André Ricardo Abed Grégio

Malware Behavior

Comportamento de Programas Maliciosos

Campinas
2012

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

André Ricardo Abed Grégio

Malware Behavior

Comportamento de Programas Maliciosos

Doctorate thesis presented to the School of Electrical and Computer Engineering in partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering. Concentration area: Computer Engineering.

Tese de doutorado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientador (Tutor): Prof. Dr. Mario Jino
Co-orientador (Co-Tutor): Prof. Dr. Paulo Licio de Geus

Este exemplar corresponde à versão final da tese defendida pelo aluno, e orientada pelo Prof. Dr. Mario Jino.

Campinas
2012

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

G861c	Grégio, André Ricardo Abed Comportamento de programas maliciosos / André Ricardo Abed Grégio. --Campinas, SP: [s.n.], 2012. Orientador: Mario Jino. Coorientador: Paulo Licio de Geus. Tese de Doutorado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação. 1. Redes de computadores - Medidas de segurança. 2. Tecnologia da informação - Segurança. 3. Software - Segurança. 4. Virus de computador. 5. Taxonomia. I. Jino, Mario, 1943-. II. Geus, Paulo Licio de, 1956-. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Título em Inglês: Malware behavior
Palavras-chave em Inglês: Computer networks - Security measures, Information technology - Security, Software - Security, Computer virus, Taxonomy
Área de concentração: Engenharia de Computação
Titulação: Doutor em Engenharia Elétrica
Banca examinadora: Mario Jino, Eleri Cardozo, Ricardo Dahab, Adriano Mauro Cansian, Luiz Fernando Rust da Costa Carmo
Data da defesa: 28-11-2012
Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE DOUTORADO

Candidato: André Ricardo Abed Grégio

Data da Defesa: 28 de novembro de 2012

Título da Tese: "Malware Behavior"

Prof. Dr. Mario Jino (Presidente): Mario Jino

Prof. Dr. Adriano Mauro Cansian:

Dr. Luiz Fernando Rust da Costa Carmo:

Prof. Dr. Eleri Cardozo: Eleri Cardozo

Prof. Dr. Ricardo Dahab: Ricardo

To my Family.

Acknowledgments

I would like to thank my advisors, Professors Paulo Lício de Geus and Mario Jino, for supporting me during the past years, listening to my frequent complaints and giving me freedom to pursue my goals. Their advice and ideas were essential not only to glue the pieces of this thesis together, but to allow me to grow as a scientist and as a better person.

I thank the Brazilian Government, my employer, for investing in me as a researcher.

I thank my former Bachelor and Masters research advisors, Professor Adriano Mauro Cansian and Dr. Rafael Duarte Coelho dos Santos, for their friendship and for always standing by my side since I met them. I also thank Otavio Carlos Cunha da Silva for the very same reasons. They prevented me from “roundhouse kicking” people on the road. =]

I also thank my friends, students and research colleagues for all the support, chit-chatting, laughs, coding, paper writing, hamburger eating and the late night coffee time, mainly Vitor Monte Afonso, Dario Simões Fernandes Filho, Victor Furuse Martins, Isabela Liane de Oliveira, Roberto Alves Gallo Filho, Henrique Kawakami, Eduardo Ellery and Alexandre Or Cansian Baruque. If I forgot anyone, I am sorry, it is just because my cache is volatile...

I finally thank my family, especially my wife Fabiana, my brother (Nê), my dad (Bastian) and my mom (Jamile). Their support, patience, love and friendship make my life more meaningful.

Oh, so they have Internet on computers now!

Homer Simpson

Resumo

Ataques envolvendo programas maliciosos (*malware*) são a grande ameaça atual à segurança de sistemas. Assim, a motivação desta tese é estudar o comportamento de *malware* e como este pode ser utilizado para fins de defesa. O principal mecanismo utilizado para defesa contra *malware* é o antivírus (AV). Embora seu propósito seja detectar (e remover) programas maliciosos de máquinas infectadas, os resultados desta detecção provêem, para usuários e analistas, informações insuficientes sobre o processo de infecção realizado pelo *malware.* Além disso, não há um padrão de esquema de nomenclatura para atribuir, de maneira consistente, nomes de identificação para exemplares de *malware* detectados, tornando difícil a sua classificação. De modo a prover um esquema de nomenclatura para *malware* e melhorar a qualidade dos resultados produzidos por sistemas de análise dinâmica de *malware*, propõe-se, nesta tese, uma taxonomia de *malware* com base nos comportamentos potencialmente perigosos observados durante vários anos de análise de exemplares encontrados em campo. A meta principal desta taxonomia é ser clara, de simples manutenção e extensão, e englobar tipos gerais de malware (*worms, bots, spyware*). A taxonomia proposta introduz quatro classes e seus respectivos comportamentos de alto nível, os quais representam atividades potencialmente perigosas. Para avaliá-la, foram utilizados mais de 12 mil exemplares únicos de malware pertencentes a diferentes classes (atribuídas por antivírus). Outras contribuições provenientes desta tese incluem um breve histórico dos programas maliciosos e um levantamento das taxonomias que tratam de tipos específicos de malware; o desenvolvimento de um sistema de análise dinâmica para extrair perfis comportamentais de malware; a especialização da taxonomia para lidar com exemplares de malware que roubam informações (*stealers*), conhecidos como *bankers*, a implementação de ferramentas de visualização para interagir com traços de execução de malware e, finalmente, a introdução de uma técnica de agrupamento baseada nos valores escritos por malware na memória e nos registradores.

