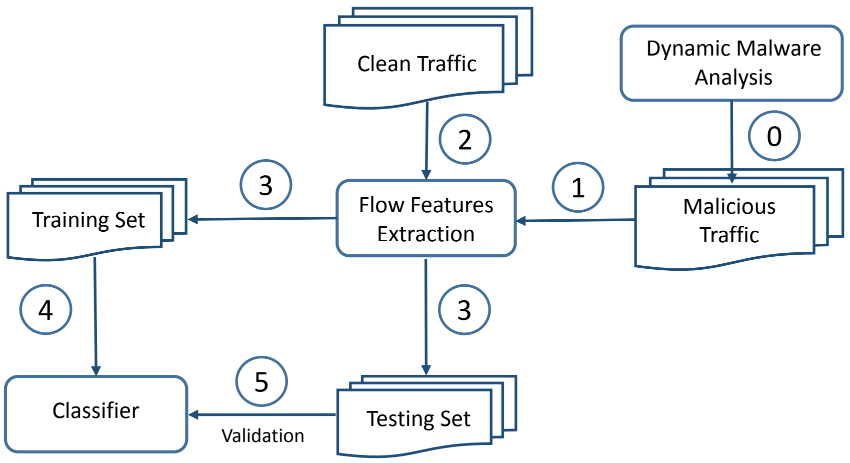


Figure 5.2: Flow-Based Detection Approach

Benign Traffic Datasets

For the purpose of building a classification model that distinguishes between malicious and benign traffic, we collected benign traffic from WISNET [8] and private companies. These datasets have been built to evaluate Intrusion Detection Systems in terms of false alerts and to detect anomalies in network traffic. In our work, we use such datasets to build baseline knowledge for benign traffic. These datasets have been used together with the malicious traffic dataset to assess classification algorithms in terms of accuracy, false positives and negatives. Table 5.1 shows the number of benign flows in each dataset. The different datasets used in this work illustrate four different location datasets, namely, residential setting, research laboratory, ISP edge router and private company networks.

Source	**Bidirectional Flows**	**Traffic Source**
WisNet (Home)	$10,513$ (85MB)	Residential setting
WisNet (ISP)	$65,000$ (1.1GB)	Research laboratory
WisNet (SOHO)	$16,504$ (1.3GB)	Edge router of an ISP
Private	$64,004$ (5.6GB)	Private company

Table 5.1: Benign Datasets

Bidirectional Flow Features Extraction

We capture malicious network traces from the execution of malware binaries in ThreatTrack's sandbox [210]. We label these traces accordingly as malicious, while the clean traffic traces obtained from trusted third parties [8] are labeled as benign. Flow features are extracted from these labeled network traces to capture the characteristics of malicious and benign traffic. It is important to mention that these features can be extracted even when the traffic is encrypted, as they are derived from flow packets headers. The flow features exploited are mainly based on flow duration, direction, inter-arrival time, number of exchanged packets, and packets size.

	Features
1	Flow Duration
2	Number of forward packets
3	Number of backward packets
4	Protocol
5	Minimum inter-arrival time for forward packets
6	Maximum inter-arrival time for forward packets
7	Mean inter-arrival time for forward packets
8	Std deviation inter-arrival time for forward packets
9	Total forward packets size
10	Minimum forward packets size
11	Maximum forward packets size
12	Mean forward packets size
13	Std deviation forward packets size
14	Minimum inter-arrival time for backward packets
15	Maximum inter-arrival time for backward packets
16	Mean inter-arrival time for backward packets
17	Std deviation inter-arrival time for backw. packets
18	Total backward packets size
19	Minimum backward packets size
20	Maximum backward packets size
21	Mean backward packets size
22	Std deviation backward packets size

Table 5.2: Bidirectional Flow Features

A bidirectional flow is a sequence of IP packets that share the 5-TCP-tuple (source IP, destination IP, source port number, destination port number, protocol). The outbound traffic is represented by the forward direction, while the backward direction represents the inbound traffic. In terms of design and implementation, the module in charge of network traces parsing, labeling, and feature extraction reads traffic using jNetPcap API [2], which decodes captured network flows in real-time or offline. This module produces values for different features from network flows. The resulted values are stored in features files that are provided to Weka [7]. The network traces parser represents each flow by a vector of 22 flow features. Table 5.2 illustrates the description of the bidirectional flow features.

Traffic Classification

The feature files resulting from the previous phase are provided as input to classification algorithms. The intent is to build models that have the ability to distinguish between malicious and benign flows. To this end, we use machine learning algorithms, namely, Boosted J48, J48, Naïve Bayesian, Boosted Naïve Bayesian, and SVMs. The classification module is based on a Java wrapper that runs these machine learning algorithms. The module has two execution phases: learning and testing. In the learning phase, we build the classifier using 70% of malicious and benign traces. In the testing phase, we evaluate the classifier with the rest of the data (30%). It is important to mention that training and testing datasets do not overlap with each other. In the sequel, we give a brief overview of the classification algorithms.

J48 Algorithm: It is a Java implementation of C4.5 classification algorithm [169]. J48 [73] builds the tree by dividing the training data space into local regions in recursive splits. The split is pure if all observations in a decision branch belong to the same class. To split the training dataset, J48 computes the goodness of each attribute (feature) to be the root of a decision branch. It begins by computing the *information need* factor. The J48 algorithm splits recursively datasets to sub-datasets and computes the *information need* per feature. If the split is not pure (presence of many classes), the same process will be used to split the sub-dataset into pure classes. The split stops if each sub-dataset belongs to one class (pure split). The decision tree result is composed of nodes (the attributes) and terminal leaves (classes). That will be used to identify the unseen data when the model is deployed [84].

Naïve Bayesian Algorithm: It is based on Bayes' theorem [84]. It is a statistical classifier, which outputs a set of probabilities that show how likely a tuple (observation) may belong to a specific class [84]. Naïve Bayesian assumes that the attributes are mutually independent. Naïve Bayesian starts by computing the probability of an observation. Once the probabilities are computed, Naïve Bayesian associates each observation with the class that has the higher probability with it. Naïve Bayesian is an incremental classifier, which means that each training sample will increase or decrease the probability that a hypothesis is correct.

Boosting Algorithm: It is a method used to construct a strong classifier from a weak learning algorithm. Given a training dataset, the boosting algorithm incrementally builds the strong classifier from multi-instances of a weak learning machine algorithm [76]. Boosting algorithm takes, as input, the training dataset and the weak learning algorithm. It divides the training dataset into many sub-datasets $(x_1, y_1), ..., (x_i, y_j)$, where x_i belongs to X (a set of observations) and y_j belongs to Y (set of class attribute values), and calls the weak learning algorithm to build the model for each sub-dataset.

The resulted models are called decision stumps. The latter examine the features and return the decision tree with two leaves either $+1$ if the observation is in a class, or -1 if it is not the case. The leaves are used for binary classification (in case the problem is multi-classes, the leaves are classes). Boosting algorithm uses the majority voting schema between decision stumps to build a stronger model.

SVM Algorithm: The Support Vector Machines (SVM) [74, 85] algorithm is designed for discrimination, which is meant for prediction and classification of qualitative variables (features) [156]. The basic idea is to represent the data in a landmark, where the different axes are represented by the features. The SVM algorithm constructs a hyper-plane or set of hyper-planes in a high-dimensional data. Then, it searches for the hyper-plane that has the largest distance to the nearest training data points of any class. The larger is the distance, the lower is the error of the classification.

5.3.2 Malicious Traffic Attribution

The attribution of malicious traffic to malware families corroborates detection since it (1) shifts the anti-malware industry from the system level to the network level, and (2) eases the mitigation of infected machines. It gives the ability to networking staff to undertake actions against botnets, depots of stolen information, spammers, etc. For instance, if an administrator notices the presence of malicious traffic in the network, and this traffic can be attributed to a bot family. He/She responds to the threat by blocking malicious connections and quarantine infected machines for a removal of malware. Thus, to enhance maliciousness fingerprinting at the network level, we decide to integrate the malware family attribution. To do so, we use network traces obtained from dynamic malware analysis and index them with malware families. For each set of traces belonging to a malware family, we extract sequences of unidirectional flows. These flows are labeled through a clustering method. The labeled sequences obtained are used to train HMMs for different malware families. Figure 5.3 illustrates how malware families' attribution is performed.

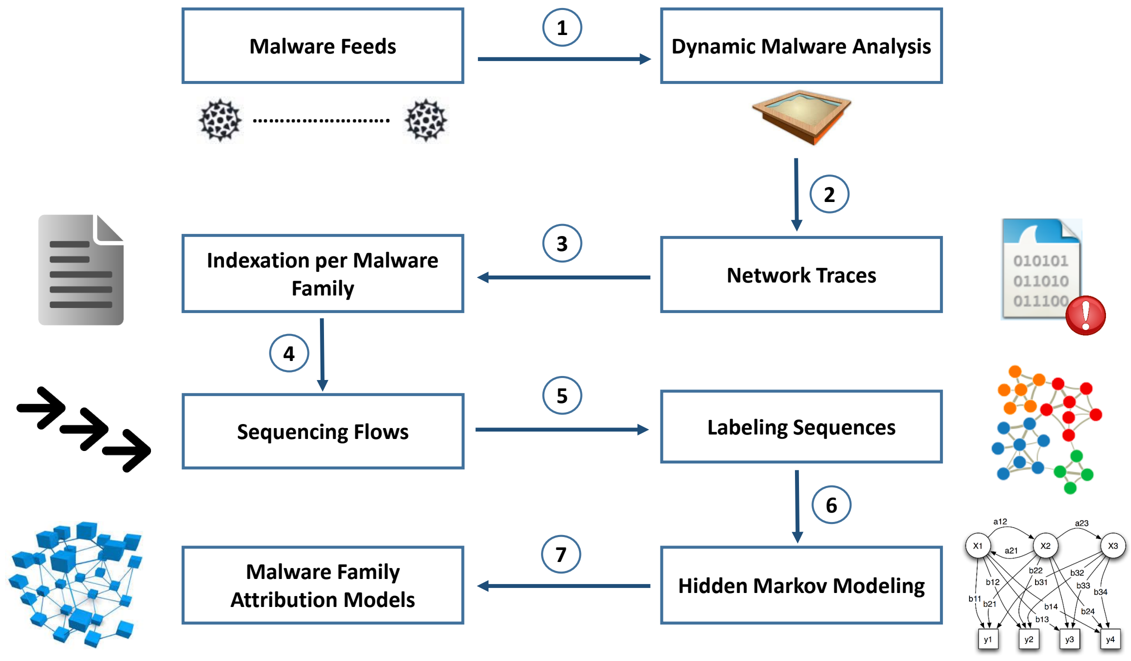


Figure 5.3: Non-Deterministic Approach for Malware Family Attribution

Malware Family Indexation

As a downstream outcome of dynamic malware analysis, we collect approximately 100, 0-00 network traces (pcap files). Each trace is labeled by the corresponding malware sample hash. We use Kaspersky malware name schema to recognize the malware family of each hash. Subsequently, we index network traces based on their malware family. In this work, we obtain 294 malware families.

Sequencing Flows

For each malware family, we browse collected network traces to extract unidirectional flows. These flows fall into inbound and outbound flows, which are used to build sequences of flows. For each sequence, a flow precedes another flow if its timestamp occurrence precedes the timestamp of the following flow. Obtained sequences are indexed by their corresponding malware family.

Labeling Sequences

In order to label different flows belonging to a sequence, we adopt a clustering approach. The reason behind doing so is to characterize outbound and inbound malicious flows into clusters representing their network behaviors. To do so, we represent flows by vectors of 45 features. Table 5.3 illustrates unidirectional flow features. To perform clustering of inbound/outbound traffic, we generate feature files that are readable by CLUTO clustering toolkit [112]. This was used in diverse research topics such as information retrieval [241] and fraud detection [162]. To label flows, we use the k-means Repeated Bi-Section algorithm [198] implemented in CLUTO. This algorithm belongs to partitional clustering algorithms. The latter are known to perform in clustering large datasets since they have

Features		
Generic	1	Total number of packets.
	2	Flow duration
	3	Minimum inter-arrival time
	4	First quartile of inter-arrival times
	5	Median of inter-arrival times
	6	Mean of inter-arrival times
	7	Third quartile of inter-arrival times
	8	Maximum inter-arrival time
	9	Variance of inter-arrival times
	10	Minimum of control data size
	11	First quartile of control data size
	12	Median of control data size
	13	Mean of control data size
	14	Third quartile of control data size
	15	Maximum of control data size
	16	Variance of control data size
	17	Total not empty packets
	18	Total packets size
Ethernet	19	Minimum size in Ethernet packets
	20	First quartile size in Ethernet packets
	21	Median size in Ethernet packets
	22	Mean size in Ethernet packets
	23	Third quartile size in Ethernet packets
	24	Maximum size in Ethernet packets
	25	variance size in Ethernet packets
Network	26	Minimum size in IP packets
	27	First quartile size in IP packets
	28	Median size in IP packets
	29	Mean size in IP packets
	30	Third quartile size in IP packets
	31	Maximum size in IP packets
	32	Variance size in IP packets
Transport	33	Total ACK packets
	34	Total PUSH packets
	35	Total SYN packets
	36	Total FINE packets
	37	Total Urgent packets
	38	Total Urgent bytes
	39	Minimum TCP segment size
	40	Maximum TCP segment size
	41	Mean TCP segment size
	42	Minimum TCP window size
	43	Maximum TCP window size
	44	Mean TCP window size
	45	Total empty TCP window packet

Table 5.3: Unidirectional Flow Features

a low computational cost [15, 113]. The k-means RBS algorithm derives clustering solutions based on a global criterion function [239]. This algorithm initially creates 2 groups, each group is then bisected until the criterion function is optimized. The k-means RBS algorithm uses the vector space model [178] to represent each unidirectional flow. Each flow is represented by a dimension vector $fv = (f_1, f_2, \ldots, f_i)$, where f_i is the i^{th} unidirectional flow feature. To compute similarity between vectors, we use the cosine function [178]. In order to cluster different unidirectional flows, we use a hybrid criterion function that is based on internal and external functions. The internal function tries to maximize the average pairwise similarities between flows that are assigned to each cluster. Unlike the internal criterion function, the external function derives the solution based on how the various clusters are different from each other. The hybrid function combines external and internal functions to simultaneously optimize both of them. Based on the k-means RBS algorithm, we create a set of experiments: inbound flow clustering solutions and outbound flow clustering solutions. We choose a solution where the internal similarity metric (ISIM) is high and the external similarity metric (ESIM) is moderate.

Hidden Markov Modeling

The Hidden Markov Model (HMM) is a popular statistical tool that models timeseries or sequential data. In this work, we use HMMs to create non-deterministic models that profile malware families. We want to establish a systematic approach to estimate attribution of unidirectional flow sequences to different malware families. We choose HMMs due to their readability since they allow sequences to be significantly interpreted, represented, and scored. We observe that collected flows have different lengths, and therefore decide to train HMMs based on sub-sequences with fixed length m. In order to fix the number of states in HMMs, we set a sliding window n to represent different combinations of inbound and outbound flows. For instance, if we want HMM states to represent a singular flow, there exist two possibilities: IN and OUT. If we want HMM states to represent a sequence of two flows, there are four possibilities: IN/IN, IN/OUT, OUT/OUT and OUT/IN. Similarly, if we want HMM states to represent a sequence of n flows, we obtain 2^n states. To train HMMs for each malware family with corresponding sequences, we use the Expectation Maximization (EM) algorithm [132] integrated in the HMMall toolbox for MATLAB [146] to learn hidden parameters of each 2^n state HMM representing a malware family.

Hidden Markov Models Initialization

To create models for different malware families, we initiate baseline HMMs. The states are computed based on a sliding window that we apply on observed sequences. The sliding window allows us to extract sub-sequences from sequences. For instance, for a sequence (a, b, c) and a sliding window of length 2, we obtain sub-sequences (a, b), (b, c). If we consider a HMM based on a sliding window of 1 flow, we result in 2 states HMM since we can have an inbound flow or an outbound flow. If a HMM is based on 2 flows, we obtain 4 states HMM since we can have an inbound/inbound pair, an

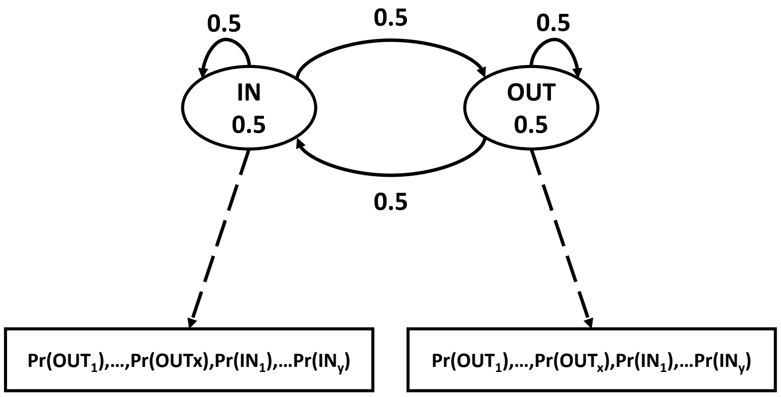

0.5
0.5
0.5
IN
0.5
OUT
0.5
0.5
Pr(OUT₁),…,Pr(OUTx),Pr(IN₁),…Pr(INy)
Pr(OUT₁),…,Pr(OUTx),Pr(IN₁),…Pr(INy)

Figure 5.4: Two-State Initialization HMM

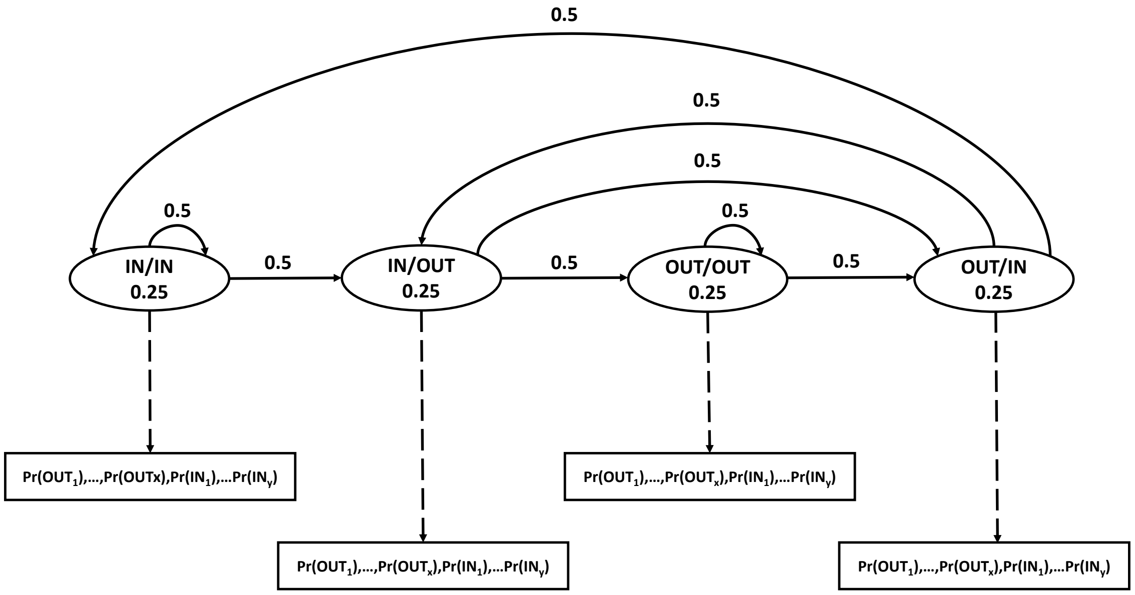

0.5
0.5
0.5
0.5
0.5
IN/IN
0.25
0.5
IN/OUT
0.25
0.5
OUT/OUT
0.25
0.5
OUT/IN
0.25
Pr(OUT₁),…,Pr(OUTx),Pr(IN₁),…Pr(INy)
Pr(OUT₁),…,Pr(OUTx),Pr(IN₁),…Pr(INy)
Pr(OUT₁),…,Pr(OUTx),Pr(IN₁),…Pr(INy)
Pr(OUT₁),…,Pr(OUTx),Pr(IN₁),…Pr(INy)

Figure 5.5: Four-State Initialization HMM

inbound/outbound pair, an outbound/outbound pair and an outbound/inbound pair. In initialized HMMs, prior probabilities are uniformly distributed over different states. For instance, if we consider a sliding window of length 2, we obtain 4 states HMM with a prior probability of 0.25 for each state. The transition probabilities matrix is initialized such that for each transition between a state s_i and other states, the probabilities are uniformly distributed. If a state has 2 transitions, each transition has a probability of 0.5. The emission probabilities matrix associates a state with an observation vector. Each element of the vector is a probability of observing an inbound or outbound clustering label. For the sake of simplicity, we illustrate, in Figures 5.4 and 5.5 initialization HMMs for a sliding window length of 1 and 2 respectively. The observation probabilities are uniformly distributed. Let us consider x as the number of input labels and y as the number of output labels. For a 2-state HMM, we associate with the state IN an observation vector, where:

$\forall i \in [1, x]$: $b(in_i) = 1/x$

$\forall j \in [1, y]$: $b(out_j) = 0$

Similarly, we associate with the state OUT an observation vector, where:

$\forall i \in [1, x]$: $b(in_i) = 0$

$\forall j \in [1, y]$: $b(out_j) = 1/y$

For a 4-state HMM, we associate with the state IN/IN an observation vector, where:

$\forall i \in [1, x]$: $b(in_i) = 1/x$

$\forall j \in [1, y]$: $b(out_j) = 0$

Similarly, we associate with the state OUT/OUT an observation vector, where:

$\forall i \in [1, x]$: $b(in_i) = 0$

$\forall j \in [1, y]$: $b(out_j) = 1/y$

Regarding states OUT/IN and IN/OUT, the observation vector is as follows:

$\forall i \in [1, x]$: $b(in_i) = 1/(x + y)$

$\forall j \in [1, y]$: $b(out_j) = 1/(x + y)$

Recursively, for a 2^n-state HMM, the observation vectors are the same as a 4-state HMM. If the states contain IN and OUT, the probabilities are equal to $1/(x + y)$. If the states contain just IN, the probabilities are equal to $1/x$ for all inbound labels and 0 for all outbound labels. Similarly, if the states contain just OUT, the probabilities are equal to $1/y$ for all outbound labels and 0 for all inbound labels.

5.4 Signal and NLP DPI Fingerprinting

In the sequel, we describe the DPI approach to detect maliciousness in network traffic. The methodology used to analyze malicious packets is described in Section 5.4.1, whereas the different knowledge base machine learning techniques are introduced in Section 5.4.2.

Section 5.4.3 describes the different steps to classify packets. In this approach, we look at the packets, including both headers and payloads. Packets are considered as signals inputs for fast spectral-based classification.

5.4.1 Core Principles

The essence of the whole packet analysis lies in the core principles, which fall into machine learning and Natural Language Processing (NLP) techniques. A set of malicious packets (a network trace) is treated as a data stream signal, where n-grams are used to build a sample amplitude value in the signal. In our case, we use bi-grams ($n = 2$) (two consecutive characters or bytes) to construct the signal. The reasons behind using bi-grams lay in: (1) It has shown its effectiveness in detecting C&Cs channels [82]; (2) it is well established artifact integrated in the MARF framework and used to analyze signal and Natural Language Processing (NLP).