Palavras-chave: Segurança de Redes e Computadores. Comportamento de Programas Maliciosos. Análise Dinâmica de Malware. Taxonomia de Malware. Classificação de Malware. Visualização de Dados de Segurança.

Abstract

Attacks involving malicious software (malware) are the major current threats to systems security. The motivation behind this thesis is to study malware behavior with that purpose. The main mechanism used for defending against malware is the antivirus (AV) tool. Although the purpose of an AV is to detect (and remove) malicious programs from infected machines, this detection usually provides insufficient information for users and analysts regarding the malware infection process. Furthermore, there is no standard naming scheme for consistently labeling detected malware, making the malware classification process harder. To provide a meaningful naming scheme, as well as to improve the quality of results produced by dynamic analysis systems, we propose a malware taxonomy based on potentially dangerous behaviors observed during several years of analysis of malware found in the wild. The main goal of the taxonomy is, in addition to being simple to understand, extend and maintain, to embrace general types of malware (e.g., worms, bots, spyware). Our behavior-centric malware taxonomy introduces four classes and their respective high-level behaviors that represent potentially dangerous activities. We applied our taxonomy to more than 12 thousand unique malware samples from different classes (assigned by AV scanners) to show that it is useful to better understand malware infections and to aid in malware-related incident response procedures. Other contributions of our work are: a brief history of malware and a survey of taxonomies that address specific malware types; a dynamic analysis system to extract behavioral profiles from malware; specialization of our taxonomy to handle information stealers known as bankers; proposal of visualization tools to interact with malware execution traces and, finally, a clustering technique based on values that malware writes into memory or registers.

Keywords: Network and Computer Security. Malicious Programs Behavior. Malware Dynamic Analysis. Malware Taxonomy. Malware Classification. Security Data Visualization.

List of Figures

List of Tables

Glossary

Antivirus A software to protect computers, either by scanning the filesystem in the search for malicious programs or by monitoring system's activities.

API Application Programming Interface, an interface to allow the communication between software objects.

Backdoor The goal of a backdoor is to allow the remote access and/or control of a system. It can be deployed as a standalone malicious program that opens a network port, can be a "feature" embedded (or compiled) into another program, or it can be a software security flaw.

Banker A type of malicious program whose main intent is to steal Internet Banking information.

BHO Browser Helper Object, a module, or plug-in, to add new functionalities for an Internet Explorer web browser. A malware sample can load a specially crafted BHO to change its victim's browser behavior, usually aiming to steal sensitive information.

Botnet A network composed of infected machines and remotely controlled by an attacker. The intent may vary from stealing sensitive data (such as credit card numbers) to performing distributed attacks.

C&C Command and Controller. The "manager" of a botnet; usually a server that launches commands and receives information from bots.

Downloader A program whose intent is to fetch and install another piece of software. In the context of this thesis, downloaders are small programs that obtain and install malware from the Internet.

Drive-by Download A downloaded performed automatically and stealthily, usually without the user's consent or awareness.

Dropper A program that carries malicious code inside itself to install it on an infected machine.

Dynamic Analysis The process of runnning a program under a controlled environment to monitor its interaction with the operating system.

Honeyclient A tool that acts as a client (usually a Web browser) and interacts with servers to verify the occurence of attacks.

Hooking A technique used to intercept events, calls and/or messages aiming to alter or monitor the operating system's behavior.

LCS Longest Common Subsequence, the longest set of string sequences (not consecutive) common to a set of sequences being compared.

Malware Malicious software. A program that perform harmful actions or behaves maliciously without the user's awareness or consent.

PAC Proxy Auto-Config, a file that defines a list of proxy servers in browsers so they can choose specific ones according to a certain URL. This type of file may be used by attackers to redirect users from legitimate domains to compromised servers in order to steal information.

Packer A program used to encrypt or obfuscate another program in a way to make its analysis (or reverse engineering) difficult.

Phishing A technique used by attackers to lure users into providing sensitive information, such as credit card numbers and credentials.

Registry A Microsoft Windows database that stores information about settings, configurations and components of the operating system.

Rootkit A special type of malware that usually operates at the kernel level. Rootkits may hide malicious programs, files and directories, and network communications. They may also disable security mechanisms and modify the infected system's behavior.