Similarly to the aforementioned approach, the whole packet methodology has two phases: (1) The training phase, where MARFPCAT learns from different samples of network traces and generates spectral signatures using signal processing techniques; and (2) the testing phase, where MARFPCAT computes how similar or distant training network traces are from testing network traces. This approach is meant to behave like a signature-based anti-virus or IDS, but using fuzzy signatures. However, we use as much as possible combinations of machine learning and signal processing algorithms to assess their precision and runtime in order to select the best trade-off combination.

At present, we look at complete pcap files, which can affect negatively the MARF-PCAT's malware family attribution accuracy in the presence of encrypted payload. However, MARFPCAT processes network traces quickly since there is no pre-processing of pcap traces (flows identification and extraction). MARFPCAT has the ability to control thresholds of different algorithms, which gives flexibility in selecting the appropriate classification technique.

5.4.2 Knowledge Base

Collected malware database's behavioral reports and network traces are considered as a knowledge base, from which we machine-learn the malicious pcap samples. As such, conducting the experiments fall into three broad steps: (1) Teach the system from the known cases of malware from their pcap data. (2) Test on the known cases. (3) Test on the unseen cases. In order to prepare data for training and testing, we used a Perl script to index pcap traces with malware classes, and we used the same malware naming conventions mentioned earlier. The index is in the form of a meta MARFCAT-IN XML file, which is used by MARFPCAT for training or testing.

In contrast to the packet headers approach, where the benign traffic is collected from third parties; the benign traffic is considered as a noise sample found in pcap traces. To segregate such traffic from the malicious one, we use the low-pass filters and silence compression. In addition, the signal of the benign traffic can be learnt and subtracted from malicious traffic (malicious signal) to increase fingerprinting accuracy.

5.4.3 MARFPCAT's DPI Methodology

In this part, we describe the different steps that are performed to fingerprint maliciousness by using DPI approach. We compile manually annotated meta-XML index files with a Perl script. The index file annotates malware network traces (pcaps) indexed by their families. Once the annotation is done, MARF is automatically trained on each pcap file by using a signal pipeline or a NLP pipeline (see Figure 5.6). The algorithms used in the training phase are detailed in [134]. The MARFPCAT tool loads training data as a set of bytes forming amplitude values in a signal (e.g, 8kHz, 16kHz, 24kHz, 44.1kHz frequency). Uni-gram, bi-gram or tri-gram approaches can be used to form such a signal. A language model works in a similar way, with the exception of not interpreting the n-grams as amplitudes in the signal. After the signal is formed, it can be pre-processed through filters or kept in its original form. The filters fall into normalization, traditional frequency domain filters, wavelet-based filters, etc. Feature extraction involves reducing an arbitrary length signal to a fixed length feature vector, which is thought to be the most relevant features in the signal (e.g., spectral features in Fast Fourier Transform (FFT) and Linear Prediction Coefficient (LPC)), min-max amplitudes, etc. The classification stage is then separated to either train by learning the incoming feature vectors (usually k-means clusters, median clusters, or plain feature vector collection combined with, for example, neural network training), or testing them against previously learned models. The testing stage is done on the training and testing data, originally two separated sets with and without annotations. In our methodology, we systematically test and select the

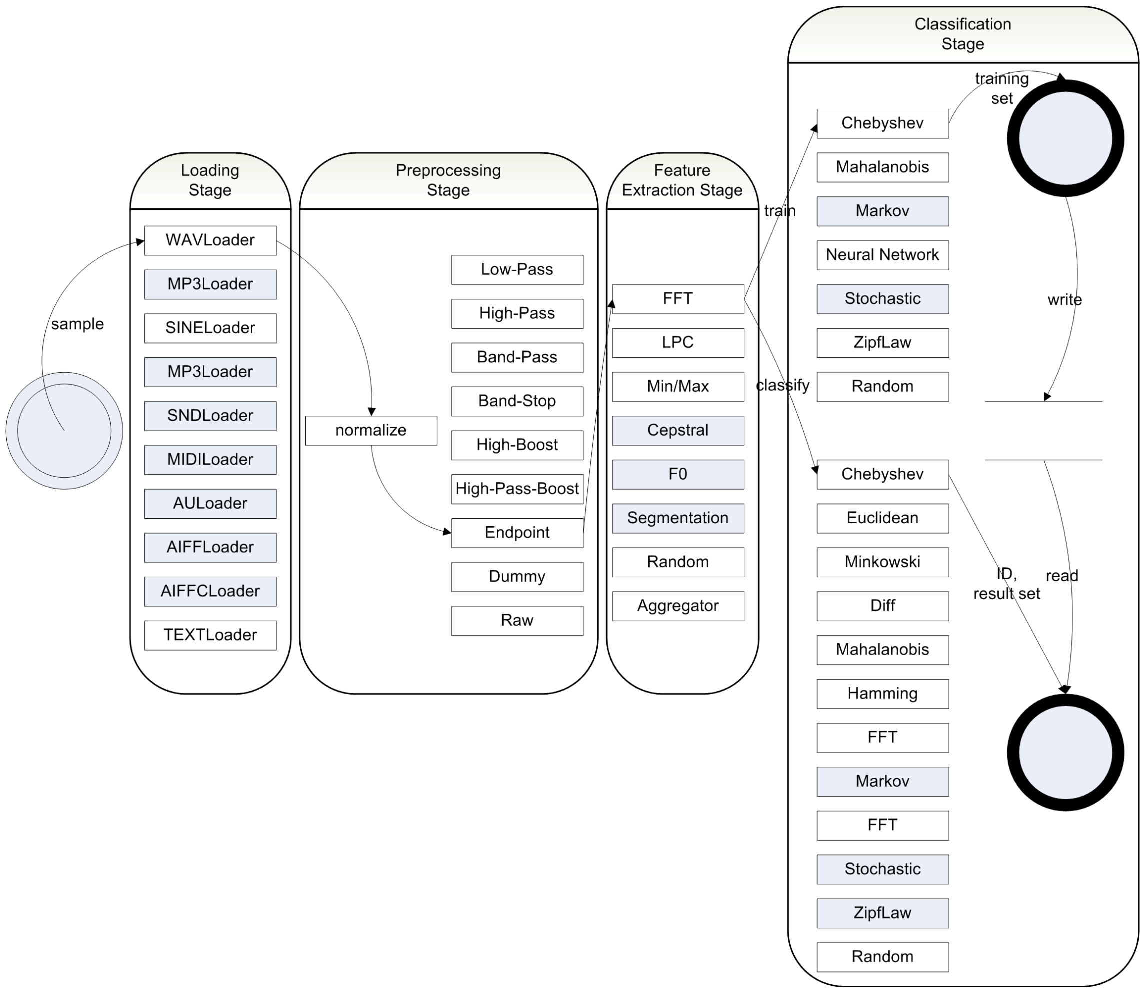


Figure 5.6: MARF's Pattern-Recognition Pipeline

best (a tradeoff between speed and precision) combination(s) of the different algorithms available in the MARF framework for subsequent testing.

5.4.4 NLP Pipeline

In this section, we present the inner-workings of MARF framework's. The latter uses algorithms that come from the classical literature (e.g., [128]) and detailed in [141]. NLP pipeline loading refers to the interpretation of the files being scanned in terms of n-grams and the associated statistical smoothing algorithms resulting in a vector, 2D or 3D matrix. In the case of static code vulnerability analysis, NLP pipeline has shown high precision [141]. However, its runtime was ≈ 10 times longer for an equivalent signal processing run. A such, for the time being we stopped using NLP pipeline for maliciousness fingerprinting in traffic. We plan to revive it with a more optimized implementation since MARF framework is an open-source software.

Initially, we compile meta-XML files to index malware traffic instances. For each indexed network trace, we use default uni-gram language models specification, described in [134]. Then, we train the system based on the index to build the knowledge base (machine learning). This is done by loading n-grams and use statistical smoothing estimators. We test again the training data against built models to deduce the best algorithms combinations for the learning phase. Once the learning phase is done, we test obtained models against testing data (network traces indexed by malware families). To improve the testing of different learning models, we use a demand-driven distributed evaluation, which is described in the sequel.

5.4.5 Demand-Driven Distributed Evaluation

To enhance the scalability of DPI evaluation approach [238], we converted the MARFPCAT stand-alone application to a distributed application using an educative model of computation (demand-driven) implemented in the General Intensional Programming System

(GIPSY)'s multi-tier run-time system [83, 97, 159, 214], which can be executed distributively using Jini (Apache River) [69]. To adapt MARFPCAT to the GIPSY's multi-tier architecture, we create Problem-Specific Distributers Generator and Worker tiers (PS-DGT and PS-DWT respectively). The generator produces demands of what needs to be computed in the form of a file (binary) to be evaluated, and deposits such demands as pending into a store managed by the demand store tier (DST). Workers pick up pending demands from the store and then process them (all tiers run on multiple nodes) using a traditional MARFPCAT instance. Once the result is computed, the PS-DWT deposits it back into the store with the status set to *computed*. The generator "harvests" all computed results and produces the final report for a test cases. Multiple test cases can be evaluated simultaneously, or a single case can be evaluated distributively. This approach helps coping with large amounts of data and avoiding recomputing tests that have already been computed and cached in the DST.

The initial basic experiment assumes the PS-DWTs have the training sets data and the test cases available from the beginning (either by a copy or via mounted volumes); thus, the distributed evaluation concerns only the classification task as of this version. The follow up work will remove this limitation. In this setup, a demand represents a file to scan, which is deposited into the DST. The PS-DWT picks up the file and checks it per training set that is already there, and returns a `ResultSet` object back into the DST under the same demand signature that was used to deposit the path to scan. The result set is sorted from the most to the least likely with a value corresponding to the distance or similarity. The PS-DGT picks up the result sets, performs the final output aggregation, and saves the report.

One of the output formats that MARFPCAT supports is FORENSIC LUCID [137], a language used to specify and evaluate digital forensic cases. Following the methodology of FORENSIC LUCID data export described in [139, 140], we use it as a format for evidential processing of the results produced by MARFPCAT. The work [139] provides details of the language; it suffices to mention that the report generated by MARFPCAT in FORENSIC LUCID is a collection of warnings, which form an evidential statement in FORENSIC LUCID.

5.4.6 Wavelets

As part of a collaboration project, wavelet-based signal processing for the purposes of noise filtering is used in this work to compare it to no-filtering, or FFT-based classical filtering. It has been also shown in [116] that wavelet-aided filtering could be used as a fast pre-processing method for network application identification and traffic analysis [119]. We rely on the algorithm and methodology described in [13, 107, 108, 183]. At this point, only a separating Discrete Wavelet Transform (DWT) [86] has been tested. Since the original wavelet implementation [183] is in MATLAB [129, 179], we use the `codegen` tool [131] from the MATLAB Coder toolbox [130] to generate C/C++ code. Then, we implement it in MARF and MARFPCAT language, namely JAVA. In addition, the function for up/down sampling used by the wavelets function described in [144], is also integrated in MARF framework.

5.5 Results

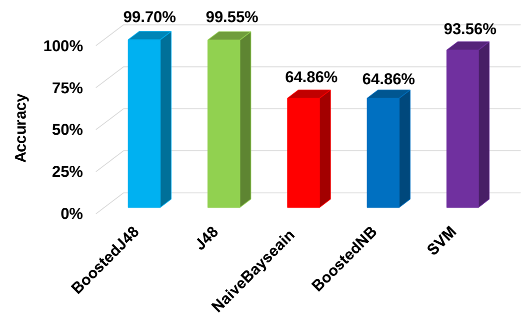
5.5.1 Non-DPI Approach

In the sequel, we present results obtained for Non-DPI fingerprinting approach. The results fall into 3 parts: (1) classification results, (2) attribution results, and (3) computational complexity of the approach.

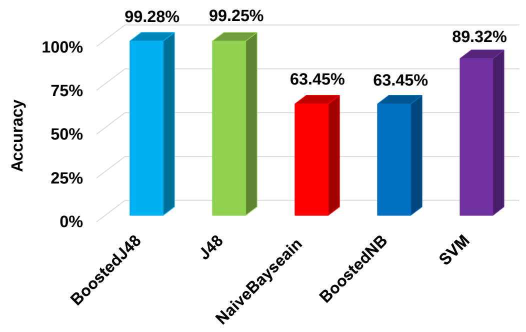
Classification

The purpose of this classification exercise is to determine whether we can segregate malicious from benign traffic. In addition, we make a comparison between different classification algorithms in terms of accuracy and recall. Our intent is to identify a classifier with high accuracy, low false positives, and low false negatives.

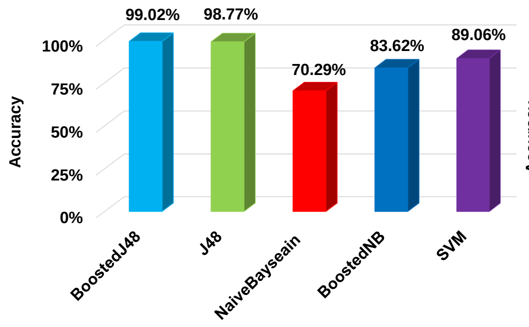
The results illustrated in Figures 5.7a, 5.7b, 5.7c, 5.7d, 5.7e and 5.7f demonstrate that the Boosted J48 and J48 algorithms have shown better results than other machine learning algorithms. They achieved 99% accuracy and less than 1% false positives and negatives, respectively. The SVM algorithm has achieved good results with an accuracy ranging between 89% and 95%. In contrast, Naïve Bayesian and Boosted Naïve Bayesian algorithms have not achieved good results. As such, we can claim that the Boosted J48 algorithm is a good mean to differentiate between malicious and benign traffic.



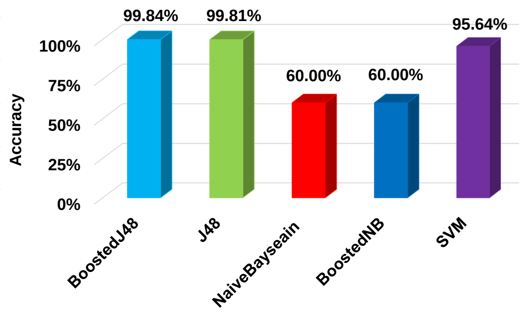
(a) Malicious vs. Benign (home)



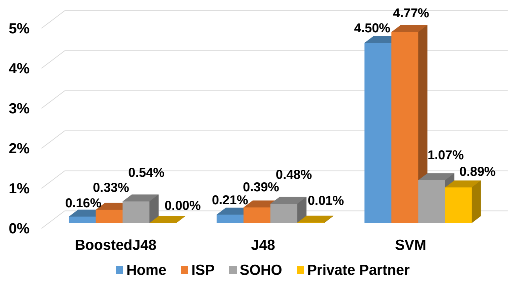
(b) Malicious vs. Benign (SOHO)



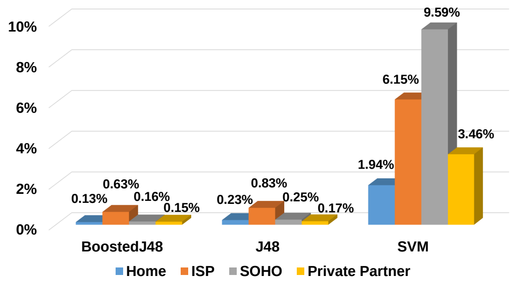
(c) Malicious vs. Benign (ISP)



(d) Malicious vs. Benign (Private)



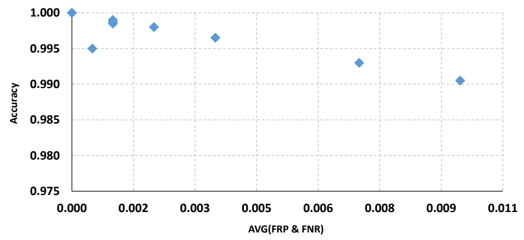
(e) False Positive Rate



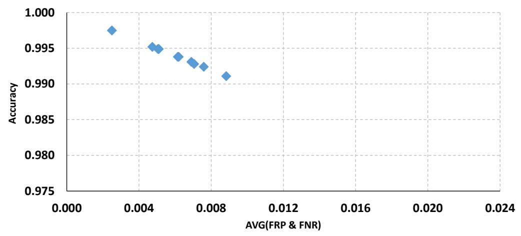
(f) False Negative Rate

Figure 5.7: Classification Algorithms Results

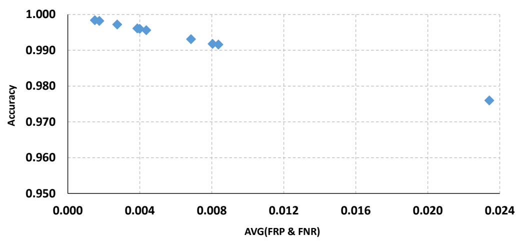
After finding that J48 is the most suitable algorithm, we used the 10-fold cross-validation method to select the training and testing data. This is done to ensure that the J48 algorithm maintains high accuracy and low false positive and negative rates, even if the training and testing data change. Figures 5.8a, 5.8b, 5.8c and 5.8d summarize the performance of the J48 algorithm in each data set by providing the accuracy and the rates of false positives and negatives.



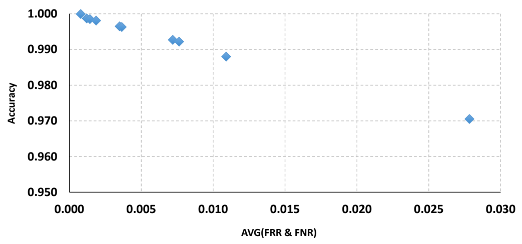
(a) Malicious and Benign Home Datasets



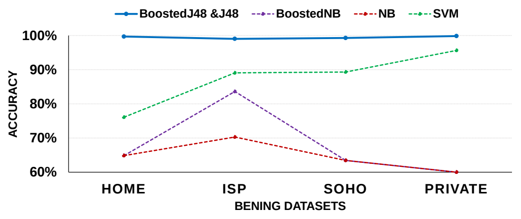
(b) Malicious and Benign ISP Datasets



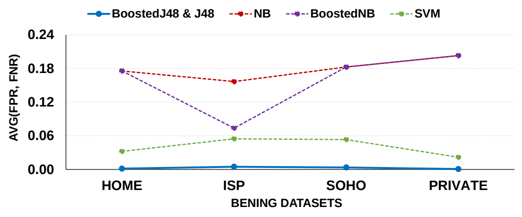
(c) Malicious and Benign SOHO Datasets



(d) Malicious and Benign Private Datasets



(e) Change in Accuracy per Dataset



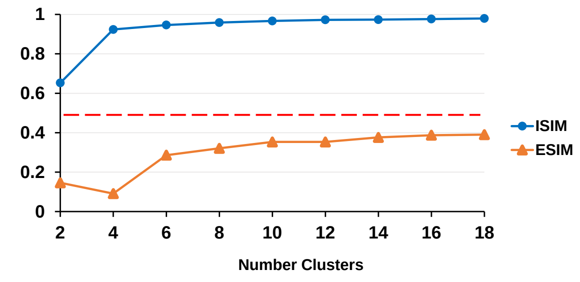
(f) Change in Average of FPR and FNR per Dataset

Figure 5.8: J48 Classifiers Performance and Generalization

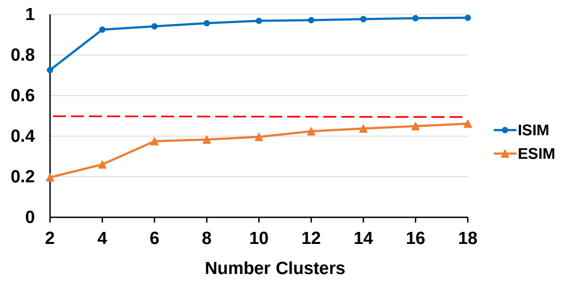
The boosted J48 and J48 algorithms have achieved high accuracy detection and low rates of false positives and negatives in multiple datasets. The fact that we use different datasets has shown that the J48 classification approach is robust since it maintains greater than 98% accuracy with less than 0.006 average false alerts for each dataset, as illustrated in Figure 5.8e and Figure 5.8f respectively. Thus, these two algorithms provide the means necessary to make malicious traffic differentiable from benign traffic. Moreover, the results conclude that our approach, based on classifying the flow features, can achieve maliciousness detection in different benign traffic with a very high detection rate and low false alerts. The J48 algorithm does not rely on features dependency and tends to perform better with a limited number of classes, which is the case of our work since we have two classes. On the other hand, Naïve Bayesian shows bad results since it relies on the independence of features, which is not the case in maliciousness classification. For example, packet length depends on frame length. Regarding SVM, we use it with the default option where linear classification is performed. This raises a problem with probabilities of class membership.

Attribution

In order to attribute malicious flows to malware families, we apply a clustering technique to label different inbound and outbound unidirectional flows. We consider k-means RBS clustering solutions for inbound and outbound traffic. The solutions are generated heuristically by incrementing by two the number of clusters for inbound and outbound flows. To evaluate the solutions, we take into account: (1) the high Internal Similarity Metric (ISIM) average in all clusters, and (2) the moderate External Similarity Metric (ESIM) average in all clusters. The ISIM average mirrors the cohesion between items (unidirectional flows) within different clusters. The ESIM average defines the isolation between different clusters. In our labeling process, we consider solutions that vary from 2 to 18 clusters. We consider up to 18 clusters to preserve the potential to have a sufficient number of labels for both inbound and outbound flows. Figures 5.9a and 5.9b illustrate the ISIM and ESIM averages for different inbound and outbound clustering solutions. The selection of labeling solutions is based on two criteria: (1) a high ISIM average ratio (greater than or equal to 0.95), and (2) a moderate ESIM average ratio between clusters (less than or equal to 0.5). As such, we consider only those solutions which vary from 12 to 18 clusters for both inbound and outbound flows.



(a) Inbound Flows Clustering



(b) Outbound Flows Clustering

Figure 5.9: ISIM, ESIM vs. Clustering Solutions

By coupling inbound and outbound clustering solutions, we obtain 16 possible labeling combinations. For each combination, we compute the uniqueness of the collected sequences. We observe the ratio of labeled sequences that are not shared by malware families. The higher the uniqueness of the sequences ratio, the higher the ability to segregate malware families. We can thus limit the attribution of malicious flows to a limited number of malware families. Table 5.4 shows the uniqueness ratio for each labeling combination. Based on the obtained ratios, we choose a solution with 16 inbound flows and 18 outbound flows, since it has the highest uniqueness ratio. This labeling combination is used to initialize HMMs and train them for each malware family.

Table 5.4: Uniqueness Ratio per Combination of Clustering Solutions

		OUT Flows Clustering			
		12	14	16	18
IN Flows Clustering	12	0.7230	0.7097	0.7227	0.7315
	14	0.7225	0.7242	0.7261	0.7337
	16	0.7325	0.7358	0.7350	0.7361
	18	0.7289	0.7319	0.7311	0.7282

We train HMMs by tuning the sliding window (number of states) to set up the number of states and the length of the training sequences. The training is based on an EM algorithm, which iterates the computation of hidden parameters until the log-likelihood reaches the maximum value. Before digging into the evaluation of HMMs, we need to determine which length of training sequences we should consider to build models. To do so, we vary the length of sequences and take note of how it impacts the prediction ability of HMMs representing malware families. Table 5.5 illustrates the number of profiled malware families per HMM state and sequence length. By increasing the length of training sequences, we obtain fewer numbers of HMMs for malware families. This is due to the fact that some malware families do not have sequences of length greater than 2. It is thus impossible to create training data to model them. Intuitively, if we increase the length of training sequences (≥ 6), the number of HMMs will reduce. If we want to create HMMs for all malware families, we have to consider training sequences of length 2. With regards to detection, the cost of detecting 2 malicious flows is less expensive than detecting between 3 to 6 malicious flows. If we consider training sequences of length 2, we need to investigate two aspects: (1) the tradeoff between HMM expressiveness and learning effort, and (2) the uniqueness ratio of sequences per malware family. These issues are explained in what follows.

Table 5.5: Number of Malware Families per State and Sequence Length

		Sequence Length				
		2	3	4	5	6
HMM States	2	294	294	274	256	242
	4	294	277	274	254	245

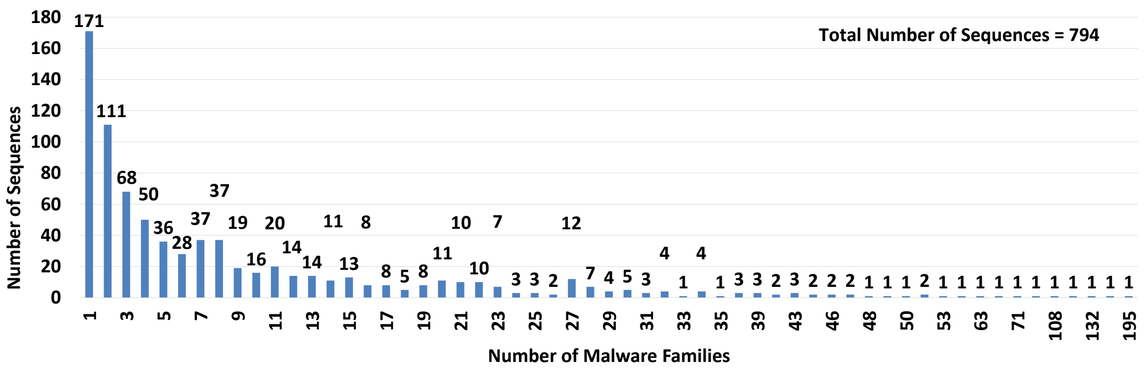


Figure 5.10: Uniqueness of Sequences

- *HMM expressiveness vs. HMM learning effort*: The former is meant to provide a high number of states to HMMs in order to generate more probabilistic HMM parameters with a greater ability to estimate potential sequences of malicious flows. However, increasing the sliding window to generate more states for HMMs results in generating more learning effort for HMMs. By varying the number of states from

2 to 4, the number of iterations increased for the majority of malware families. Table 5.6 shows the number of iterations per HMM configuration (2 states to 4 states, sequence length of 2 to 4). For 2 states HMMs obtained from training sequences of length 2 to 3, the number of iterations does not exceed 2. For 2 states HMMs obtained from training sequences of length 4, the number of iterations varies from 3 to 40. Similar results are shown for 4 states HMMs obtained from training sequences of length 2 to 4. Since the training sequence of length 2 allows profiling all malware families, it is recommended to use 2 states HMMs with training sequences of length 2 if we do not consider expressiveness of HMMs, or to use 4 states HMMs with training sequences of length 2 if we require more expressive HMMs.

Table 5.6: HMMs vs. Number of Iterations

	1	2	[3,20]	[21,40]	[41,60]	[61,200]
HMM 2-2	16	278	0	0	0	0
HMM 2-3	0	294	0	0	0	0
HMM 2-4	0	0	183	78	9	3
HMM 4-2	0	0	145	125	19	5
HMM 4-3	0	1	178	86	8	4
HMM 4-4	0	0	191	71	8	4

- *Length of training sequences*: Does limiting the length of training sequences to 2 impact the uniqueness of sequences per malware family? To answer this question, we test different sequences of length 2 on all malware family HMMs. Figure 5.10 illustrates the distribution of training sequences with the number of malware families (i.e., HMMs). We observe that approximately 21.5% of sequences are predicted by 1 malware family, and approximately 89% of sequences are predicted by at most

22 malware families. As such, we can conclude that the tradeoff between prediction and uniqueness is maintained since a big proportion of sequences are predicted by 22 HMMs over 294 HMMs.

Computational Complexity

In this section, we investigate computational complexity for different techniques used to non-DPI fingerprint malicious traffic. Computational complexity falls into:

- *Features Extraction*: In [117], the authors studied the computational complexity and memory requirements associated with flow features extraction in the context of classification. The authors claimed that extracting each feature from traffic is associated with a computational cost less than or equal to $O(n \times log_2\, n)$, and a memory footprint less than or equal to $O(n)$, where n is the number of packets in a flow used for extracting the feature. The total cost of extracting K features is bounded to $O(K \times n \times log_2\, n)$.
- *J48 Decision Tree*: J48 (its C4.5 Java implementation) has a training time complexity of $O(m \times n^2)$, where m is the size of the training data and n is the number of attributes [203]. Regarding the classification, the complexity is $O(n \times h)$, where h is the height of the tree and n is the number of instances [104].
- *Labeling*: To label unidirectional inbound and outbound flows, we use a K-means

RBS algorithm (a clustering partitional algorithm). The advantage of these algorithms is that they have a relatively low computational cost [240]. A 2-way clustering solution can be computed in time linear to the number of flows. In our case, the number of iterations used by the greedy refinement algorithm is less than 20. By assuming that the clusters are reasonably balanced during each bisection step, the time required to compute $n - 1$ bisections is $O(n \times log_2 n)$.

- *HMMs Convergence*: In our approach, we use the EM algorithm (also known as the Baum-Welch algorithm) [54]. It is based on the computation of forward and backward probabilities for each state and transition. The computing complexity is of order $O(n^2 \times t)$, where n is the number of states and t is the number of transitions [175]. However, in our experiments, we consider training HMMs with labeled sequences by varying the length of sequences. In addition, the EM algorithm has a computation which iterates until the maximization of the log-likelihood is satisfied. As such, the computing complexity is of order $O(n^2 \times t \times l \times i)$, where l is the length of sequences and i is the number of iterations.

5.5.2 DPI Approach

In the sequel, we present results obtained for DPI fingerprinting approach. We introduce: (1) classification and attribution setup, (2) classification results, and (3) computational complexity of this approach.

Classification and Attribution Setup

The MARFPCAT's algorithm parameters are based on the empirically-determined default setup detailed in [134, 136]. To perform classification, we load each pcap as a wave form signal. The latter encloses flows having both the header and payload sections. It is important to mention that all classification experiments are done through modules tuned with default parameters (if desired, they can be varied, but due to the overall large number of combinations, no parameters tuning has been considered). The default settings are picked up throughout MARF's lifetime, empirically and/or based on the related literature [134]. Hereafter, we provide a brief summary of the default parameters used for each module:

- The default quality of the recorded WAV files used in the experiment is 8000 Hz, mono, 2 bytes per sample, Pulse-Code Modulation (PCM) encoded.
- `LPC` – has 20 poles (and therefore 20 features), thus produces a vector of 20 features and a 128-element window.
- `FFT` – does 512×2-based FFT analysis (512 features).
- `MinMaxAmplitudes` – 50 smallest and 50 largest amplitudes (100 features).
- `MinkowskiDistance` – has a default of Minkowski factor $r = 4$.
- `FeatureExtractionAggregator` – concatenates the default processing of `FFT` and `LPC` (532 features).
- `DiffDistance` – has a default allowed error 0.0001 and a distance factor of 1.0.

- `HammingDistance` – has a default allowed error of 0.01 and a lenient double comparison mode.
- `CosineDistance` – has a default allowed error of 0.01 and a lenient double comparison mode.
- `NeuralNetwork` – has 32 output layer neurons (interpreted as a 32-bit integer n), a training constant of 1.0, an epoch number of 64, and a minimum error of 0.1. The number of input layer neurons is always equal to the number of incoming features f (the length of the feature vector), and the size h of the middle hidden layer is $h = |f - n|$; if $f = n$, then $h = f/2$. By default, the network is fully interconnected.

Classification Results

In this section, we summarize the results obtained per test case using NLP-processing of malicious network traces classification. We present various selected statistical measurements of the precision in recognizing different malware classes under different algorithm configurations. In addition, we use the "second guess" measure to test the hypothesis that if our first estimate of the class is incorrect, the next in-line one is probably correct. In Appendix A, we list the classification results sorted by fingerprinting accuracy. In Figure 5.11 and Figure 5.13, we set forth no-filtering classification results. In this case, no noise filtering is applied, which impacts positively in fingerprinting runtime. Figure 5.11 illustrates the corresponding summary per various algorithm combinations.

Figure 5.13 shows some malware families' classification results. It is noteworthy to mention that while the latter has overall low precision, many individual malware families are correctly identified. The low precision at the combination level is explained primarily by the "generic" malware class (the largest) that skewed the results and was not filtered in this experiment. The same experiments are replicated using wavelet transform-based filters in Figure 5.12 and Figure 5.14. Overall, we notice the same decline in precision as in the earlier filter-less solution, raising the question of whether pre-processing is really needed to quickly pre-classify a packet stream while lowering precision and hindering accuracy. It is also interesting to note that some malware classes (e.g., VBKrypt) are poorly identified in the first guess, but correctly in the second guess (illustrated by red spikes to the right of the graphs).

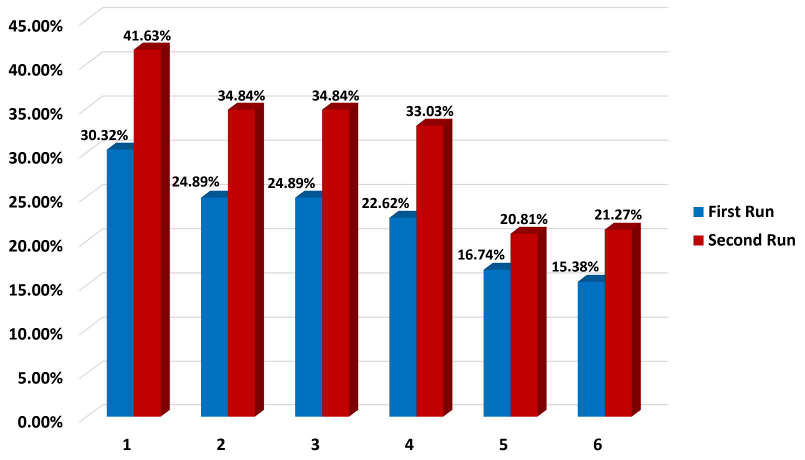


Figure 5.11: No-Filtering Malware Algorithms Results

The initial global scan results are listed in Table A.1, Table A.2, and Table A.3 (see Appendix A). Many of the malware families are nearly identified with an accuracy of 100%, often even using a single packet. We discover that the data feed had some malware

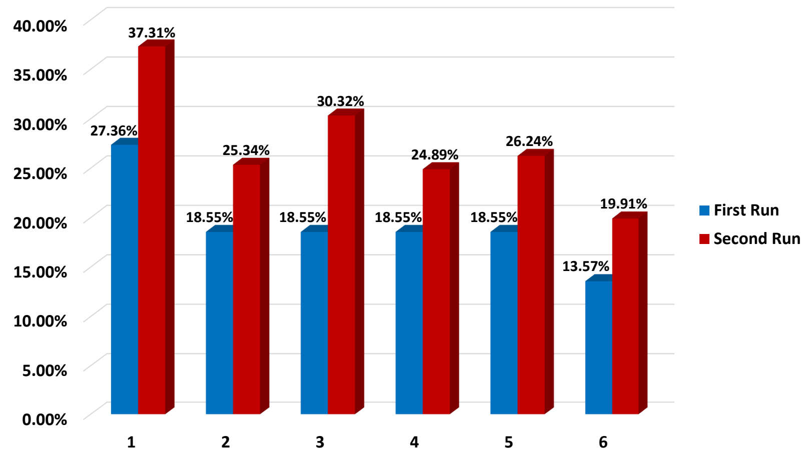


Figure 5.12: Wavelet Malware Algorithms Results

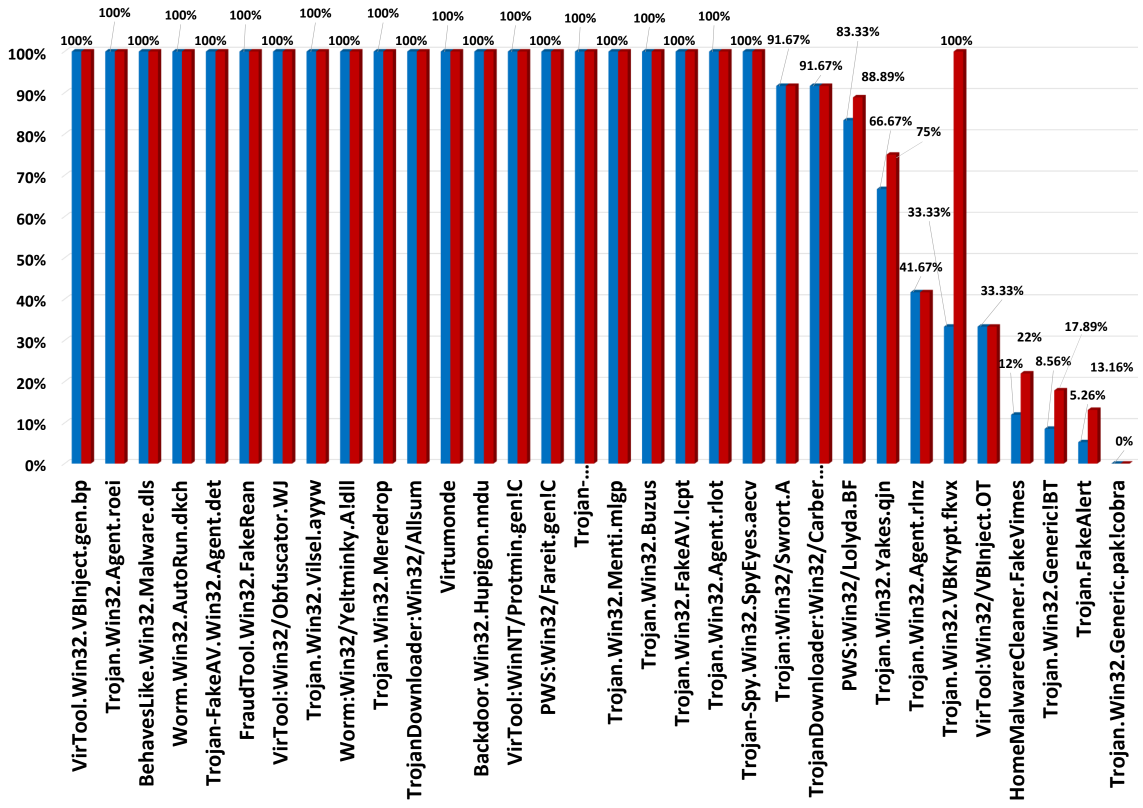


Figure 5.13: No-Filtering Malware Family Results

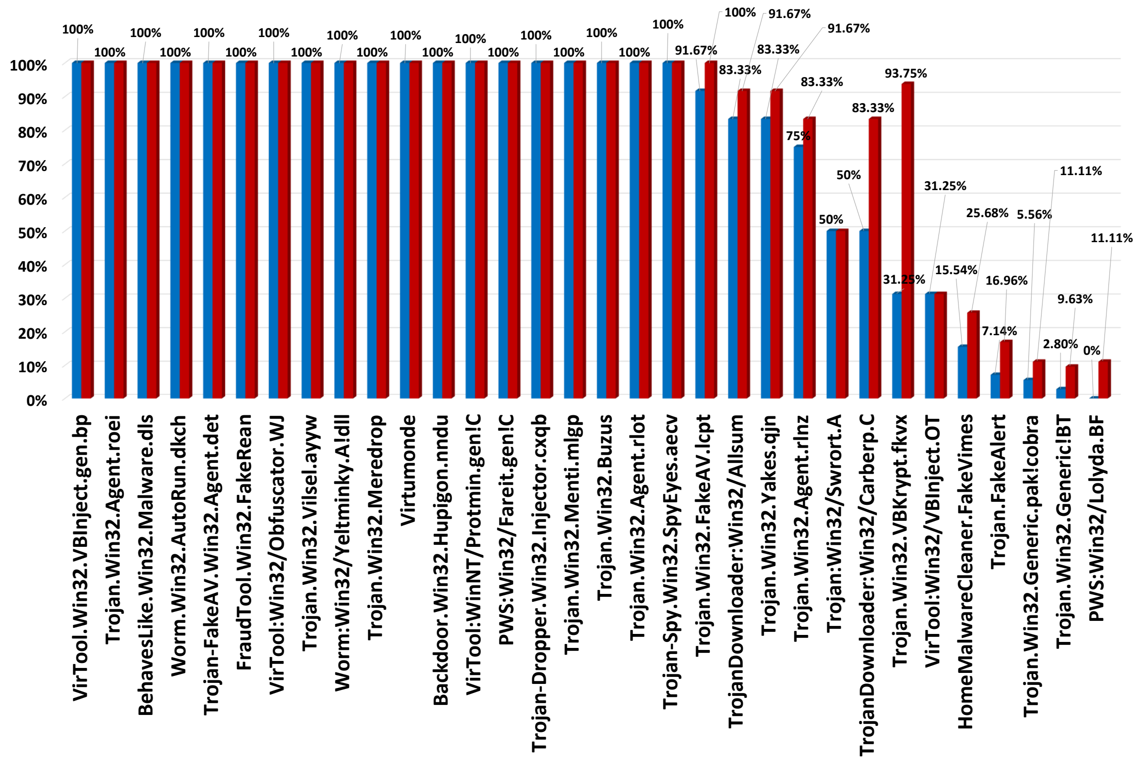


Figure 5.14: Wavelet Malware Family Results

classes labeled as “generic”. A lot of distinct malware families belong to such classes. The presence of such malware families results in noise and overfitting when training, which impacts negatively on the overall per-configuration (combination) precision. However, despite the presence of noise, many classes (771 of $1,063$) are classified with an accuracy of 100%. The rest of malware classes have less than 75%, dropping quickly to low classification accuracies, (e.g., *Virus:Win32/Viking.gen!B [generic]*, *PWS:Win32/Fareit.gen!C [generic]*, and *VirTool:Win32/Fcrypter.gen!A [generic]*, etc).

Computational Complexity

The computational complexity of the MARFPCAT data depends on the algorithms chosen at each stage of the pipeline. Most of them are one-dimensional processing modules with an average complexity of $O(n)$ where n is the number of the elements at each stage. Here is the breakdown for some of the tasks:

- Sample loading has to do with interpreting the pcap data in a wave form, which is a straightforward interpretation of every two bytes per an amplitude. Thus, it depends on the size of the pcap file in bytes b – $O(b/2)$.

- The pre-processing stage's complexity depends on the algorithm chosen. Raw has no processing, just passes data further, so the complexity is of $O(1)$. Normalization complexity is $O(n)$. FFT-based low-pass filters have the complexity of $O(2 \times O(FFT))$ to convert to time domain and back.

- Feature extraction depends on the chosen algorithms. Most common algorithms are FFT and LPC. LPC has a complexity of $O(n \times (\log_2 n)^2)$ in general. MinMax has a complexity of $O(2n \times \log n)$ (to sort and copy).

- Classification has the complexity of the chosen classifier, such as distance or similarity measures. Cosine similarity has a complexity of $O(n^2)$, but for normalized data, the complexity is $O(n)$. Euclidean, Chebyshev, and Diff distances have a complexity of $O(n)$, Minkowski distance has a complexity of $O(n^6)$; and Hamming distance has a complexity of $O(n + \log(n + 1))$ at the worst.

5.6 Discussion

In this section, we review the current results of this experimental work, including its advantages, disadvantages and practical implications. First, we discuss the positive and negative aspects of the non-DPI (Section 5.6.1) followed by the DPI (Section 5.6.2) approaches. The discussion encloses some observations noticed when performing experiments.

5.6.1 Non-DPI Fingerprinting

Advantages

In the sequel, we present the key advantages of the flow packet headers approach:

- *Classification accuracy*: Using packet header bidirectional flow attributes to classify malicious and benign traffic has shown excellent accuracy with low rates of false positives and negatives. J48 classifier has the ability to segregate malicious from benign traffic based on packet header attributes.

- *Independence from packet payloads*: All detection and attribution features are extracted from packet headers. The detection and attribution, therefore, avoid noisy data generated by encrypted traffic.

- *Generalization*: To segregate malicious from benign data, we use different benign datasets collected from different sources, namely, home networks, laboratory networks, corporation networks, and ISP networks. Different models achieve high accuracy in terms of differentiation between malicious and benign traffic. A 10-fold cross validation has been used to check whether the detection accuracy is maintained.

- *Detection attributes*: Decision trees in general are considered as a set of conditions involving the values of attributes. The classifier behaves as a white-box, where the attributes play roles in the decidability of flows maliciousness. The J48 decision tree models generate decision rules where the roots are usually features that highly overlap malicious and benign datasets. The distinctive features are mainly used as leaves to make final decisions on benign and malicious traffic. We notice, for instance, that forward and backward inter-arrival time values, duration of flow, and number of forward packets and bytes are good indicators that distinguish between malicious and benign traffic.

- *Labeling attributes*: Using inbound/outbound flow attributes for the purpose of traffic characterization is a good mean to create (sequences) patterns for malicious flows. These patterns are subjected to mining tools (HMMs) to attribute maliciousness to malware families.

- *Possibility to fingerprint zero day attack*: Characterizing the detection and attribution through flow features may provide opportunities to fingerprint unknown malware families that share identical network behavior with known malware families. For example, it has been shown in [171] that Citadel malware (appeared in 2013) is a variant of Zeus malware (appeared in 2009).

- *Decoupling between detection and attribution*: In general, this is considered a positive aspect in the sense that attribution is implicit to detection. The attribution does not impact the detection accuracy. However, there is a negative aspect of this decoupling that we discuss in the sequel.

Disadvantages

Hereafter, we present a list of issues identified in packet flows headers approach.

- *Datasets overfitting*: Decision trees that fit training and testing data too well may not be as good as it has been shown in our experiments. Overfitting trees can have a low re-substitution error but a high generalization error. As such, it is a must to consider more benign datasets to check whether the obtained models are subjected to generalization errors. J48 decision trees are static classifiers and are not resilient to additional noisy benign data (traffic). It is thus imperative to investigate the noise resiliency of obtained classifiers and to determine how we can build a committee modeling approach based on multi-decision trees.

- *Complexity*: Fingerprinting of maliciousness based on packet header flow features generates a computational complexity related to the extraction of features, the classification and clustering of feature' vectors, as well as the construction and sequencing of flows. For instance, we observed the following runtime for features extraction, models detection and labeling:

1. Bidirectional flow features extraction takes on average 0.94 seconds (0.042 seconds per feature).
2. Unidirectional flow features extraction takes on average 1.19 seconds (0.026 seconds per feature).
3. Detection:
   - Malicious vs. Home Model: 15 milliseconds per feature vector.
   - Malicious vs. SOHO Model: 16 milliseconds per feature vector.
   - Malicious vs. ISP Model: 21 milliseconds per feature vector.
   - Malicious vs. Private Model: 18 milliseconds per feature vector.
4. Labeling:
   - Inbound flows: The 16-k clustering solution takes about 7.298 seconds (0.1300 milliseconds per feature vector).
   - Outbound flows: The 18-k clustering solution takes about 9.556 seconds (0.1671 milliseconds per feature vector).

A deployment of such approach in a real-time traffic needs a traffic sampling technique since the computation of flow features on the fly is expensive. Moreover, detection and attribution models must response quickly to vectors of flow features created on sampled data. This means that we need to synchronize flow features extraction with detection and attribution.

- *Corroborating attribution*: The HMMs-based attribution is not mature. We need to establish an algorithm to limit the non-determinism of HMMs. This can be done by considering longer sequences when we have non-determinism between malware families. For example, with a malicious flows sequence of length 2 that is classified by ten 2 sequences trained HMMs, we can consider a potential third detected flow to create a new sequence of length 3 and classify it with 3 sequences trained HMMs. As such, ten 2 sequences trained HMMs play the role of filters, whereas the 3 sequences trained HMMs limit the number of malware family attribution possibilities.

- *Decoupling between detection and attribution*: In a way, this is a double-edged sword. The negative aspect lies in the fact that it generates deployment challenges, which break into flows sampling, flows construction, and strong detection to implicitly obtain a good attribution. Thus, it is necessary to conduct a thorough analysis to deploy this Non-DPI fingerprinting solution. The analysis should cope with the following issue: how to sample data from network to be tested against detection and attribution models.

5.6.2 DPI Fingerprinting

Advantages

In the sequel, we present the key advantages of the DPI approach:

- *Performance*: The DPI approach has shown an ability to learn and classify relatively quickly than flow packet headers approach. For instance, results shown in Table A.2 took from 58 milliseconds to 598 milliseconds per pcap file. The complete run considering all algorithms' combinations, including training and testing phases took 27 minutes and 74 seconds. Some results go as low as below 10 milliseconds per pcap file (including loading, pre-processing, feature extraction, and classification). A complete training on an algorithm combination was 1 to 3 seconds depending on the algorithm. Detailed performance statistics from the log files can be released depending on the need and appropriateness at an external resource, such as arXiv [118].

- *Learning scalability*: Given the ability shown in training runtime, DPI approach has the flexibility to learn on a large knowledge base to test on known and unknown cases as well as label them. The results shown in terms of runtime allow to design and integrate easily an online learning system, where the detection and attribution can be improved by time. This approach can be used to quickly pre-scan projects for further analysis by humans or other tools that do in-depth maliciousness analysis.

- *Flexibility*: Tuning algorithms' combinations allows the selection of the best learning process for malware classes. Accordingly, we can identify appropriate algorithm combinations that maintain the tradeoff between accuracy and runtime. This approach can be used on any target malware without modifications to the methodology.

- *Pluggability*: The developed tool, namely, MARFPCAT, can learn from binary signatures obtained from other intrusion detection systems (e.g, Snort [195], Bro [161], etc.). In addition, since it is an open-source, it can be easily plugged to existing firewalls or intrusion detection systems.

Disadvantages

Hereafter, we list of the most prominent issues related to the DPI approach. Some are more "permanent", while others are solvable and intended to be addressed in a future work.

- *Dependency*: The detection accuracy depends on the quality of the collected knowledge base (see Section 5.4.2). The annotation of pcap indexes are done manually, hence, it is prone to errors.

- *Accuracy*: Despite the fact that some malware families are identified with a high accuracy, MARFPCAT has shown limited accuracy for some malware families, especially the ones clustered as being "generic".

- *Fuzziness*: DPI fingerprinting has many algorithms' combinations (currently $\approx$ 1800 permutations), which try to get the best top N. This can lead to incoherence in some classification cases when there is a shift from a combination to another.

5.6.3 Summary

Our core finding is that the two approaches are not necessarily in competition with each other, but are rather complementary with DPI being much faster (no parsing and picking out select headers; in addition, signal processing techniques and related classifiers were simpler and more efficient in comparison with the flow packet headers approach). The DPI approach can work with either one or two packets already and does not depend on benign traffic learning (which, if it did, would be like a noise signal), whereas the header-based flow approach strictly requires a flow before it can classify. Thus, the DPI approach can prioritize classification targets, specifically for the headers-based approach (and go deeper as necessary). While listening first on the network interface, MARFPCAT can predict or hint to maliciousness, whereas flow packet headers can increase subsequently the confidence in maliciousness fingerprinting.

5.7 Conclusion

In this work, we presented a research effort dedicated to fingerprinting maliciousness at the traffic level. The maliciousness fingerprinting falls into: NLP/wavelets Deep Packet Inspection (DPI) and flow packet headers. Moreover, we produced a comparison between these two approaches.

Regarding the DPI approach, considering results shown by MARFCAT in the classification of vulnerable code, we used NLP and wavelets classification of signals techniques to fingerprint maliciousness. Despite showing some problems in classifying the generic malware families, it managed to show a large scalability and accuracy for less noisy malicious traffic. As a result, we released a MARFPCAT alpha version, the MARFCAT's predecessor, as open-source [135]. The distributed demand-driven version of MARFPCAT is available in GIPSY open source repository.

Regarding flow packet headers approach, we employed several supervised machine learning algorithms, namely, J48, Boosted J48, Naïve Bayesian, Boosted Naïve Bayesian, and SVM in order to classify malicious and non-malicious traffic. The aforementioned learning algorithms were used to build classification models. Thus far, the results show that the J48 and Boosted J48 algorithms performed better than other algorithms. They reached over 99% precision with a rate of false positives less than 1%. In summary, we illustrated that it is possible to detect malicious traffic and differentiate it from non-malicious traffic by using attributes extracted from packets. This is a preliminary result toward the classification of malicious traffic at the network level. Therefore, we aim to investigate the degree to which our classification results are generalizable to a wide class of representative networks. Despite the fact that fingerprinting maliciousness at the IP level is important, a focus must be shown to another network layer, namely, the application layer. Among numerous application layer protocols, DNS acts as a platform that binds internally host-names and IPs, and externally web-surfers to existing domains. As it has been demonstrated in Chapter 4, IPs and domains are cornerstones of different cyber-threat infrastructures. As such, there is a need to investigate DNS protocol streams for the purpose of identifying its misuses and indicators of compromise. In the next chapter, we will describe the big data system used to cope with the nature of passive DNS streams for the purpose of anomaly detection.

Chapter 6

Near-Real-Time and Scalable Detection of Anomalies in Passive DNS Streams

6.1 Overview

In this chapter, we present a near-real time anomalies identification system in passive DNS streams. By anomalies, we mean suspicious domain names, DNS records misuses, fast-fluxing malicious networks, where the change of IP addresses, geo-locations (countries and cities), and short Time To Live (TTL) requests are observed. The system is integrated on top of a high computational clustering solution namely, Apache Spark, to handle streams of passive DNS logs on near real-time. In addition, we corroborate the system with the ability to monitor specific DNS profiles for organizations. The remainder of this chapter is as follows: Passive DNS anomaly detection approach is discussed in

Section 6.2. The experimental results are reported in Section 6.3. Finally, some concluding remarks on this work, together with a discussion of future research, are provided in Section 6.4.

6.2 Passive DNS Anomalies and Abuse Detection

6.2.1 Approach

DNS protocol has been turned to a platform to perpetrate malicious activities. Thus, there has been a desideratum in the generation of cyber-threat intelligence based on DNS traffic replica, known as passive DNS. The latter is a technique, defined by Weimer [218], which captures inter-server DNS messages through sensors. These messages are forwarded to a collection service for further analysis. Some research efforts [22, 23, 29, 30, 164] put an emphasis on using Passive DNS to detect DNS abuses and malicious activities. In spite of interesting results obtained by proposed systems in the aforesaid works, they have not integrated all-in-one solution to gather threat-intelligence. They use mainly classification techniques to segregate malicious domains from benign domains or to detect fast-flux malicious services. In essence, we aim to address the generation of cyber-threat intelligence from passive DNS by trying to answer the following questions: (1) How to transform intensive real-time passive DNS feeds into cyber-threat intelligence? (2) What are the techniques that ensure scalability and allow the identification of anomalies in passive DNS?

We target to design and integrate a system that monitors all domains and their DNS records, observed in passive DNS stream of data. The main intent is to deploy a system that detects in a near real-time, potential anomalies and abuses of DNS protocol observed on captured logs. However, the design and deployment of our system has to overcome some challenges related mainly to the high volume of data. Thus, we need to consider scalable techniques to monitor DNS ecosystem for the purpose of anomalies and abuses identification, as well as a reliable storage system to archive results. The system aims to: (1) Aggregate real-time data for the purpose of online analysis. (2) Segregate different types of DNS record messages observed on data streams. (3) Identify suspicious domain names and alias. (4) Extract features representing time series analytic. (5) Detect anomalies based on collected features. (6) Pinpoint abuses of DNS records. (7) Correlate with other sources of cyber-threat intelligence. (8) Monitor and archive historical DNS activities observed in some organizations. In the sequel, we describe the system architecture.

The emergence of big data processing frameworks offers encouraging approaches to collect, retrieve, and analyze intelligence out of different sources like malware feeds, spam-traps, Darknet and passive DNS. Accordingly, it is possible to discover threats in near-real time. Motivated by contributions reported in [120, 184, 222, 231] that used Apache Spark for big data analysis, we decide to employ it to monitor the huge load of passive DNS data to extract anomalies. Such framework is a promising, multi-purpose data processing designed for intensive in-memory and distributed clustering computations. It emphasizes on improving performance of applications that cannot be expressed efficiently as acyclic data flows, where a working set of data across multiple parallel operations [233]. It includes two use-cases, namely, iterative jobs and interactive computing, where Hadoop [71] is deficient. A key asset of Spark lies in introducing the Resilient Distributed Dataset (RDD) [232]. RDDs are known to ensure the abstraction of data such that large data sets can be cached effectively in memory or disk. They represent immutable collection of objects grouped into partitions. Spark uses RDDs to allow re-usability of memory cached objects, which improves significantly performance in work-flow execution.

In addition to in-memory and disk caching, Spark supports many data abstractions, namely, graphs, streaming logs (e.g., Twitter feeds), databases (e.g., MySQL, Cassandra) and hadoop data formats. Moreover, Spark has a mature programming model, which has been initially integrated by SCALA programming language [64], then wrapped to other languages, i.e., object-oriented programming Java [155], script programming Python [72], and statistical computing language R [68]. Spark provides programmers with the ability to: (1) Construct RDDs from files in a shared file system, (2) divide collections into slices that can be sent to multiple nodes, resulting in computation parallelism, (3) smooth transformation of data from one type to another (e.g, a log to a mapping object) and, (4) alter persistence of objects through two actions, namely, the cache action, which leaves the dataset lazy but kept in memory for re-usability, and, save action, which dumps data into a distributed file-system like Hadoop File System (HDFS).

Spark supports also several parallel operations, namely, *Filter*, *Collect*, *Reduce*, *Map-Reduce* and *Foreach*. The *Filter* operation allows eliminating items into collections, that do not satisfy a Boolean predicate function. The *Collect* operation sends all elements of a dataset to a driver for a parallel gathering of items into a collection (e.g., arrays, lists, etc.). The *Reduce* operation combines dataset elements through an associative function. The *Map-Reduce*, known also as grouped reduce function, which allows to map datasets to common mapping objects like tuples and reduce them by a key entry and an associative function. The *Foreach* operation allows a streaming loop through collection to execute functions like exporting results to databases or copying them into shared variables in a program. In the sequel, we detail the system architecture described in Figure 6.1.

6.2.2 System Architecture

The passive DNS anomaly detection system is meant to monitor DNS replica logs sent on a minute-by-minute basis. These logs contain passive DNS entries enclosing information about different record types (e.g., "A", "AAAA", "CNAME", "TXT", etc). DNS anomaly detection system falls into the following components: (1) Dispatcher, (2) Record Extraction, (3) Geo-location, (4) Prediction per Partial Matching (PPM) detection and, (5) Record misuses filter.

Dispatcher

The dispatcher component plays the role of transforming passive DNS logs into streams of RDDs. These RDDs are dispatched into four categories: (1) RDDs containing A records,

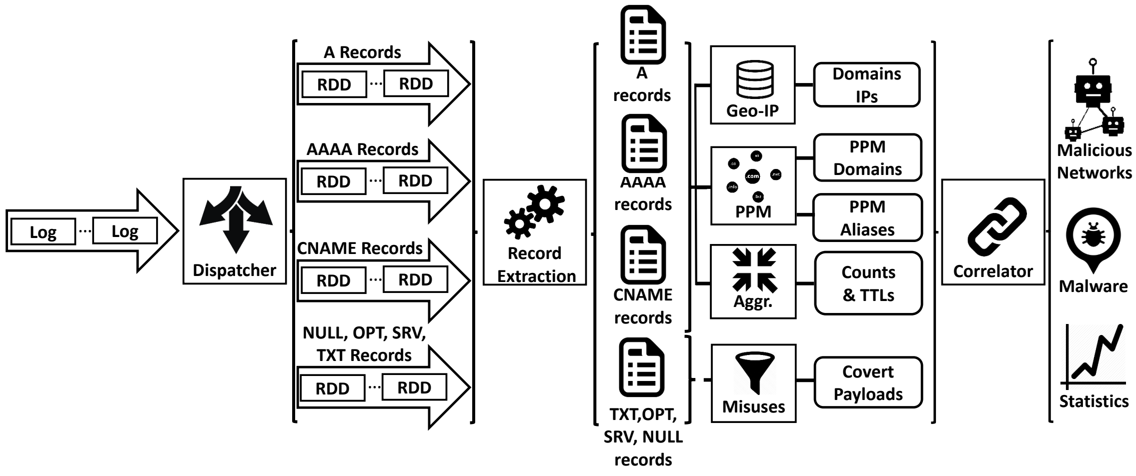


Figure 6.1: DNS Anomaly Detection Architecture

(2) RDDs containing AAAA records, (3) RDDs containing CNAME records and, (5) RDDs containing records (e.g., “TXT”, “OPT”, “SRV”, “NULL”) that are candidates for DNS abuses like covert channels and tunneling. The dispatcher uses Spark filter action to check the type of the record and push it to RDDs.

Record Extraction

This component plays the role of the bridge that takes RDD streams as inputs and generate tuples labeled by record type. Record extraction use the map operation to get different tuples needed for components that identify different anomalies.

Geo-location

DNS protocol allows a domain to be mapped to different IPs. Usually, DNS server answer consists of DNS “A” records, when the host maps to IPv4 addresses or DNS “AAAA” records, when the host maps to IPv6 addresses. Malicious domains resolve to infected machines located in different Autonomous Systems (ASNs), countries, and regions. Therefore, the attackers build botnets that spread worldwide; and each node with these botnets may play the role of a C&C server. With insight, we use a geo-location database, namely, MaxMind [4] to geo-locate different IPs collected from “A” records (IPv4 addresses). We plan for a near future to use it also for “AAAA” records (IPv6 addresses). Despite the fact that benign domain names (e.g., google.com, yahoo.com, etc.) use IP addresses located in many countries and cities, our system uses, on the fly, a white-list to filter such domains. We focus only on domains that do not appear in the white-list. The system has a conservative approach, since it quarantines all domains changing frequently cities and countries. Collected information is cross-validated with other features, namely, other IP-based features and TTL-based features described in the rest of the components. To detect changes of countries and cities, we collect them into immutable sets labeled by domains. We use Spark filter action to check if these collections have a cardinality higher than one.

PPM Detection

Malicious programs use a technique called domain fluxing, where malware samples change the fully qualified domain names. They employ such domain names as C&Cs, depot of stolen information, spam campaigns, bitcoining and infection vectors (drive by download malware). The easiest way to achieve domain fluxing is to use a domain generation algorithm (DGA) that dynamically generates random domain strings for botnet communication, spamming, etc. In addition to domain fluxing, malware misuse canonical name ("CNAME") records to manage botnets. Usually, they use suspicious domains to point to alias domains, which look like benign.

In order to detect domain fluxing activities and misuse of "CNAME" records, we use a technique proposed by Begleiter et al. [27] to segregate between benign and malicious domains. As such, we use Prediction per Partial Matching (PPM) algorithm, which belongs to Variable order Markov Model (VoMM) algorithms [27]. It is an algorithm that predicts a symbol based on previous symbols. PPM algorithm has two phases, which are: training phase and prediction phase. At the training phase, it builds a structure, namely, digital tree (Trie), which stores the sub-sequences of the training sequences and the counts of the symbols that appear after them. At the prediction phase, it calculates the probability estimation of sub-sequences of a new sequence and compares it with the estimation of the training set. Our abnormal domains detector computes the probability of a domain name to verify whether it is benign or not. In our case, the training dataset is composed of different domain white-lists (e.g., Alexa top one million domains [1], Quantcast US domains ranking [168]). PPM algorithm has two steps to build the training sequences. First, it reverses each domain name. Second, it adds 2 delimiters at the beginning of each reversed domain name. For example, *google.com* is reversed to *##moc.elgoog*. The delimiters are used to separate domains and not let them being concatenated, which avoids appearance of noisy context in the training sequence. The domains are reversed to let the training be based on domain TLDs. Once the sequences are ready, the PPM classifier is built by storing the count of symbols that occurs after sub-sequences of a specific length (distance) in the training sequences.

Then, the classifier is used to calculate the average per symbol probability of the 50% of the training sequences, which are uniformly sampled. The computed average is considered as a pivot score. The reason behind choosing half of the training sequences lies in the fact that the pivot score should represent the average benign domain name. To determine if a new domain is abnormal or not, we first reverse the domain name and add the delimiters. After that, we use the PPM classifier to compute the average per symbol probability. Then, we calculate the ratio of the average per symbol probability and the pivot score. If the ratio is less than a threshold (0.8), the domains is considered as abnormal. Otherwise, it is considered as benign.

Aggregation

This component has the ability to monitor attributes of passive DNS records for the purpose of anomaly detection. The gathering of streams is done every minute since records are received once a minute. However, we use Spark streaming capability to aggregate statistics of attributes during a predefined time window (2 hours for specific IPs and top level domains, 1 hour for the rest of passive DNS records) to collect sufficient samples of attributes, where we can apply anomaly detection techniques to segregate between benign and malicious behaviors. Thanks to the map-reduce Spark action, we reduce tuples generated by the record extraction component (see Section 6.2.2 and Figure 6.1) and group them by domain names. The aggregation is done on the following attributes:

Number of Queries The number of queries that target a specific domain name is approximately in the same range over a specific period of time. This feature is a good indicator to detect the following anomalies: (1) If there is an IP address that maps to many suspicious domains (fluxing domains), it is more likely that these domains have abrupt changes in their "A" or "AAAA" queries number. Domains fluxing is a phenomenon where a domain is usually less active for a certain period of time and suddenly exhibits abrupt increase in the number of queries then followed by an abrupt decrease. In [29, 30], Bilge *et al* stipulates that such behavior is an indicator of domain fluxing, a technique used by cyber-criminals to use many machine generated domains to change C&Cs and proxies. This technique makes cyber-threat infrastructures more robust and flexible against takedowns. To detect abrupt changes, we use Chauvenet test [45]; the latter detects outliers by rejecting time series points that do not fit the normal distribution probability. The test has two inputs, the time series data and a significance level α (set to 0.5 in Chauvenet Criterion Test). Initially, the mean of time series data (M) is computed as well as the standard deviation (S). Then, we create a normal distribution using the mean and standard deviation. For each value in time series data, we compute its cumulative probability from the normal distribution (P). Thus, the criterion value (P') is computed by multiplying the inverse probability of P by 2. If the product of the criterion value (P') and the length of the time series data is less than the significance level, then the time series value is considered as an outlier. Algorithm 6 summarizes the different computations' steps done to identify outliers for queries number.

Algorithm 6 Chauvenet Test

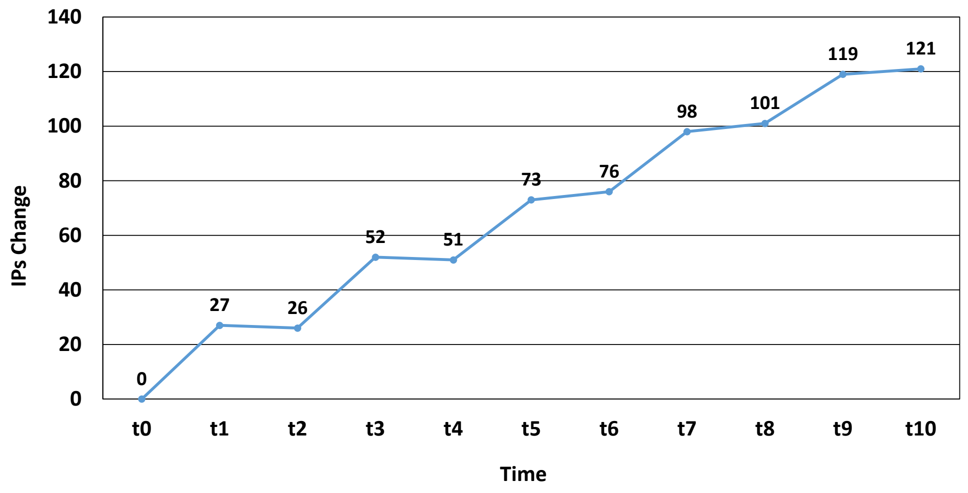
```
Input: $Data$
Input: $\alpha = 0.5$
Output: $Outliers$
  $M = Mean(Data)$
  $S = StDev(Data)$
  $D = NormalDist(M, S)$
  for $V \in Data$ do
    $P$=$CumulativeProb(D, V)$
    $P' = 2 \times (1 - P)$
    if $P' \times Length(Data) < \alpha$ then
      $Outliers.Append(V)$
    end if
  end for
  return $Outliers$
```

Number of IP changes per domain Each DNS response from "A" or "AAAA" resource records contain IP addresses that map to a domain name. Usually, malicious domains map to different IP addresses. We use Spark map action to collect immutable sets of IPs on 2 minutes slide window time. Then, we use another Spark map-reduce action to index different immutable sets with domains and transform them to time series. This is done every hour. To generate the time series, we compute the cardinality of items (IPs) observed in a set that appears at time $t + 1$ and not appearing a time t. Although unusual, some benign domains use many IPs to load balance queries sent by Internet users. To overcome this issue, we use a white-list to filter false positives. In addition, we create a scoring function based on a time series observed per domain. Domains having high scores are considered to be investigated. Domains having low scores are discarded. We score the time series as follows: if the time series contain only zero values, they are omitted. For each time series, we compute: (1) the number of positive shifts ($shifts$) between values observed at times t and $t + 1$, (2) the number of values that are greater or equal to the average of time series' values ($avgs$), and (3) the number of values that are greater than 0 ($vals$). Let T be a time series, the IP changes scoring function is computed by Algorithm 7. Figure 6.2 illustrates how a score is computed for a domain representing a bitcoin network. We observe from the example that the score is high (0.75), which means that it is worth being investigated.

Algorithm 7 IP Changes Score Function

```
Input: $T$
Output: $Score$
  $M = Mean(T)$
  $S = Size(T)$
  for $i := 0; i < S; i++$ do
      if $i \neq S - 1$ & $T[i] < T[i+1]$ then
          $shifts++$
      end if
      if $T[i] > M$ then
          $avgs++$
      end if
      if $T[i] > 0$ then
          $vals++$
      end if
  end for
  $Score := \frac{1}{3} \times [(shifts/S - 1) + ((avgs + vals)/S)]$
  return $Score$
```

TTL values Each DNS response has a TLL (Time to live) field. It indicates how long the DNS record will be cached. Most of the benign domains specify the TTL with high

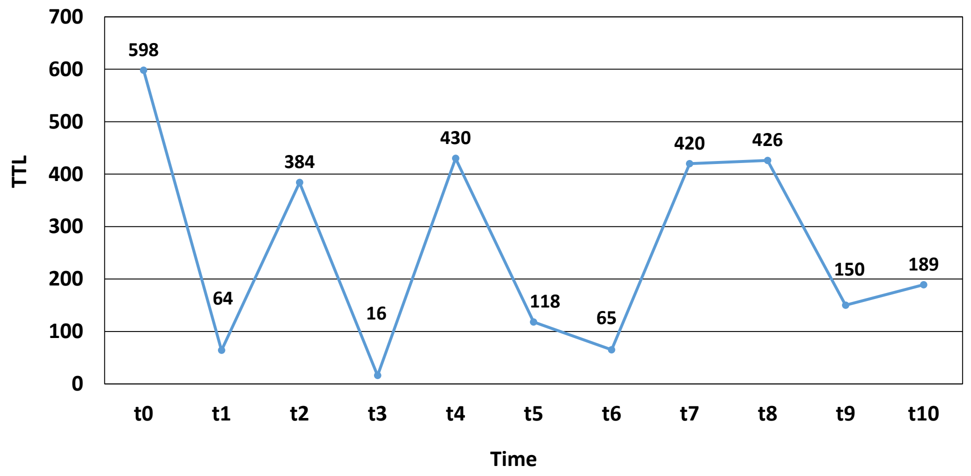


Shifts = 8
Values>0= 10
Size=11

Average=67.63
Values>=Average= 6
Score=0.33*[(8/10)+(6+10)/11]≈0.75

Figure 6.2: Example of Number of IPs Change

values as one day or more to get the benefit of DNS caching [11]. However, Content Delivery Networks (CDNs) systems specify the TTL to low values. They combine a low TTL with Round Robin technique [10] in order to make the systems scalable and available all the time. Round Robin is a technique that provides many IP addresses rather than a single IP address. In case one of the IP addresses is not available, another IP address will be available since the TLL value is low. Many malicious systems such as fast-flux systems use the round-robin technique with the low TTL to prevent DNS Blacklists [61] detection. Low TTL values observation has been used by many techniques to detect Fast-Flux systems besides other observations such as number of distinct IP addresses. In [29], Bilge *et al.* explained that some malicious networks change TTL values frequently. In these networks, some infected machines are selected to be proxies and C&Cs, and TTL values are assigned with different levels of priorities. There is a high probability that a proxy running on an ADSL IP is less reliable than a proxy running on a university or an organization server. Therefore, low TTL values are assigned to dynamic IPs (home Internet connections) and high TTL values are assigned to static IPs (Servers). The authors corroborate this assumptions with TTL values observed in *Conficker* botnet domains. In order to detect malicious networks, we use a TTL scoring function, which relies on the following parameters: (1) Unchanged low TTL values, (2) the usage ratio of specific TTL ranges, (3) the number of TTL changes. The TTL scoring function is computed by Algorithm 8. This algorithm has two parts. The first part checks if TTL values with time series are the same. If so, we pick time series that have TTLs less or equal to one hour (3600 seconds). We assign the score based on TTL values that belong to different ranges illustrated in Algorithm 8. The second part deals with time series that contain different TTL values. In this case, we compute the ratio of TLL values per range as well as the number of TTL value changes. Regarding the ratio of TTL values per range, we multiply it with a priority number. The lower are TTL values in a range, the higher is the priority number. For instance, the range $[0, 1]$ has a high priority number (6) since this range is an indicator of an intensive fast-flux activity. The range $]3600, \infty[$ has the lower priority number, since the TTL values are more than one hour. The first part of the score is computed by summing different ratios multiplied by their priority number. The result is divided by six to normalize it to a values between 0 and 1. The second part consists of computing the number of TTL value changes observed in a time series. The final score is the sum of both parts divided by two. Figure 6.3 illustrates a TTL score computation done on a suspicious domain.



Size=11
TTL changes = 10/11 ≈ 0.91
TTL ranges = (6*0)+(5*1)+(4*5)+(3*5)+(2*0)+(1*0)/(6*11) ≈ 0.61
Score=(0.91+0.61)/2≈0.76

Figure 6.3: TTLs Change Example

Record Misuses Filter

The democratization of DNS protocol has pushed cyber-criminals to abuse it and use it as a carrier for communication between malware infecting machines and remote bot-masters or proxies. Being inspired by the emergence of DNS tunneling tools (e.g., iodine [110], NSTX [206], OzymanDNS [100], Heyoka [173], etc.), cyber-criminals have been misusing DNS records, namely, “TXT”,“SRV”,“OPT”, “NULL” and “ANY”, to perpetrate malicious activities. In the sequel, we present examples how these DNS records can be abused.

Algorithm 8 TTL Score Function

```
Input: $T$,$cpt$
Output: $Score$
  $Freq := Frequency(T, T[0])$
  $S := Size(T)$
  if $S = Freq$ then
      if $T[0] \in [0, 1]$ then
          $Score := 1.0$
      else if $T[0] \in ]1, 60]$ then
          $Score := 0.9$
      else if $T[0] \in ]60, 300]$ then
          $Score := 0.8$
      else if $T[0] \in ]300, 900]$ then
          $Score := 0.7$
      else if $T[0] \in ]300, 3600]$ then
          $Score := 0.6$
      else
          $Score := 0.0$
      end if
  else
      for $i := 0; i < S; i++$ do
          if $T[i] \in [0, 1]$ then
              $cpt[6]++$
          else if $T[i] \in ]1, 60]$ then
              $cpt[5]++$
          else if $T[i] \in ]60, 300]$ then
              $cpt[4]++$
          else if $T[i] \in ]300, 900]$ then
              $cpt[3]++$
          else if $T[i] \in ]300, 3600]$ then
              $cpt[2]++$
          else
              $cpt[1]++$
          end if
          if $i + 1 \neq S \& T[i] \neq T[i + 1]$ then
              $changes++$
          end if
      end for
      $RatioChanges := changes/(S - 1)$
      $RatioRanges := \frac{1}{6 \times S} \times \sum_{j=1}^{6} j \times cpt[j]$
      $Score := \frac{1}{2} \times (RatioChanges + RatioRanges)$
  end if
  return $Score$
```

DNS tunneling Although being considered as a benign service provided online or by customized tools, this technique is a good artifact to exfiltrate data from networks that permit traffic to be sent only through a trusted server or proxy. By having a moderated bandwidth (110 Kilobytes per second) and latency (150 Milliseconds) [213], attackers considered DNS tunneling as a good medium to send blocked IP traffic through and conduct stealthy communications between bot masters. In [190], Ed Skoudis claimed that DNS tunneling malware is among the most dangerous attacks.

Malware covert channels In [58], the authors dissected the modus-operandi of Feederbot botnet. Malware belonging to Feederbot family exfiltrates data within DNS query sub-domain labels and infiltrates attack payloads in DNS response packets. To detect DNS traffic generated by Feederbot botnet, the authors defined empirically a set of features that span over: record data features and behavioral communication features. Based on these features, they adapted an hybrid approach (clustering & classification techniques) to detect malicious DNS traffic. In [145], Mullaney introduced another malware family, namely, Morto, which uses a more resilient method to exchange communication through covert DNS channels. Morto botnet sends a limited amount of payloads, which makes its distribution stealthier in comparison with Feederbot botnet.

DNS malicious responses Attackers have put forward tools to send malicious responses in reply to DNS queries in order to test if DNS look-up servers are vulnerable. For instance, dnsxss [192] is a tool that returns a string containing JavaScript to "MX", "CNAME", "NS", and "TXT" requests. By looking at Passive DNS stream, we have found a lot of domains having "TXT" records containing script and frame tags. If a vulnerable server does not sanitize "TXT" record data returned by such domains, XSS attack is performed leading to some abnormal behavior in the server side.

Indicators of DNS DDoS attacks In [43], the authors stipulated that "ANY" record is usually used in amplification DDoS attacks since it has an amplification factor of 52 since it replies with a response packet with $3,336$ bytes to a request packet of 64 bytes. In [66], the authors monitored Darknet for the purpose of inferring DDoS attacks. They observed that "ANY" records are in order of 52.23% of observed records involved in DDoS attacks during a period of three months. In the prevailing of these facts, we decide to monitor "ANY" records observed in passive DNS stream of data.

To detect record misuses, we used Spark filter functions on tuples. For instance, "TXT" record is used to publish email sender policies associated with domains (e.g., Sender Policy Framework (SPF) [126], Domain Keys Identified Mail (DKIM) [63], Domain-based Message Authentication, Reporting and Conformance (DMARC) [125]), or the verification identifier of different search engines like Google and Yandex. In order to detect misuse of "TXT" records, we create patterns that identify benign usage of "TXT" record data. The "SRV" record [12] is used to publish services (service protocols, e.g, LDAP, Minecraft, SIP, Skype for corporations, etc.) for domains. Based on observations done on "SRV" record data, we create patterns that identify if the record data is suspicious or not. In addition, we capture all passive DNS entries that have "OPT", "NULL" or "ANY" as a record type. The reason behind doing so lies in the fact that such records are rare and can be indicators of compromise.

6.3 Experimental Results

6.3.1 Application Performance

Our passive DNS anomalies identification system has been deployed on a cluster of three servers. The first server is dedicated to detect PPM domains, “TXT”,“SRV”,“OPT”, “NULL” and “ANY” record misuses. The second one is committed to monitor countries, cities and IP changes, as well as TTL values per domains. The third server is employed to observe PPM domains, countries, cities and IP changes, as well as TTL values associated with specific top level domains and IPs. All the servers run Debian Operation System version 7.8. In each server, we use instances of a document-based database, namely, MongoDB [142] to store collections of data. The servers have the following characteristics:

- Server 1: Dell Poweredge T410, 24 CPU cores, 64Gb memory and 3.8Tb space.
- Server 2: HP Proliant DL580 G7, 48 CPU cores, 125Gb memory and 4.5Tb space.
- Server 3: SuperMicro, 48 CPU cores, 125Gb memory and 4.0Tb space.

Hereafter, we provide performance results, which falls into: CPU usage, memory consumption and time delay. The latter means the sum of scheduling time and processing times for batches. We use JavaVisual VM [98] to monitor the CPU usage and the memory consumption and Spark user interface to observe the processing time delay [70]. We sampled approximately 18 hours logs of the CPU usage, the memory consumption and the processing delay time for all servers. We show through these benchmarks that Spark tool has the ability to cope with passive DNS streams since it uses memory resource to complete different computations. In the sequel, we provide few observations done on collected benchmarks.

PPM Detection and Record Misuses

We dedicate the first server to detect PPM domains, record misuses of all passive DNS records. In the sequel, we describe the application performance in terms of CPU usage, memory consumption and processing delay time.

CPU Usage We observe for the PPM detection and record misuses application a moderate usage of CPU (see Figure 6.4). This is due to the fact that the batches are done every minutes and no aggregation is done. In this case, Spark uses a filtering capability to segregate between benign and suspicious tuples collected from RDD streams. However, we observe occasionally spikes in CPU usage. This is due to heavy loads of data, which can reach 2 Gb of passive DNS entries on a minute-by-minute basis.

Memory Regarding memory consumption, we notice that the memory is extensively used. This is due to the high number of tuples that are loaded for processing. The garbage collection is triggered periodically to avoid memory usage problems. This is illustrated in

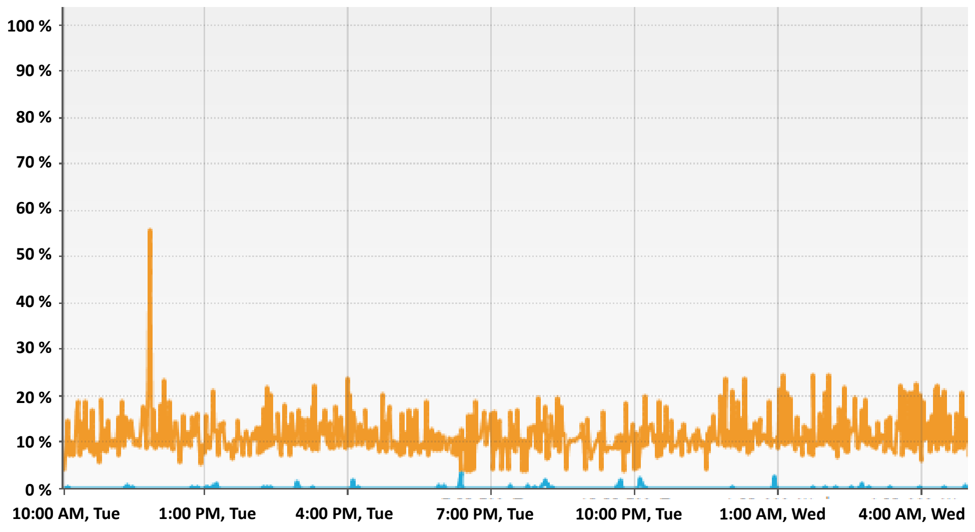


Figure 6.4: Server 1 CPU Usage

Figure 6.5, where the garbage collection is done twice to three times every 3 hours.

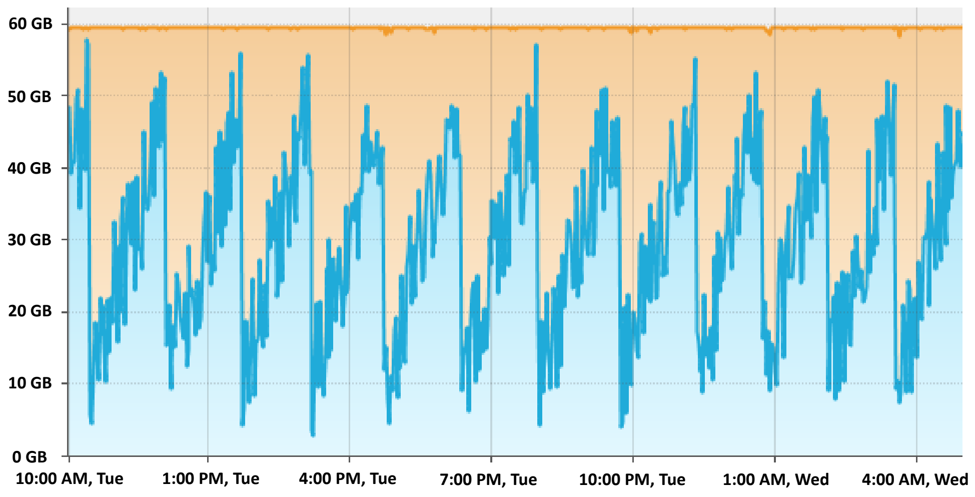


Figure 6.5: Server 1 Memory Consumption

Delay Time On sampled data, we observe that delay time may differ from a batch to another. This is due to the volatility of passive DNS records' number, which depends on how much data is received once a minute. The delay varies from 1 minute to 22 minutes (see Figure 6.5). However, the delay average is 2 minutes and 23 seconds, which is acceptable knowing that the passive DNS data is loaded on the fly.

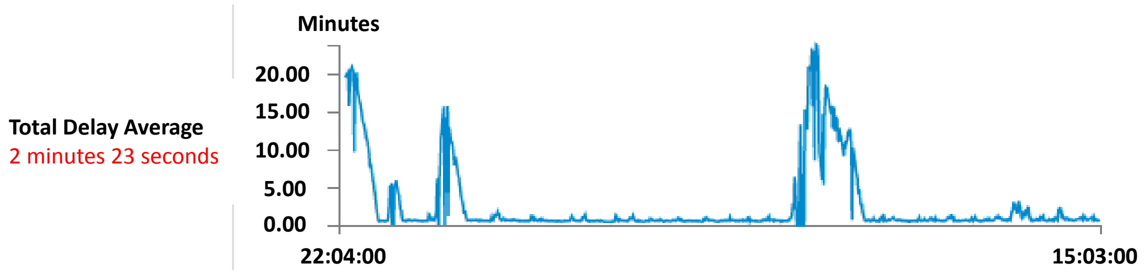


Figure 6.6: Server 1 Processing Delay Time

Monitoring IPs and Domains Features

The second server is used to observe different DNS features. To do so, we deploy Spark streaming capability, which reads data in slide batches of 2 minutes and aggregates it in a window batch of 1 hour. Hereafter, we report the different performance benchmarks (CPU usage, memory consumption and processing delay time).

CPU Usage We observe from Figure 6.7 that the CPU is intensively used every 2 minutes (the period of a slide batch). These batches are created to save different statistical features, namely, cities changes, countries changes, IP changes and TTL values. The map-reduce operation is done every hour for TTL values, cities, countries and IP number changes. Then, scoring functions are applied at the time of insertion into database collections.

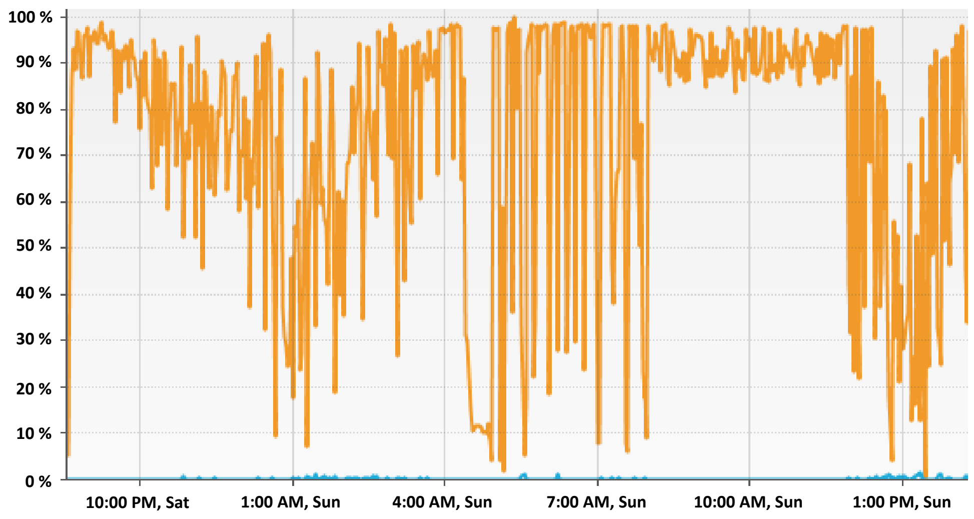


Figure 6.7: Server 2 CPU Usage

Memory Due to the tremendous computation of statistics, the application employs a lot of memory (see Figure 6.8). We notice some spikes, where the system uses sometimes more than 100Gb of memory. The massive use of memory is done usually at the times when the window batches is considered to be processed. The garbage collection is done more frequently than previous case since we need to maintain free memory as much as possible.

Delay Time Unlike the previous application (Server 1 application), this application has a consequent delay time (see Figure 6.9). This is due to the huge collection of statistics. However, the average delay time for the sampled data is 40 minutes 48 seconds, which is less than the batch window period (1 hour). Consequently, we claim that despite the huge number of collected statistics and performance overhead, Spark maintains a respectable near-real time processing.

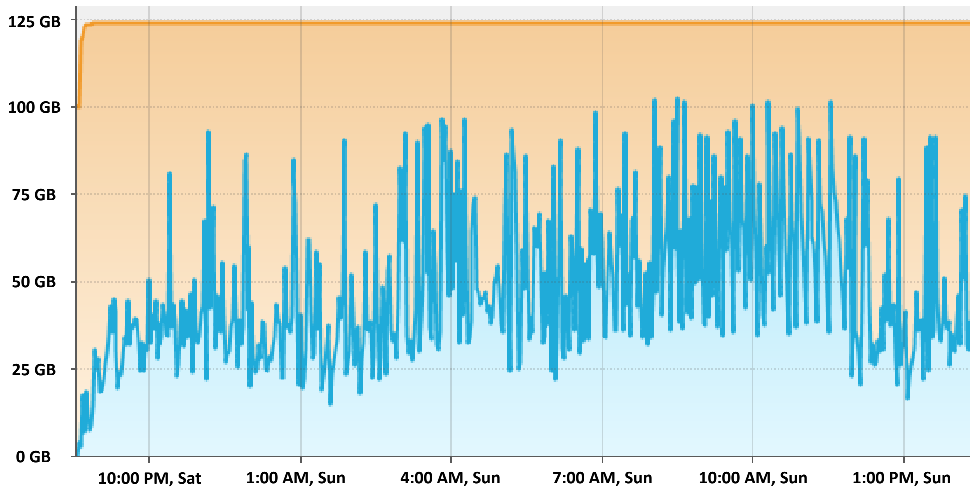


Figure 6.8: Server 2 Memory Consumption

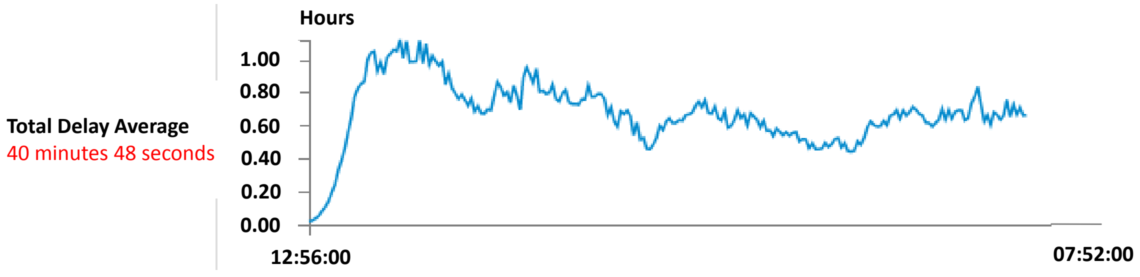


Figure 6.9: Server 2 Processing Delay Time

Monitoring Top Level Domains and IPs of Interest

In addition to the identification of PPM domains, DNS record misuses and features statistics, we have a keen interest to monitor top level domains and IP spaces of interest. In this case, we use filter Spark to detect IPs geo-located in specific countries and their top level domains. We consider all the capabilities integrated in previous servers (Server 1 and Server 2). As such, we monitor PPM domains, record misuses, DNS features statistics. The only difference is the fact that we use a batch window of 2 hours instead of 1 hour. The intent is to collect as much data as we can for feature statistics computations since specific top level domains and IP spaces are a small subset of records collected in passive DNS.

CPU Usage The CPU usage for Server 3 is depicted in Figure 6.10. We notice that the CPU is not overwhelmed like in Server 2 application. However, we observe spikes every two hours, which represents the aggregation period.

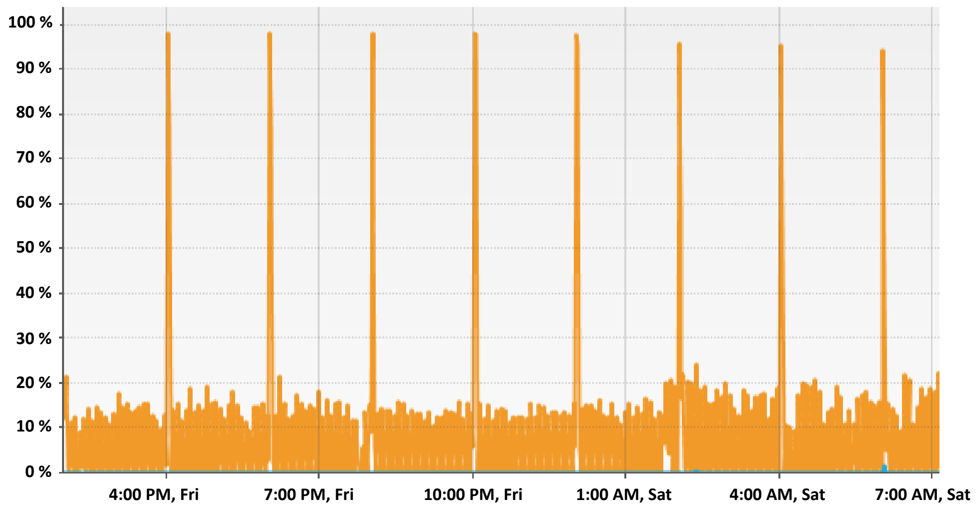


Figure 6.10: Server 3 CPU Usage

Memory The memory consumption has a linear trend. We note that the garbage collection is dissimilar than the previous cases. It takes longer periods to be performed (see Figure 6.11).

Delay Time The delay time has the same trend of CPU usage (see Figures 6.10 and 6.12). The processing time before the aggregation period (2 hours) is less than 1 minute.

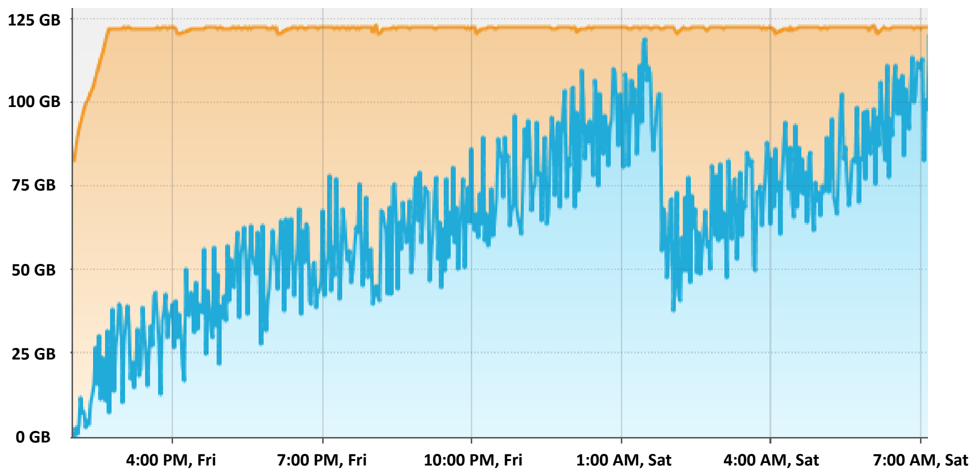


Figure 6.11: Server 3 Memory Consumption

However, at the end of the window batch, we discern some overhead in terms of processing time. This is due to Spark map-reduce operations done every two hours to compute features' time series. The average delay time is 23 seconds and 658 milliseconds, which is the most effective in comparison with previous applications.

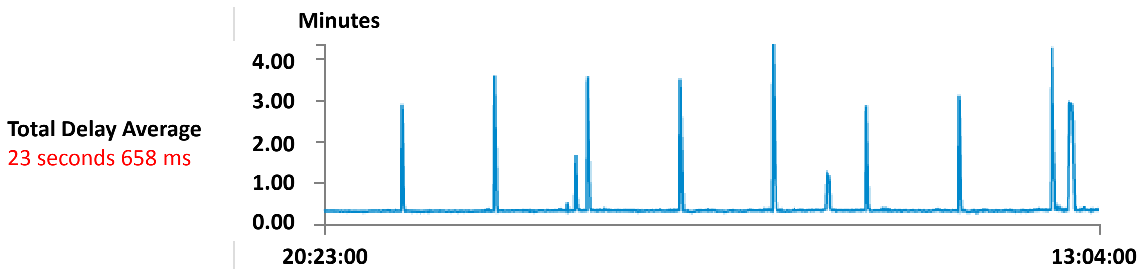


Figure 6.12: Server 3 Processing Delay Time

6.4 Conclusion

In this chapter, we presented a prototype designed and implemented for the purpose of pinpointing anomalies in passive DNS streams. We use Spark framework to integrate a near real-time distributed detection system. We use different operations (e.g., filter, map, reduce) to detect uncommon patterns observed in some DNS records, namely, “TXT”, “CNAME”, “NULL” and “SRV”. We use the outlier detection algorithm to detect abrupt changes in DNS “A” and “AAAA” records. This insight is good to detect fluxing domains. We collect real-time timeseries of TTL values and IP changes to detect fast fluxing of IPs. We corroborate the system with country and city geo-location of IPv4 addresses as well as scoring functions to rank potential fluxing activities. We provide different benchmarks for memory and CPU usages as well as the delay time processing of the deployed system. Initially, we incorporated some use cases where we illustrated some anomalies detected by our system. However, due to third party restrictions, they have not been introduced in this thesis.

Chapter 7

Conclusion

The rise of cyber-threats reported by companies and anti-virus vendors has pushed security researchers to propose new methodologies to extract intelligence about these threats to counter them. In this context, we have attempted in this thesis to define a new guideline to observe and understand the behavior of such threats. We have elaborated four threads of research, where we provide interesting insights about cyber-threat intelligence. We have shown how static and dynamic analyses of malware along with passive DNS monitoring help the security community to identify threats as well as their cyber infrastructures.

We began our research with reverse engineering exercises of two prominent crimeware tool-kits, namely, Mariposa and Zeus. We have unveiled their underlying networking infrastructure. Moreover, we provided detailed description of their components and techniques used to perpetrate malicious activities. The results of these reverse-engineering exercises are entailed in Chapter 5. This step is important since it allowed NCFTA-Canada team to gain an expertise in malware analysis and define new perspectives related to malware research. Despite the importance of reverse-engineering prominent malware threats. This process turns to be tedious due to the huge number of observed malware collected in the wild. To keep track with the huge number of collected malware, we use a dynamic malware analysis framework to collect malware behavior reports and network traces. We used the latter as a ground-truth to create models that detects malicious traffic, whereas malware behavior reports were used to create a situational awareness study about cyber-threat infrastructures.

In Chapter 6, we proposed a graph-theoretic approach to study cyber-threat infrastructures. We used a one-year malware analysis dataset to generate important insights about cyber-threat infrastructures. Moreover, we characterized cyber-threat infrastructures through networking graphs. In this setting, we used Google PageRank algorithm to rank badness of IPs, domains, owners, etc. Thus, we identified key players of cyber-treat infrastructures. We also utilized Min-hashing technique to monitor the sharing between cyber-threat infrastructures on a daily basis. As such, we inferred patterns of cyber-criminal activities. We illustrated a situational awareness of the cyber-threat infrastructures. We introduced results related to malware, IPs, domains and organizations, where we rank their badness. Through patterns inference, we found out that domains are persistent and periodic, whereas patterns of IP addresses tend to be more ephemeral.

As such, in Chapter 4, we initiated a research effort to fingerprint maliciousness in

IP traffic. We put forward a comparative study between two traffic maliciousness fingerprinting techniques, Deep Packet Inspection (DPI) and IP packet headers classification. We evaluated each approach based on its detection and attribution accuracy as well as its level of complexity. Both approaches showed promising results in terms of detection; they are good candidates to strengthen network detection systems since they are based on ground truth collected from dynamic malware analysis. We used data mining algorithms to fingerprint maliciousness based on packet header flow features and Deep Packet Inspection signal processing analysis. Regarding the DPI approach, we used MARFCAT NLP and wavelets classification to fingerprint maliciousness. This approach has shown some troubles to classify the generic malware families. However, it exhibited large scalability and accuracy for less noisy malicious traffic. Regarding flow packet headers approach, we utilized J48, Boosted J48, Naïve Bayesian, Boosted Naïve Bayesian, and SVM to classify malicious and non-malicious traffic. The J48 and Boosted J48 algorithms performed better than other algorithms. We concluded that these two approaches are not in competition but they can create a synergy to identify maliciousness in IP traffic. The DPI approach can classify targets on the fly since it does not need parsing, whereas flow packet headers can increase subsequently the confidence in maliciousness fingerprinting since it shows the ability to segregate malicious and benign traffic.

Finally, in Chapter 7, we entailed the design and implementation of a big data monitoring system, which put under the zoom massive passive DNS data to identify potential attacks. We implemented a prototype on top of Spark cluster computational framework.

We used its stream monitoring capability to identify suspicious domains, DNS record misuses, fluxing IPs and domains. We built a capability to generate real-time timeseries of TTL values and IP addresses changes to detect fast fluxing networks. We integrated a country and city geo-location of IPv4 addresses as well as scoring functions to rank potential fluxing activities. We have shown the scalability in terms of memory, CPU usages as well as the delay time processing.

In order to validate the different research efforts illustrated in this dissertation, we consider the validation classes described in [224], namely, construct validity, internal validity and external validity. Regarding internal validity, we have to know what is the degree of causality between malware and network cyber-threat intelligence. The answer lies in the fact that malware samples have the ability to communicate with third parties through existing network media and protocols, whereas, the analysis of the network traffic may pinpoint some anomalies and misuses that can be malware indicators of compromises. As such, we can conclude that both types of cyber-threat intelligence evolves together, and are causal to each other. Regarding external validity, the different problems tackled within this thesis are of high importance for the security research community, where we try to cover the analysis of prominent threats and their underlying infrastructures as well as indicators of maliciousness potentially observed in the network traffic. These research efforts are published in international peer reviewed journals and conference papers. Regarding construct validity, in each work, we consider adequate measures to illustrate the different theoretic concepts; for instance, in Chapter 4, we use PageRank algorithm to rank badness of entities based on influence concept, the observed results are highly linked to different malware families. In addition, we use min-hashing algorithm to abstract complex graphs and infer patterns, some of them were highly occurring during the analysis period. In Chapter 5, we use machine learning techniques to fingerprint maliciousness. We investigated two approaches with many algorithms to cover as much potential candidates that are good to detect maliciousness. Moreover, we use standard classification metrics like accuracy, false positives, 10-times cross validation, etc. In Chapter 6, we measured the memory and CPU consumption of our big data passive DNS anomaly detection system to gain insight about its performance.

Despite the insights generated from the research efforts discussed in this thesis, some issues need to be addressed. Concerning the investigation of cyber-threat infrastructures, we plan to integrate a near real-time cyber-threat situational awareness dashboard. In addition, based on the observations found in the evolution of badness scores for domains and connected IPs, we aim to look for the empirical periods to consider for domains badness persistence and connected IPs badness sporadicalness. Regarding malicious traffic fingerprinting, our future works fall into improving classification of the malicious traffic according to malware types and families, and deploying the model on a network in order to test its performance on real-time traffic. In addition, malicious traffic covers a wide range of types: DDoS, C&C channels, and intrusion payloads. It is in our plan to further refine the classification of malicious traffic into these types. At present, we only focus on the captured pcaps from known malware to determine maliciousness.

DDoS can also be aided through other existing means (e.g., built into `iptables`). In addition, we have not studied possible evasion from malware trying to avoid detection at the network level. While we believe the headers-only are robust to detect some share of evasive malware, the extent to which our algorithms are robust, stills a challenging research question. It is also worth investigating why SVM and its parameters performed worse in fingerprinting maliciousness in flows. A passive DNS platform has been integrated, however, we target to identify zero-day attacks on the fly by automating the correlation with other sources like malware database, Virus-Total. In addition, we need to incorporate an extendible white-list to eliminate false positives that pollute our database. Thus, easing the analysis and identification of zero-day attacks, including phishing, spamming campaigns, identification of C&Cs and correlation with malware database. Moreover, we look thoroughly into rare records like “TXT”, “ANY”, “SRV” and “NULL”. The existence of “ANY” records can be an indicator for Reflective Distributed Denial of Service (DRDoS) attacks, whereas “NULL” and “TXT” records are good candidates to detect malicious payload communications. A study on malicious payloads is a must to identify encrypted and encoded messages.

Appendix A

Signal and NLP DPI Results

Hereafter, we list results of the DPI detection approach. They are based on the whole packet examination (i.e., headers and payload) that illustrate the precision per algorithm combinations as well as attribution for the top precise malware types. The methodology behind them is described in Section 5.4 and the results are discussed in Section 5.5.2. The algorithms' options, in addition to those described in [134], are:

- `-dynaclass` – treat learned classes as labels automatically from the reports (no predefined classes are set at the beginning),
- `-binary` – treat data as pure binary non-formatted data,
- `-nopreprep` – to skip extra pre-pre-processing,
- `-sdwt` – use separating discrete wavelet transform, and

- `-flucid` – generate FORENSIC LUCID expressions for subsequent forensic investigations and reasoning in an external system [137].

guess	run	algorithms	good	bad	%
1st	1	`-dynaclass -binary -nopreprep -raw -fft -cos -flucid`	67	154	30.32
1st	2	`-dynaclass -binary -nopreprep -raw -fft -diff -flucid`	55	166	24.89
1st	3	`-dynaclass -binary -nopreprep -raw -fft -cheb -flucid`	55	166	24.89
1st	4	`-dynaclass -binary -nopreprep -raw -fft -eucl -flucid`	50	171	22.62
1st	5	`-dynaclass -binary -nopreprep -raw -fft -hamming -flucid`	37	184	16.74
1st	6	`-dynaclass -binary -nopreprep -raw -fft -mink -flucid`	34	187	15.38
2nd	1	`-dynaclass -binary -nopreprep -raw -fft -cos -flucid`	92	129	41.63
2nd	2	`-dynaclass -binary -nopreprep -raw -fft -diff -flucid`	77	144	34.84
2nd	3	`-dynaclass -binary -nopreprep -raw -fft -cheb -flucid`	77	144	34.84
2nd	4	`-dynaclass -binary -nopreprep -raw -fft -eucl -flucid`	73	148	33.03
2nd	5	`-dynaclass -binary -nopreprep -raw -fft -hamming -flucid`	46	175	20.81
2nd	6	`-dynaclass -binary -nopreprep -raw -fft -mink -flucid`	47	174	21.27
guess	run	class	good	bad	%
1st	1	VirTool.Win32.VBInject.gen.bp (v)	6	0	100.00
1st	2	Trojan.Win32.Agent.roei	6	0	100.00
1st	3	BehavesLike.Win32.Malware.dls (mx-v)	6	0	100.00
1st	4	Worm.Win32.AutoRun.dkch	6	0	100.00
1st	5	Trojan-FakeAV.Win32.Agent.det	6	0	100.00
1st	6	FraudTool.Win32.FakeRean	6	0	100.00
1st	7	VirTool:Win32/Obfuscator.WJ (suspicious)	6	0	100.00
1st	8	Trojan.Win32.Vilsel.ayyw	6	0	100.00
1st	9	Worm:Win32/Yeltminky.A!dll	6	0	100.00
1st	10	Trojan.Win32.Meredrop	6	0	100.00
1st	11	TrojanDownloader:Win32/Allsum	12	0	100.00
1st	12	Virtumonde	6	0	100.00
1st	13	Backdoor.Win32.Hupigon.nndu	6	0	100.00
1st	14	VirTool:WinNT/Protmin.gen!C [generic]	6	0	100.00

1st	21	Trojan-Spy.Win32.SpyEyes.aecv	6	0	100.00
1st	22	Trojan:Win32/Swrort.A	11	1	91.67
1st	23	TrojanDownloader:Win32/Carberp.C	11	1	91.67
1st	24	PWS:Win32/Lolyda.BF	15	3	83.33
1st	25	Trojan.Win32.Yakes.qjn	8	4	66.67
1st	26	Trojan.Win32.Agent.rlnz	5	7	41.67
1st	27	Trojan.Win32.VBKrypt.fkvx	6	12	33.33
1st	28	VirTool:Win32/VBInject.OT	6	12	33.33
1st	29	HomeMalwareCleaner.FakeVimes	36	264	12.00
1st	30	Trojan.Win32.Generic!BT	56	598	8.56
1st	31	Trojan.FakeAlert	6	108	5.26
1st	32	Trojan.Win32.Generic.pak!cobra	0	18	0.00
2nd	1	VirTool.Win32.VBInject.gen.bp (v)	6	0	100.00
2nd	2	Trojan.Win32.Agent.roei	6	0	100.00
2nd	3	BehavesLike.Win32.Malware.dls (mx-v)	6	0	100.00
2nd	4	Worm.Win32.AutoRun.dkch	6	0	100.00
2nd	5	Trojan-FakeAV.Win32.Agent.det	6	0	100.00
2nd	6	FraudTool.Win32.FakeRean	6	0	100.00
2nd	7	VirTool:Win32/Obfuscator.WJ (suspicious)	6	0	100.00
2nd	8	Trojan.Win32.Vilsel.ayyw	6	0	100.00
2nd	9	Worm:Win32/Yeltminky.A!dll	6	0	100.00
2nd	10	Trojan.Win32.Meredrop	6	0	100.00
2nd	11	TrojanDownloader:Win32/Allsum	12	0	100.00
2nd	12	Virtumonde	6	0	100.00
2nd	13	Backdoor.Win32.Hupigon.nndu	6	0	100.00
2nd	14	VirTool:WinNT/Protmin.gen!C [generic]	6	0	100.00
2nd	21	Trojan-Spy.Win32.SpyEyes.aecv	6	0	100.00
2nd	22	Trojan:Win32/Swrort.A	11	1	91.67
2nd	23	TrojanDownloader:Win32/Carberp.C	11	1	91.67
2nd	24	PWS:Win32/Lolyda.BF	16	2	88.89

2nd	25	Trojan.Win32.Yakes.qjn	9	3	75.00
2nd	26	Trojan.Win32.Agent.rlnz	5	7	41.67
2nd	27	Trojan.Win32.VBKrypt.fkvx	18	0	100.00
2nd	28	VirTool:Win32/VBInject.OT	6	12	33.33
2nd	29	HomeMalwareCleaner.FakeVimes	66	234	22.00
2nd	30	Trojan.Win32.Generic!BT	117	537	17.89
2nd	31	Trojan.FakeAlert	15	99	13.16
2nd	32	Trojan.Win32.Generic.pak!cobra	0	18	0.00

Table A.1: No-Filtering Results by Algorithm Combination and Malware

guess	run	algorithms	good	bad	%
1st	1	`-dynaclass -binary -nopreprep -sdwt -fft -cos -flucid`	55	146	27.36
1st	2	`-dynaclass -binary -nopreprep -sdwt -fft -diff -flucid`	41	180	18.55
1st	3	`-dynaclass -binary -nopreprep -sdwt -fft -mink -flucid`	41	180	18.55
1st	4	`-dynaclass -binary -nopreprep -sdwt -fft -cheb -flucid`	41	180	18.55
1st	5	`-dynaclass -binary -nopreprep -sdwt -fft -eucl -flucid`	41	180	18.55
1st	6	`-dynaclass -binary -nopreprep -sdwt -fft -hamming -flucid`	30	191	13.57
2nd	1	`-dynaclass -binary -nopreprep -sdwt -fft -cos -flucid`	75	126	37.31
2nd	2	`-dynaclass -binary -nopreprep -sdwt -fft -diff -flucid`	56	165	25.34
2nd	3	`-dynaclass -binary -nopreprep -sdwt -fft -mink -flucid`	67	154	30.32
2nd	4	`-dynaclass -binary -nopreprep -sdwt -fft -cheb -flucid`	55	166	24.89
2nd	5	`-dynaclass -binary -nopreprep -sdwt -fft -eucl -flucid`	58	163	26.24
2nd	6	`-dynaclass -binary -nopreprep -sdwt -fft -hamming -flucid`	44	177	19.91
guess	run	class	good	bad	%
1st	1	VirTool.Win32.VBInject.gen.bp (v)	6	0	100.00
1st	2	Trojan.Win32.Agent.roei	6	0	100.00
1st	3	BehavesLike.Win32.Malware.dls (mx-v)	6	0	100.00
1st	4	Worm.Win32.AutoRun.dkch	6	0	100.00
1st	5	Trojan-FakeAV.Win32.Agent.det	6	0	100.00
1st	6	FraudTool.Win32.FakeRean	6	0	100.00

1st	7	VirTool:Win32/Obfuscator.WJ (suspicious)	6	0	100.00
1st	8	Trojan.Win32.Vilsel.ayyw	6	0	100.00
1st	9	Worm:Win32/Yeltminky.A!dll	6	0	100.00
1st	10	Trojan.Win32.Meredrop	6	0	100.00
1st	11	Virtumonde	6	0	100.00
1st	12	Backdoor.Win32.Hupigon.nndu	6	0	100.00
1st	13	VirTool:WinNT/Protmin.gen!C [generic]	6	0	100.00
1st	14	PWS:Win32/Fareit.gen!C [generic]	6	0	100.00
1st	15	Trojan-Dropper.Win32.Injector.cxqb	6	0	100.00
1st	16	Trojan.Win32.Menti.mlgp	6	0	100.00
1st	17	Trojan.Win32.Buzus (v)	6	0	100.00
1st	18	Trojan.Win32.Agent.rlot	6	0	100.00
1st	19	Trojan-Spy.Win32.SpyEyes.aecv	6	0	100.00
1st	20	Trojan.Win32.FakeAV.lcpt	11	1	91.67
1st	21	TrojanDownloader:Win32/Allsum	10	2	83.33
1st	22	Trojan.Win32.Yakes.qjn	10	2	83.33
1st	23	Trojan.Win32.Agent.rlnz	9	3	75.00
1st	24	Trojan:Win32/Swrort.A	6	6	50.00
1st	25	TrojanDownloader:Win32/Carberp.C	6	6	50.00
1st	26	Trojan.Win32.VBKrypt.fkvx	5	11	31.25
1st	27	VirTool:Win32/VBInject.OT	5	11	31.25
1st	28	HomeMalwareCleaner.FakeVimes	46	250	15.54
1st	29	Trojan.FakeAlert	8	104	7.14
1st	30	Trojan.Win32.Generic.pak!cobra	1	17	5.56
1st	31	Trojan.Win32.Generic!BT	18	626	2.80
1st	32	PWS:Win32/Lolyda.BF	0	18	0.00
2nd	1	VirTool.Win32.VBInject.gen.bp (v)	6	0	100.00
2nd	2	Trojan.Win32.Agent.roei	6	0	100.00
2nd	3	BehavesLike.Win32.Malware.dls (mx-v)	6	0	100.00
2nd	4	Worm.Win32.AutoRun.dkch	6	0	100.00

2nd	5	Trojan-FakeAV.Win32.Agent.det	6	0	100.00
2nd	6	FraudTool.Win32.FakeRean	6	0	100.00
2nd	7	VirTool:Win32/Obfuscator.WJ (suspicious)	6	0	100.00
2nd	8	Trojan.Win32.Vilsel.ayyw	6	0	100.00
2nd	9	Worm:Win32/Yeltminky.A!dll	6	0	100.00
2nd	10	Trojan.Win32.Meredrop	6	0	100.00
2nd	11	Virtumonde	6	0	100.00
2nd	12	Backdoor.Win32.Hupigon.nndu	6	0	100.00
2nd	13	VirTool:WinNT/Protmin.gen!C [generic]	6	0	100.00
2nd	14	PWS:Win32/Fareit.gen!C [generic]	6	0	100.00
2nd	15	Trojan-Dropper.Win32.Injector.cxqb	6	0	100.00
2nd	16	Trojan.Win32.Menti.mlgp	6	0	100.00
2nd	17	Trojan.Win32.Buzus (v)	6	0	100.00
2nd	18	Trojan.Win32.Agent.rlot	6	0	100.00
2nd	19	Trojan-Spy.Win32.SpyEyes.aecv	6	0	100.00
2nd	20	Trojan.Win32.FakeAV.lcpt	12	0	100.00
2nd	21	TrojanDownloader:Win32/Allsum	11	1	91.67
2nd	22	Trojan.Win32.Yakes.qjn	11	1	91.67
2nd	23	Trojan.Win32.Agent.rlnz	10	2	83.33
2nd	24	Trojan:Win32/Swrort.A	6	6	50.00
2nd	25	TrojanDownloader:Win32/Carberp.C	10	2	83.33
2nd	26	Trojan.Win32.VBKrypt.fkvx	15	1	93.75
2nd	27	VirTool:Win32/VBInject.OT	5	11	31.25
2nd	28	HomeMalwareCleaner.FakeVimes	76	220	25.68
2nd	29	Trojan.FakeAlert	19	93	16.96
2nd	30	Trojan.Win32.Generic.pak!cobra	2	16	11.11
2nd	31	Trojan.Win32.Generic!BT	62	582	9.63
2nd	32	PWS:Win32/Lolyda.BF	2	16	11.11

Table A.2: Wavelet-Filtered Results by Algorithm Combination and Malware

guess	run	algorithms	good	bad	%
1st	1	`-dynaclass -binary -nopreprep -low -fft -cos -flucid`	60	161	27.15
1st	2	`-dynaclass -binary -nopreprep -low -fft -cheb -flucid`	54	167	24.43
1st	3	`-dynaclass -binary -nopreprep -low -fft -diff -flucid`	54	167	24.43
1st	4	`-dynaclass -binary -nopreprep -low -fft -eucl -flucid`	46	175	20.81
1st	5	`-dynaclass -binary -nopreprep -low -fft -hamming -flucid`	35	186	15.84
1st	6	`-dynaclass -binary -nopreprep -low -fft -mink -flucid`	33	188	14.93
2nd	1	`-dynaclass -binary -nopreprep -low -fft -cos -flucid`	88	133	39.82
2nd	2	`-dynaclass -binary -nopreprep -low -fft -cheb -flucid`	74	147	33.48
2nd	3	`-dynaclass -binary -nopreprep -low -fft -diff -flucid`	74	147	33.48
2nd	4	`-dynaclass -binary -nopreprep -low -fft -eucl -flucid`	69	152	31.22
2nd	5	`-dynaclass -binary -nopreprep -low -fft -hamming -flucid`	49	172	22.17
2nd	6	`-dynaclass -binary -nopreprep -low -fft -mink -flucid`	48	173	21.72
guess	run	class	good	bad	%
1st	1	Trojan:Win32/Swrort.A	12	0	100.00
1st	2	VirTool.Win32.VBInject.gen.bp (v)	6	0	100.00
1st	3	Trojan.Win32.Agent.roei	6	0	100.00
1st	4	BehavesLike.Win32.Malware.dls (mx-v)	6	0	100.00
1st	5	Worm.Win32.AutoRun.dkch	6	0	100.00
1st	6	Trojan-FakeAV.Win32.Agent.det	6	0	100.00
1st	7	FraudTool.Win32.FakeRean	6	0	100.00
1st	8	VirTool:Win32/Obfuscator.WJ (suspicious)	6	0	100.00
1st	9	Trojan.Win32.Vilsel.ayyw	6	0	100.00
1st	10	Worm:Win32/Yeltminky.A!dll	6	0	100.00
1st	11	Trojan.Win32.Meredrop	6	0	100.00
1st	12	Virtumonde	6	0	100.00
1st	13	Backdoor.Win32.Hupigon.nndu	6	0	100.00
1st	14	VirTool:WinNT/Protmin.gen!C [generic]	6	0	100.00
1st	21	Trojan-Spy.Win32.SpyEyes.aecv	6	0	100.00
1st	22	TrojanDownloader:Win32/Allsum	11	1	91.67

1st	23	TrojanDownloader:Win32/Carberp.C	10	2	83.33
1st	24	PWS:Win32/Lolyda.BF	15	3	83.33
1st	25	Trojan.Win32.Yakes.qjn	8	4	66.67
1st	26	Trojan.Win32.Agent.rlnz	6	6	50.00
1st	27	Trojan.Win32.VBKrypt.fkvx	6	12	33.33
1st	28	VirTool:Win32/VBInject.OT	6	12	33.33
1st	29	HomeMalwareCleaner.FakeVimes	37	263	12.33
1st	30	Trojan.Win32.Generic.pak!cobra	2	16	11.11
1st	31	Trojan.FakeAlert	8	106	7.02
1st	32	Trojan.Win32.Generic!BT	35	619	5.35
2nd	1	Trojan:Win32/Swrort.A	12	0	100.00
2nd	2	VirTool.Win32.VBInject.gen.bp (v)	6	0	100.00
2nd	3	Trojan.Win32.Agent.roei	6	0	100.00
2nd	4	BehavesLike.Win32.Malware.dls (mx-v)	6	0	100.00
2nd	5	Worm.Win32.AutoRun.dkch	6	0	100.00
2nd	6	Trojan-FakeAV.Win32.Agent.det	6	0	100.00
2nd	7	FraudTool.Win32.FakeRean	6	0	100.00
2nd	8	VirTool:Win32/Obfuscator.WJ (suspicious)	6	0	100.00
2nd	9	Trojan.Win32.Vilsel.ayyw	6	0	100.00
2nd	10	Worm:Win32/Yeltminky.A!dll	6	0	100.00
2nd	11	Trojan.Win32.Meredrop	6	0	100.00
2nd	12	Virtumonde	6	0	100.00
2nd	13	Backdoor.Win32.Hupigon.nndu	6	0	100.00
2nd	14	VirTool:WinNT/Protmin.gen!C [generic]	6	0	100.00
2nd	21	Trojan-Spy.Win32.SpyEyes.aecv	6	0	100.00
2nd	22	TrojanDownloader:Win32/Allsum	11	1	91.67
2nd	23	TrojanDownloader:Win32/Carberp.C	10	2	83.33
2nd	24	PWS:Win32/Lolyda.BF	15	3	83.33
2nd	25	Trojan.Win32.Yakes.qjn	9	3	75.00
2nd	26	Trojan.Win32.Agent.rlnz	8	4	66.67

2nd	27	Trojan.Win32.VBKrypt.fkvx	18	0	100.00
2nd	28	VirTool:Win32/VBInject.OT	6	12	33.33
2nd	29	HomeMalwareCleaner.FakeVimes	66	234	22.00
2nd	30	Trojan.Win32.Generic.pak!cobra	2	16	11.11
2nd	31	Trojan.FakeAlert	14	100	12.28
2nd	32	Trojan.Win32.Generic!BT	105	549	16.06

Table A.3: Low-Pass-Filtered Results by Algorithm Combination and Malware

Bibliography

[1] Alexa, actionable analytics for the web. `https://www.alexa.com`, visited on December 18, 2015.

[2] jNetPcap OpenSource. `http://www.jnetpcap.com/`, visited on December 18, 2015.

[3] Malware Classification. `http://tinyurl.com/p539nnd`, visited on December 18, 2015.

[4] MaxMind: IP Geolocation and Online Fraud Prevention. `https://www.maxmind.com`, visited on December 18, 2015.

[5] PaiMei - a reverse engineering framework. `http://code.google.com/p/paimei/`, visited on December 18, 2015.

[6] Python plugin for interactive disassembler pro. `https://code.google.com/p/idapython/`, visited on December 18, 2015.

[7] Weka 3 Data Mining with Open Source Machine Learning Software in Java. `http://www.cs.waikato.ac.nz/ml/weka/`, visited on December 18, 2015.

[8] WISNET: Downloads. `http://wisnet.seecs.nust.edu.pk/downloads.php`, visited on December 18, 2015.

[9] wisegeek clear answers for common questions, 2012. `http://www.wisegeek.com/what-is-cyber-intelligence.htm`, visited on December 18, 2015.

[10] IETF RFC 1794. DNS support for load balancing. `http://tools.ietf.org/html/rfc1794`, visited on December 18, 2015.

[11] IETF RFC 1912. Common DNS operational and configuration errors. `http://tools.ietf.org/html/rfc1912`, visited on December 18, 2015.

[12] A. Gulbrandsen, P. Vixie, L. Esibov. A DNS rr for specifying the location of services (DNS srv). `https://www.ietf.org/rfc/rfc2782.txt`, visited on December 18, 2015.

[13] A. F. Abdelnour and I. W. Selesnick. Nearly symmetric orthogonal wavelet bases. In *proceedings of IEEE International Conference in Acoustics, Speech, Signal Processing (ICASSP)*, volume 6, 2001.

[14] Emile Aben. Conficker/conflicker/downadup as seen from the ucsd network telescope, 2008. `http://www.caida.org/research/security/ms08-067/conficker.xml`, visited on December 18,2015.

[15] Charu C. Aggarwal, Stephen C. Gates, and Philip S. Yu. On the merits of building categorization systems by supervised clustering. In *KDD*, KDD'99, pages 352–356, New York, NY, USA, 1999.

[16] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. Oba2: an onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, 2014.

[17] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[18] Riyad Akla Alshammari. *Automatically Generating Robust Signatures Using a Machine Learning Approach To Unveil Encrypted VOIP Traffic Without Using Port Numbers, IP Addresses and Payload Inspection.* PhD thesis, Dalhousie University, Halifax, Nova Scotia, Canada, May 2012.

[19] Riyad Akla Alshammari and A. N. Zincir-Heywood. Investigating two different approaches for encrypted traffic classification. In *proceedings of the Sixth Annual Conference on Privacy, Security and Trust (PST'08)*, pages 156–166, October 2008.

[20] Riyad Akla Alshammari and A. N. Zincir-Heywood. Machine learning based encrypted traffic classification: Identifying SSH and Skype. In *proceedings of the IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA 2009)*, pages 1–8, July 2009.

[21] Dennis Andriesse, Christian Rossow, Brett Stone-Gross, Daniel Plohmann, and Herbert Bos. Highly resilient peer-to-peer botnets are here: An analysis of gameover zeus. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*, pages 116–123. IEEE, 2013.

[22] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for DNS. In *proceedings of USENIX security symposium*, pages 273–290, 2010.

[23] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou II, and David Dagon. Detecting malware domains at the upper DNS hierarchy. In *USENIX Security Symposium*, page 16, 2011.

[24] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou II, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From throw-away traffic to bots: Detecting the rise of dga-based malware. In *USENIX security symposium*, pages 491–506, 2012.

[25] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of Internet malware. Technical report, University of Michigan, April 2007. `http://www.eecs.umich.edu/techreports/cse/2007/CSE-TR-530-07.pdf`, visited on December 18, 2015.

[26] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 9, 2009.

[27] Ron Begleiter, Yuval Elovici, Yona Hollander, Ori Mendelson, Lior Rokach, and Roi Saltzman. A fast and scalable method for threat detection in large-scale DNS logs. In *proceedings of IEEE International Conference on Big Data*, pages 738–741. IEEE, 2013.

[28] Biddle, P. and England, P. and Peinado, M. and Willman, B. The Darknet and the Future of Content Protection. In *proceedings of the 2003 Digital Rights Management*, pages 344–365. Springer, 2003.

[29] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. Exposure: Finding malicious domains using passive DNS analysis. In *proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[30] Leyla Bilge, Sevil Sen, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. Exposure: a passive DNS analysis service to detect and report malicious domains. *ACM Transactions on Information and System Security (TISSEC)*, 16(4):14, 2014.

[31] James R. Binkley and Suresh Singh. An algorithm for anomaly-based botnet detection. In *proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet - Volume 2*, SRUTI'06, pages 1–7, Berkeley, CA, USA, 2006. USENIX Association.

[32] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *proceedings of Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.

[33] Eric Bloedorn, Alan D. Christiansen, William Hill, Clement Skorupka, Lisa M. Talbot, and Jonathan Tivel. Data mining for network intrusion detection: How to get started. Technical report, The MITRE Corporation, 2001.

[34] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, (10), 2008.

[35] Nathaniel Boggs, Sharath Hiremagalore, Angelos Stavrou, and Salvatore J. Stolfo. Cross-domain collaborative anomaly detection: so far yet so close. In *proceedings of Recent Advances in Intrusion Detection*, pages 142–160. Springer, 2011.

[36] Phillip Bonacich. Some unique properties of eigenvector centrality. *Social Networks*, 29(4):555–564, October 2007.

[37] Stephen P. Borgatti. Centrality and network flow. *Social Networks*, 27(1):55–71, January 2005.

[38] Amine Boukhtouta, Nour-Eddine Lakhdari, and Mourad Debbabi. Inferring malware family through application protocol sequences signature. In *proceedings of the 6th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2014.

[39] Amine Boukhtouta, Nour-Eddine Lakhdari, Serguei Andrei Mokhov, and Mourad Debbabi. Towards fingerprinting malicious traffic. In *proceedings of the 4th International Conference on Ambient Systems, Networks and Technologies*, ANT'13. Elsevier, 2013.

[40] Amine Boukhtouta, Serguei A Mokhov, Nour-Eddine Lakhdari, Mourad Debbabi, and Joey Paquet. Network malware classification comparison using DPI and flow packet headers. *Journal of Computer Virology and Hacking Techniques*, pages 1–32, 2015.

[41] Amine Boukhtouta, Djedjiga Mouheb, Mourad Debbabi, Omar Alfandi, Farkhund Iqbal, and May El Barachi. Graph-theoretic characterization of cyber-threat infrastructures. *Digital Investigation*, 14:S3–S15, 2015.

[42] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web*, WWW7, pages 107–117. Elsevier Science Publishers B. V., 1998.

[43] Tao Cai, Jian Yang, and Xinxin Jin. Inferring DNS flooding attack through passive data analysis.

[44] Su Chang and Thomas E. Daniels. P2P botnet detection using behavior clustering & statistical tests. In *proceedings of the second ACM workshop on Security and artificial intelligence*, AISec'09, pages 23–30, New York, NY, USA, 2009. ACM.

[45] William Chavuenet. *A manual of spherical and practical astronomy*. Philadelphia, J. B. Lippincott & co. London, Trübner & co., 1871.

[46] Ken Chiang and Levi Lloyd. A case study of the rustock rootkit and spam bot. In *proceedings of the First Workshop in Understanding Botnets*, volume 20, 2007.

[47] Hyunsang Choi and Heejo Lee. Identifying botnets by capturing group activities in DNS traffic. *Computer Networks*, 56(1):20–33, 2012.

[48] Miha Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 5–14. ACM, 2007.

[49] Lucian Constantin. DDoS attack against spamhaus was reportedly the largest in history, March 2013. `http://tinyurl.com/mx22qvr`, visited on December 18, 2015.

[50] Heather Crawford and John Aycock. Kwyjibo: automatic domain name generation. *Software: Practice and Experience*, 38(14):1561–1567, 2008.

[51] Chavdar Dangalchev. Residual closeness in networks. *Physica A: Statistical Mechanics and its Applications*, 365(2):556–564, 2006.

[52] Neil Daswani and Michael Stoppelman. The anatomy of Clickbot.A. In *proceedings of the First Workshop on Hot Topics in Understanding Botnets*, pages 1–11. USENIX Association, 2007.

[53] DataRescue. IDAPro - multi-processor disassembler and debugger, 2009. `http://www.hex-rays.com/idapro/`, visited on December 18, 2015.

[54] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

[55] Luca Deri, Simone Mainardi, Maurizio Martinelli, and Enrico Gregori. Exploiting DNS traffic to rank internet domains. In *proceedings of IEEE International Conference on Communications, ICC'13*, pages 1325–1329, 2013.

[56] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *proceedings of the Eleventh symposium on High performance interconnects*, pages 44–51. IEEE, 2003.

[57] Inderjit S. Dhillon, Subramanyam Mallela, and Rahul Kumar. A divisive information theoretic feature clustering algorithm for text classification. *The Journal of Machine Learning Research*, 3:1265–1287, 2003.

[58] Christian J Dietrich, Christian Rossow, Felix C Freiling, Herbert Bos, Maarten van Steen, and Norbert Pohlmann. On botnets that use DNS for command and control. In *proceedings of the Seventh European Conference on Computer Network Defense*, pages 9–16. IEEE, 2011.

[59] Christian J. Dietrich, Christian Rossow, and Norbert Pohlmann. CoCoSpot: Clustering and recognizing botnet command and control channels using traffic analysis. *Computer Networks*, 57(2):475–486, 2013.

[60] David Dittrich and Sven Dietrich. P2p as botnet command and control: a deeper insight. In *proceedings of the Third International Conference on Malicious and Unwanted Software*, MALWARE 2008, pages 41–48. IEEE, 2008.

[61] DNSBL. Spam database lookup. `http://www.dnsbl.info`, visited on December 18, 2015.

[62] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, 2012.

[63] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, M. Thomas. Domainkeys identified mail (dkim) signatures draft-ietf-dkim-base-10. `https://tools.`

ietf.org/html/draft-ietf-dkim-base-10, visited on December 18, 2015.

[64] École Polytechnique Fédérale de Lausanne. The SCALA programming language. http://www.scala-lang.org, visited on December 18, 2015.

[65] Sean R Eddy. Hidden markov models. *Current Opinion in Structural Biology*, 6(3):361–365, 1996.

[66] Claude Fachkha, Elias Bou-Harb, and Mourad Debbabi. Fingerprinting internet DNS amplification DDoS activities. In *proceedings of the Sixth International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2014.

[67] Wei Fan, M. Miller, S.J. Stolfo, Wenke Lee, and P.K. Chan. Using artificial anomalies to detect unknown and known network intrusions. In *proceedings of the IEEE International Conference on Data Mining (ICDM 2001)*, pages 123–130, 2001.

[68] R Foundation. The R project for statistical computing. https://www.r-project.org, visited on December 18, 2015.

[69] The Apache Software Foundation. Apache river news. http://river.apache.org/, visited on December 18, 2015.

[70] The Apache Software Foundation. Apache spark is a fast and general engine for large-scale data processing. `http://spark.apache.org/`, visited on December 18, 2015.

[71] The Apache Software Foundation. Welcome to apache hadoop. `https://hadoop.apache.org/`, visited on December 18, 2015.

[72] The Python Software Foundation. Welcome to Python.org. `https://www.python.org`, visited on December 18, 2015.

[73] Eibe Frank. J48, 2012. `http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html`, visited on December 18, 2015.

[74] Eibe Frank, Shane Legg, and Stuart Inglis. Class SMO, 2012. `http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SMO.html`, visited on December 18, 2015.

[75] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry Journal*, 40(1):35–41, March 1977.

[76] Yoav Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, September 1995.

[77] Thomas Gärtner. A survey of kernels for structured data. *SIGKDD Explorations Newsletter*, 5(1):49–58, July 2003.

[78] Vladimir Golovko, Sergei Bezobrazov, Pavel Kachurka, and Leonid Vaitsekhovich. Neural network and artificial immune systems for malware and network intrusion detection. In *Advances in Machine Learning II*, pages 485–513. Springer, 2010.

[79] Nidhi Grover and Ritika Wason. Comparative analysis of pagerank and hits algorithms. In *International Journal of Engineering Research and Technology*, volume 1, pages 1–15. ESRSA Publications, Oct 2012.

[80] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *proceedings of the 17th conference on Security symposium*, SS'08, pages 139–154, Berkeley, CA, USA, 2008. USENIX Association.

[81] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. Bothunter: detecting malware infection through ids-driven dialog correlation. In *proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[82] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[83] Bin Han. Towards a multi-tier runtime system for GIPSY. Master's thesis, 2010.

[84] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[85] M. A. Hearst, S.T. Dumais, E. Osman, J. Platt, and B. Scholkopf. Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28, July 1998.

[86] Christopher E Heil and David F Walnut. Continuous and discrete wavelet transforms. *SIAM review*, 31(4):628–666, 1989.

[87] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. *Computer Security ESORICS 2009*, pages 1–18, 2009.

[88] Thorsten Holz, Christian Gorecki, Konrad Rieck, and Felix C Freiling. Measuring and detecting fast-flux service networks. In *proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[89] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix C Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *proceedings of the First Usenix Workshop on Large-Scale Exploits and Emergent Threats*, volume 8, pages 1–9, 2008.

[90] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. MutantX-S: Scalable malware clustering based on static features. In *proceedings of the USENIX Annual Technical Conference*, pages 187–198, 2013.

[91] Galen Hunt and Doug Brubacher. Detours: binary interception of win32 functions. In *proceedings of the Third Conference on USENIX Windows NT Symposium*, volume 3 of *WINSYM'99*, 1999.

[92] IBM. Addressing TCP/IP hosts, 2009. `http://tinyurl.com/hft8nfq`, visited on December 18, 2015.

[93] Iczelion. Tutorial 24: Windows hooks, 2009. `http://win32assembly.programminghorizon.com/tut24.html`, visited on December 18, 2015.

[94] IDEFENCE. Sysanalyzer overview. `https://github.com/dzzie/SysAnalyzer`, visited December 18, 2015.

[95] Defence Intelligence. Mariposa botnet analysis. Technical report, October 2009.

[96] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: Understanding microblogging usage and communities. In *proceedings of Workshop on Web Mining and Social Network Analysis*, pages 56–65, New York, NY, USA, 2007. ACM.

[97] Yi Ji. Scalability evaluation of the GIPSY runtime system. Master's thesis, March 2011. `http://spectrum.library.concordia.ca/7152/`, visited on December 18, 2015.

[98] Jiri Sedlacek, Tomas Hurka. Visualvm. `https://visualvm.java.net`, visited on December 18, 2015.

[99] Lars Backstrom Cameron Marlow Johan Ugander, Brian Karrer. The anatomy of the facebook social graph. *Computing Research Repository (CoRR)*, abs/1111.4503, 2011.

[100] Kaminsky, Dan. Dan kaminsky's Blog. `http://dankaminsky.com/2004/07/29/51/`, visited on December 18, 2015.

[101] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 3–14, New York, NY, USA, 2008. ACM.

[102] A Mert Kara, Hamad Binsalleeh, Mohammad Mannan, Amr Youssef, and Mourad Debbabi. Detection of malicious payload distribution channels in DNS. In *proceedings of IEEE International Conference on Communications (ICC)*, pages 853–858. IEEE, 2014.

[103] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. Wide-scale botnet detection and characterization. In *proceedings of Hot Bots*, HotBots'07, pages 1–7, Berkeley, CA, USA, 2007. USENIX Association.

[104] G. Katz, A. Shabtai, L. Rokach, and N. Ofek. ConfDTree: Improving decision trees using confidence intervals. In *proceedings of the 12th IEEE International Conference on Data Mining (ICDM)*, pages 339–348, December 2012.

[105] Nizar Kheir, Gregory Blanc, Hervé Debar, Joaquin Garcia-Alfaro, and Dingqi Yang. Automated classification of C&C connections through malware URL clustering. In *proceedings of ICT Systems Security and Privacy Protection*, pages 252–266. Springer, 2015.

[106] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, September 1999.

[107] Manesh Kokare, P. K. Biswas, and B. N. Chatterji. Texture image retrieval using new rotated complex wavelet filters. *IEEE Transaction on Systems, Man, and Cybernetics-Part B: Cybernetics*, 6(35):1168–1178, 2005.

[108] Manesh Kokare, P. K. Biswas, and B. N. Chatterji. Rotation-invariant texture image retrieval using rotated complex wavelet filters. *IEEE Transaction on Systems, Man, and Cybernetics-Part B: Cybernetics*, 6(36):1273–1282, 2006.

[109] Maria Konte, Nick Feamster, and Jaeyeon Jung. Dynamics of online scam hosting infrastructure. In *proceedings of Passive and Active Network Measurement*, pages 219–228. Springer, 2009.

[110] Kryo. iodine. `http://code.kryo.se/iodine/`, visited on December 18, 2015.

[111] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for

deep packet inspection. *ACM SIGCOMM Computer Communication Review*, 36(4):339–350, 2006.

[112] Karypis Lab. Data clustering software. `http://glaros.dtc.umn.edu/gkhome/views/cluto`, visited on December 18, 2015.

[113] Bjornar Larsen and Chinatsu Aone. Fast and effective text mining using linear-time document clustering. In *Knowledge Discovery and Data Mining*, KDD'99, pages 16–22, New York, NY, USA, 1999. ACM.

[114] Wenke Lee. Applying data mining to intrusion detection: the quest for automation, efficiency, and credibility. *ACM SIGKDD Explorations Newsletter*, 4(2):35–42, 2001.

[115] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14:533–567, 2000.

[116] Ru Li, Ou-Jie Xi, Bin Pang, Jiao Shen, and Chun-Lei Ren. Network application identification based on wavelet transform and k-means algorithm. In *proceedings of the IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS2009)*, volume 1, pages 38–41, November 2009.

[117] Wei Li, Marco Canini, Andrew W. Moore, and Raffaele Bolla. Efficient application identification and the temporal and spatial stability of classification schema. *Elsevier Computer Network*, pages 790–809, 2009.

[118] Cornell University Library. e-Print Archive. `http://arxiv.org/`, visited on December 18, 2015.

[119] Kriangkrai Limthong, Fukuda Kensuke, and Pirawat Watanapongse. Wavelet-based unwanted traffic time series analysis. In *proceedings of the International Conference on Computer and Electrical Engineering*, pages 445–449, 2008.

[120] Chieh-Yen Lin, Cheng-Hao Tsai, Ching-Pei Lee, and Chih-Jen Lin. Large-scale logistic regression and linear support vector machines using spark. In *proceedings of the IEEE International Conference on Big Data*, pages 519–528. IEEE, 2014.

[121] Carl Livadas, Robert Walsh, David E. Lapsley, and W. Timothy Strayer. Using machine learning techniques to identify botnet traffic. In *proceedings of the 31st IEEE Conference on Local Computer Networks*, pages 967–974, Washington, DC, USA, 2006. IEEE Computer Society.

[122] Michael E. Locasto, Janak J. Parekh, Angelos D. Keromytis, and Salvatore J. Stolfo. Towards collaborative security and P2P intrusion detection. In *proceedings of the Information Assurance Workshop (IAW'05), from the Sixth Annual IEEE SMC*, pages 333–339, 2005.

[123] Michael E. Locasto, Janak J. Parekh, Sal Stolfo, and Vishal Misra. Collaborative distributed intrusion detection. Technical Report CUCS-012-04, 2004. `http://academiccommons.columbia.edu/item/ac:109765`, visited on December 18, 2015.

[124] Maurizio Martinelli Enrico Gregori Luca Deri, Simone Mainardi. Graph theoretical models of DNS traffic. In *proceedings of the 9th International Wireless Communications and Mobile Computing Conference, IWCMC*, pages 1162–1167, Jul 2013.

[125] M. Kucherawy, E. Zwicky. Domain-based message authentication, reporting and conformance (dmarc). `http://tools.ietf.org/html/draft-kucherawy-dmarc-base-01`, visited on December 18, 2015.

[126] M. Wong, W. Schlitt. Sender Policy Framework (SPF) for authorizing use of domains in e-mail, version 1. `https://www.ietf.org/rfc/rfc4408.txt`, visited on December 18, 2015.

[127] Spyros Makridakis and Michele Hibon. Arma models and the box–jenkins methodology. *Journal of Forecasting*, 16(3):147–163, 1997.

[128] Christopher D. Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing*. MIT Press, 2002.

[129] MathWorks. MATLAB, 2000-2015. `http://www.mathworks.com/products/matlab/`, visited on December 18, 2015.

[130] MathWorks. MATLAB Coder, 2012. `http://www.mathworks.com/products/matlab-coder/`, visited on December 18, 2015.

[131] MathWorks. MATLAB Coder: `codegen` – generate C/C++ code from MATLAB code, 2012. `http://tinyurl.com/je6jem7`, visited on December 18, 2015.

[132] Geoffrey McLachlan and Thriyambakam Krishnan. *The EM algorithm and extensions*, volume 382. John Wiley & Sons, 2007.

[133] E Messmer. Downadup/conflicker worm: When will the next shoe fall, 2009. http://tinyurl.com/nlnv633.

[134] Serguei A. Mokhov. Study of best algorithm combinations for speech processing tasks in machine learning using median vs. mean clusters in MARF. In Bipin C. Desai, editor, *proceedings of C3S2E'08*, pages 29–43, Montreal, Quebec, Canada, May 2008.

[135] Serguei A. Mokhov. MARFCAT – MARF-based Code Analysis Tool. Published electronically within the MARF project, 2010–2015. `http://sourceforge.net/projects/marf/files/Applications/MARFCAT/`, visited on December 18, 2015.

[136] Serguei A. Mokhov. The use of machine learning with signal and NLP processing of source code to fingerprint, detect, and classify vulnerabilities and weaknesses with MARFCAT. Technical Report NIST SP 500-283, NIST, October 2011. `http://www.nist.gov/manuscript-publication-search.`

cfm?pub_id=909407, online e-print at http://arxiv.org/abs/1010.2511, visited on December 18, 2015.

[137] Serguei A. Mokhov. *Intensional Cyberforensics*. PhD thesis, September 2013. http://arxiv.org/abs/1312.0466, visited on December 18, 2015.

[138] Serguei A. Mokhov and Mourad Debbabi. File type analysis using signal processing techniques and machine learning vs. file unix utility for forensic analysis. In Oliver Goebel, Sandra Frings, Detlef Guenther, Jens Nedon, and Dirk Schadt, editors, *proceedings of the IT Incident Management and IT Forensics (IMF'08)*, LNI140, pages 73–85. GI, September 2008.

[139] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Formally specifying operational semantics and language constructs of Forensic Lucid. In Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, and Dirk Schadt, editors, *proceedings of the IT Incident Management and IT Forensics (IMF'08)*, volume 140 of *LNI*, pages 197–216. GI, September 2008. http://subs.emis.de/LNI/proceedings/proceedings140/gi-proc-140-014.pdf, visited on December 18, 2015.

[140] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Towards automatic deduction and event reconstruction using Forensic Lucid and probabilities to encode the IDS evidence. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, *proceedings of Recent Advances in Intrusion Detection RAID'10*, volume 6307 of

Lecture Notes in Computer Science (LNCS), pages 508–509. Springer Berlin Heidelberg, September 2010.

[141] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. The use of NLP techniques in static code analysis to detect weaknesses and vulnerabilities. In Maria Sokolova and Peter van Beek, editors, *proceedings of Canadian Conference on AI'14*, volume 8436 of *LNAI*, pages 326–332. Springer, May 2014. Short paper.

[142] MongoDB, Inc. Mongodb. `https://www.mongodb.org/`, visited on December 18, 2015.

[143] David Moore, Colleen Shannon, and k claffy. Code-red: a case study on the spread and victims of an internet worm. In *proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 273–284, New York, NY, USA, 2002. ACM.

[144] Motorola. Efficient polyphase FIR resampler for `numpy`: Native C/C++ implementation of the function *upfirdn()*, 2009. `http://code.google.com/p/upfirdn/source/browse/upfirdn`, visited on December 18, 2015.

[145] Cathal Mullaney. Morto worm sets a (DNS) record. `http://www.symantec.com/connect/blogs/morto-worm-sets-dns-record`, visited on December 18, 2015.

[146] Kevin P. Murphy. HMM toolbox, 2002–2014. `http://www.cs.ubc.ca/~murphyk/Software/HMM/hmm_download.html`,.

[147] Yacin Nadji, Manos Antonakakis, Roberto Perdisci, and Wenke Lee. Connected colors: Unveiling the structure of criminal networks. In SalvatoreJ. Stolfo, Angelos Stavrou, and CharlesV. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, pages 390–410. Springer, 2013.

[148] Saeed Nari and Ali A. Ghorbani. Automated malware classification based on network behavior. In *proceedings of the 2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 642–647, 2013.

[149] Jose Nazario. Blackenergy DDoS bot analysis. 2007.

[150] Jose Nazario and Thorsten Holz. As the net churns: Fast-flux botnet observations. In *proceedings of the Third International Conference on Malicious and Unwanted Software*, MALWARE'08, pages 24–31. IEEE, 2008.

[151] Sang-Kyun Noh, Joo-Hyung Oh, Jae-Seo Lee, Bong-Nam Noh, and Hyun-Cheol Jeong. Detecting P2P botnets using a multi-phased flow model. In *Third International Conference on Digital Society*, ICDS'09, pages 247–253, Washington, DC, USA, 2009. IEEE Computer Society.

[152] The Federal Bureau of Investigation. Botnets 101 what they are and how to avoid them, June 2013. `http://tinyurl.com/nmy7add`, visited on December 18, 2015.

[153] Y. Okada, S. Ata, N. Nakamura, Y. Nakahira, and I. Oka. Comparisons of machine learning algorithms for application identification of encrypted traffic. In *proceedings of the 10th International Conference on Machine Learning and Applications and Workshops (ICMLA)*, volume 2, pages 358–361, December 2011.

[154] opendns.com. The role of DNS in botnet command & control. `http://tinyurl.com/hxyp7or`, visited on December 18, 2015.

[155] Oracle. Download Java for Windows. `https://www.java.com/en/download/`, visited on December 18, 2015.

[156] Samir Ouchani, Otmane Ait'Mohamed, and Mourad Debbabi. A non-convex classifier support for abstraction-refinement framework. In *24th International Conference on Microelectronics (ICM)*, pages 1–4, 2012.

[157] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet background radiation. In *proceedings of IMC'04*, pages 1–14. ACM, 2004.

[158] Fragkiskos Papadopoulos, Constantinos Psomas, and Dmitri Krioukov. Replaying the geometric growth of complex networks and application to the AS internet. *SIGMETRICS Perform. Eval. Rev.*, 40(3):104–106, January 2012.

[159] Joey Paquet. Distributed eductive execution of hybrid intensional programs. In *proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pages 218–224, July 2009.

[160] Emanuele Passerini, Roberto Paleari, Lorenzo Martignoni, and Danilo Bruschi. Fluxor: Detecting and monitoring fast-flux service networks. In *proceedings of Detection of intrusions and malware, and vulnerability assessment*, pages 186–206. Springer, 2008.

[161] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[162] Yi Peng, Gang Kou, A. Sabatka, Zhengxin Chen, D. Khazanchi, and Yong Shi. Application of clustering methods to health insurance fraud detection. In *proceedings of the 2006 International Conference on Service Systems and Service Management*, volume 1, pages 116–120, 2006.

[163] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53(6):864–881, 2009.

[164] Roberto Perdisci, Igino Corona, David Dagon, and Wenke Lee. Detecting malicious flux service networks through passive analysis of recursive DNS traces. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 311–320. IEEE, 2009.

[165] Roberto Perdisci, Igino Corona, and Giorgio Giacinto. Early detection of malicious flux networks via large-scale passive DNS traffic analysis. *Dependable and Secure Computing, IEEE Transactions on*, 9(5):714–726, 2012.

[166] Nicole Perlroth. Security experts expect shellshock software bug in bash to be significant, September 2014. `http://tinyurl.com/n9joxuh`, visited on December 18, 2015.

[167] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. Technical report, Technical report, Computer Science Laboratory, SRI International, 2007.

[168] Quantcast. Rankings. `https://www.quantcast.com/top-sites`, visited on December 18, 2015.

[169] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[170] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[171] Ashkan Rahimian, Raha Ziarati, Stere Preda, and Mourad Debbabi. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*, pages 408–425. Springer, 2014.

[172] Liva Ralaivola, Sanjay Joshua Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. *Neural Networks*, 18(8):1093–1110, 2005.

[173] Revelli, Alberto and Leidecker, Nico. Heyoka: your fast&spoofed DNS tunnel. `http://heyoka.sourceforge.net/`, visited on December 18, 2015.

[174] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[175] Luis Javier Rodríguez and Inés Torres. Comparative study of the baum-welch and viterbi training algorithms applied to read and spontaneous speech recognition. In *Pattern Recognition and Image Analysis*, volume 2652 of *Lecture Notes in Computer Science*, pages 847–857. Springer Berlin Heidelberg, 2003.

[176] Christian Rossow, Christian J. Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten Van Steen, Felix C. Freiling, and Norbert Pohlmann. Sandnet: Network traffic analysis of malicious software. In *proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 78–88, 2011.

[177] Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[178] Gerard Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Boston, MA, USA, 1989.

[179] Rob Schreiber. MATLAB. *Scholarpedia*, 2(6):2929, 2007. `http://www.scholarpedia.org/article/MATLAB`, visited on December 18, 2015.

[180] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *proceedings of IEEE Symposium on Security and Privacy*, pages 38–49, Oakland, 2001.

[181] SecurityFocus. Windows anti-debug reference, 2009. `http://www.securityfocus.com/infocus/1893`, visited on December 18, 2015.

[182] securixLive. Securix-nsm project page, 2005-2009. `http://www.securixlive.com/knoppix-nsm/` , visited on December 18, 2015.

[183] Ivan Selesnick, Shihua Cai, Keyong Li, Levent Sendur, and A. Farras Abdelnour. MATLAB implementation of wavelet transforms. Technical report, Electrical Engineering, Polytechnic University, Brooklyn, NY, 2003. `http://taco.poly.edu/WaveletSoftware/`, visited on December 18, 2015.

[184] James G Shanahan and Laing Dai. Large scale distributed data science using apache spark. In *proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2323–2324. ACM, 2015.

[185] Reza Sharifnya and Mahdi Abadi. A novel reputation system to detect dga-based botnets. In *Computer and Knowledge Engineering (ICCKE), 2013 3th International eConference on*, pages 417–423. IEEE, 2013.

[186] Reza Sharifnya and Mahdi Abadi. Dfbotkiller: Domain-flux botnet detection based on the history of group activities and failures in DNS traffic. *Digital Investigation*, 12:15–26, 2015.

[187] Paria Shirani, Mohammad Abdollahi Azgomi, and Saed Alrabaee. A method for intrusion detection in web services based on time series. In *proceedings of the 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 836–841. IEEE, 2015.

[188] György J. Simon, Hui Xiong, Eric Eilertson, and Vipin Kumar. Scan detection: A data mining approach. In *proceedings of SDM 2006*, pages 118–129, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics. `http://www.siam.org/meetings/sdm06/proceedings/011simong.pdf`, visited on December 18, 2015.

[189] Prosenjit Sinha, Amine Boukhtouta, Victor Heber Belarde, and Mourad Debbabi. Insights from the analysis of the mariposa botnet. In *proceedings of the Fifth International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–9. IEEE, 2010.

[190] Ed Skoudis. The six most dangerous new attack techniques and what's coming next. In *RSA Conference (RSA'12)*, 2012.

[191] Ed Skoudis and Lenny Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[192] SkullSecurity. Dnsxss, 2015. `https://wiki.skullsecurity.org/Dnsxss`, visited on December 18, 2015.

[193] Yingbo Song, Angelos D. Keromytis, and Salvatore Stolfo. Spectrogram: A Mixture-of-Markov-Chains model for anomaly detection in web traffic. In *proceedings of the Network and Distributed System Security Symposium*, pages 121–135. Internet Society, February 2009.

[194] Aditya K Sood, Richard J Enbody, and Rohit Bansal. Dissecting spyeye–understanding the design of third generation botnets. *Computer Networks*, 57(2):436–450, 2013.

[195] Sourcefire. Snort: Open-source network intrusion prevention and detection system (IDS/IPS), 1999–2015. `http://www.snort.org/`, visited on 18 December, 2015.

[196] Eugene H. Spafford. The internet worm program: An analysis. *COMPUTER COMMUNICATION REVIEW*, 19, 1989.

[197] Etienne Stalmans and Barry Irwin. A framework for DNS based detection and mitigation of malware infections on a network. In *proceedings of Information Security South Africa (ISSA), 2011*, pages 1–8. IEEE, 2011.

[198] Michael Steinbach, George Karypis, Vipin Kumar, et al. A comparison of document clustering techniques. In *Knowledge Discovery and Data Mining Workshop on Text Mining*, volume 400, pages 525–526. ACM, 2000.

[199] Karen Stephenson and Marvin Zelen. Rethinking centrality: Methods and examples. *Social Networks*, 11(1):1–37, March 1989.

[200] Ben Stock, Jan Gobel, Markus Engelberth, Felix C Freiling, and Thorsten Holz. Walowdac-analysis of a peer-to-peer botnet. In *proceedings of the European Conference on Computer Network Defense (EC2ND)*, pages 13–20. IEEE, 2009.

[201] Salvatore J. Stolfo, Wenke Lee, Philip K. Chan, Wei Fan, and Eleazar Eskin. Data mining-based intrusion detectors: an overview of the Columbia IDS Project. *ACM SIGMOD Record*, 30(4):5–14, 2001.

[202] Brett Stone-Gross, Marco Cova, Bob Gilbert, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Analysis of a botnet takeover. *Security & Privacy, IEEE*, 9(1):64–72, 2011.

[203] Jiang Su and Harry Zhang. A fast decision tree learning algorithm. In *proceedings of the 21st National Conference on Artificial Intelligence*, volume 1 of *AAAI'06*, pages 500–505. AAAI Press, 2006.

[204] P. Ször and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, pages 123–144. Virus Bulletin, 2001.

[205] Peter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[206] Tamas Szerb. NSTX tunneling network-packets over DNS. `http://savannah.nongnu.org/projects/nstx/`, visited on December 18, 2015.

[207] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. Botfinder: finding bots in network traffic without deep packet inspection. In *CoNEXT*, CoNEXT '12, pages 349–360, New York, NY, USA, 2012. ACM.

[208] Carlos H. C. Teixeira, Arlei Silva, and Wagner Meira Jr. Min-hash fingerprints for graph kernels: A trade-off among accuracy, efficiency, and compression. *JOURNAL OF INFORMATION AND DATA MANAGEMENT*, 3(3):227–242, 2012.

[209] Sandeep A. Thorat, Amit K. Khandelwal, Bezawada Bruhadeshwar, and K. Kishore. Payload content based network anomaly detection. In *proceedings of the First International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2008)*, pages 127–132, 2008.

[210] ThreatTrack Security. ThreadAnalyzer: Dynamic sandboxing and malware analysis (formerly GFI SandBox). `http://www.threattracksecurity.com/enterprise-security/sandbox-software.aspx`, visited on December 18, 2015.

[211] Virus Total. Free online virus, malware and url scanner. `https://www.virustotal.com`, visited on December 18, 2015.

[212] Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. A malware instruction set for behavior-based analysis. Technical report, 2011. `www.mlsec.org/malheur/docs/mist-tr.pdf`, visited on December 18, 2015.

[213] Tom van Leijenhorst, Kwan-Wu Chin, and Darryn Lowe. On the viability and performance of DNS tunneling. `www.uow.edu.au/~kwanwu/DNSTunnel.pdf`, visited on December 18, 2015.

[214] Emil Iordanov Vassev. General architecture for demand migration in the GIPSY demand-driven execution engine. Master's thesis, June 2005. `http://spectrum.library.concordia.ca/8681/`, visited on December 18, 2015.

[215] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, August 2010.

[216] VMware. Vmware server, 2010. `http://www.vmware.com/products`, visited December 18, 2015.

[217] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *proceedings of Recent Advances in Intrusion Detection*, pages 203–222. Springer, 2004.

[218] Florian Weimer. Passive DNS replication. In *proceedings of the FIRST Conference on Computer Security Incident*, page 98, 2005.

[219] Sean Whalen, Nathaniel Boggs, and Salvatore J. Stolfo. Model aggregation for distributed content anomaly detection. In *proceedings of the 2014 Workshop on Artificial Intelligent and Security*, pages 61–71. ACM, 2014.

[220] Whois. Domain names & identity for everyone. `http://www.whois.com/`, visited on December 18, 2015.

[221] Georg Wicherski. pehash: A novel approach to fast malware clustering. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, LEET'09, 2009.

[222] Marek S Wiewiórka, Antonio Messina, Alicja Pacholewska, Sergio Maffioletti, Piotr Gawrysiak, and Michał J Okoniewski. Sparkseq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, page btu343, 2014.

[223] Wikipedia. Dll injection, 2009. `http://en.wikipedia.org/wiki/DLL_injection#cite_note-Waddington-9`, visited on December 18, 2015.

[224] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[225] Meng-Da Wu and Stephen D. Wolfthusen. Network forensics of partial SSL/TLS encrypted traffic classification using clustering algorithms. In Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, and Dirk Schadt, editors, *proceedings of the IT Incident Management and IT Forensics (IMF'08)*, LNI140, pages 157–172, September 2008.

[226] JianYun Xu, Andrew H. Sung, Patrick Chavez, and Srinivas Mukkamala. Polymorphic malicious executable scanner by api sequence analysis. In *proceedings of the Fourth International Conference on Hybrid Intelligent Systems*, HIS '04, pages 378–383. IEEE Computer Society, 2004.

[227] Sandeep Yadav and AL Narasimha Reddy. Winning with DNS failures: Strategies for faster botnet detection. In *Security and privacy in communication networks*, pages 446–459. Springer, 2012.

[228] Sandeep Yadav, Ashwath Kumar Krishna Reddy, AL Narasimha Reddy, and Supranamaya Ranjan. Detecting algorithmically generated domain-flux attacks with DNS traffic analysis. *Networking, IEEE/ACM Transactions on*, 20(5):1663–1677, 2012.

[229] Vinod Yegneswaran, Paul Barford, and Dave Plonka. On the design and use of internet sinks for network abuse monitoring. In *proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 146–165, 2004.

[230] Ting-Fang Yen and Michael K. Reiter. Traffic aggregation for malware detection. In *proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 207–227, Berlin, Heidelberg, 2008. Springer-Verlag.

[231] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX*, 37(4):45–51, 2012.

[232] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[233] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[234] Stefano Zanero. Analyzing TCP traffic patterns using self organizing maps. In *Image Analysis and Processing (ICIAP 2005)*, pages 83–90. Springer, 2005.

[235] Stefano Zanero and Sergio M. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *proceedings of the 2004 ACM symposium on Applied computing*, pages 412–419, 2004.

[236] Stefano Zanero and Giuseppe Serazzi. Unsupervised learning algorithms for intrusion detection. In *Network Operations and Management Symposium (NOMS 2008)*, pages 1043–1048, 2008.

[237] Kim Zetter. How Digital Detectives Deciphered Stuxnet, the Most Menacing Malware in History, July 2011. `http://tinyurl.com/oa7v6bx`, visited on December 18, 2015.

[238] Dazhi Zhang, Donggang Liu, Christoph Csallner, David Kung, and Yu Lei. A distributed framework for demand-driven software vulnerability detection. *J. Syst. Softw.*, 87:60–73, January 2014.

[239] Ying Zhao and George Karypis. Criterion functions for document clustering: Experiments and analysis. Technical report, University of Minnesota, 2002.

[240] Ying Zhao, George Karypis, and Usama Fayyad. Hierarchical clustering algorithms for document datasets. *Data Mining and Knowledge Discovery*, 10(2):141–168, 2005.

[241] Shi Zhong and Joydeep Ghosh. Generative model-based document clustering: a comparative study. *Knowledge and Information Systems*, 8(3):374–384, 2005.