Sandbox	A controlled environment used to run malicious code without spreading the infection to other systems or through the network. In general, malware sandboxes are deployed using virtual machines or emulators.
Spyware	A type of malware specialized in collecting users' information (e.g., keystrokes, browsing profiles).
SSDT	System Service Dispatch Table, a Microsoft Windows kernel structure that contains the addresses of native functions, or system calls.
Static Analysis	The process of extracting information of a binary file without running it.
Trojan	A computer program that can present legitimate features, but that also has hidden functions whose intent is malicious.
Virus	One of the first popular types of malware. A computer virus is a code that appends itself to another program and spreads when this infected program is executed.
VMI	Virtual Machine Introspection, a tecnique to monitor internal states and events of a virtual machines without inserting a component inside the virtualized system. Thus, VMI allows one to "observe" a VM's inside information from outside.
VMM	Virtual Machine Monitor, a "probe" inserted in a layer located between the host and the guest in a virtualization system.
Worm	Another popular type of malware, a worm is a program capable of spreading itself autonomously. Worms usually scan for vulnerable services, exploit them and copy themselves to the compromised machine.
Zombie	A compromised system that can be remotely controlled by an attacker, the owner of a botnet or a C&C. Zombies are also known as "botclients" or simply "bots".

Contents

Chapter 1

Introduction

In this thesis, we cover several applications regarding malware behavior, such as classification, detection, visualization, clustering and so on. Due to this diversity and to the fact that the "converging factor" of all applications presented in this text relies on malware behavior, we chose this thesis' current short title. However, it does not mean that we cover all aspects of malware behavior nor that this thesis is a final word on the subject.

1.1 Motivation

The dissemination of malicious programs through computer networks, mainly the Internet, has been a major threat to the security of interconnected systems. Malicious programs, commonly referred to as **malware**, can be understood as applications whose development's intent is to compromise a system. Those applications are also commonly named as viruses, worms, Trojans, backdoors, keyloggers, and so on.

The increasingly growing interaction among several types of devices—notebooks, desktops, servers, tablets, cell phones, television sets, video game consoles—allied to the wide offer of (vulnerable) services creates an ideal scenario for attackers. In addition, careless, naive or unaware users that are continuously online, eager to obtain and to provide information on the Internet, contribute to the success of malware attacks.

One of the greatest motivations for malware attacks is the underground economy that is currently established [53] [68] [145], based on compromised infrastructures renting (e.g., network-connected systems that are invaded and remote controlled by attackers, such as botnets), sensitive information stealing (e.g., Internet Banking credentials, usernames and passwords of e-mail accounts, credit card numbers) [148], unsolicited messages (e.g., spam, fake product offers) [91] [144] and advertisement clicking [146] [90].

Malware threats have been reaching alarming levels so high that even simple activities, such as Web browsing [29] [40], social networking [149] [170] and the use of cell phones to access the Internet and run the so-called *apps* [18] [46] became risky. An article from the IBM Systems Magazine [70] evaluates the cost of data breaches in several countries to be of millions of dollars and blames the losses mainly to malware activity.

As time goes by, it is possible to notice a greater incidence of attacks performed by specific groups (or "families") of malware during certain intervals. For instance, in the last few years there has been an increase of botnet-related perceived activity, encouraging researchers to perform attempts (sometimes frustrated) to shut those malicious networks off [147] [171] [136] [32]. The particular incidence of certain malware families usually causes serious security incidents with global proportions, such as the StuxNet [52]. More examples of notorious past incidents involving malware are discussed in Chapter 2.

Although malware families tend to exploit the vulnerabilities that are "popular" at a determined time frame, it is common that they revisit older attack techniques through the sharing of code, and consequently functionalities, which perform those malicious activities, such as scanning engines, mutation of code excerpts, hiding mechanisms, etc. This code sharing leads to "mutations" that can cause the bypass of security barriers and incur in less effective detection rates for samples belonging to an already known family. However, the mutant sample keeps the malicious behavior "inherited" by the malware ancestors.

Thus, the mere identification of a program as being a known malware (already collected and analyzed, maybe defeated) allows for efficient and effective incident response. The countermeasures taken, for its part, can facilitate the damage containment process, decrease losses and mitigate side infections through security blocking rules and patch application.

Currently, one of the most popular defense mechanism against malware is still the antivirus (AV). Basically, an AV is a program that scans files or monitors predefined actions to search for evidences of malicious activities occurring in a system. Usually, AV engines rely on two techniques to identify malicious programs: pattern matching of signatures against a database of known malware and heuristics related to certain behaviors observed in malicious programs while compromising a system. In signature-based malware detection, a binary file is usually broken into small chunks of code, which are then scanned for patterns within the AV signature database. Hence, if one or more chunks of the scanned file are found in the signature database as pertaining to a specific malware family, the AV assigns a label identifying the file. In heuristic-based detection, a monitored file is executed in a minimal emulator and its behavior is evaluated to check for known suspicious activities. If an evidence of malicious behavior is found, the AV launches an alert and takes a measure (block, remove, quarantine the suspicious file or leave the decision to the user).

The major issue regarding AV engines is the frequent and increasing rise of malware variants. Malware variants correspond to previously identified malware families modified until they do not match a known signature or until they are able to evade, to compromise or to subvert the protection mechanisms in order to remain stealthy. To corroborate this assertive, a US DHS (Department of Homeland Security) report alleges that "*A/V and IDS/IPS approaches are becoming less effective because malware is becoming increasingly sophisticated (...) Malware polymorphism is outpacing signature generation and distribution in A/V and IDS/IPS* [159]." Variants usually need to be addressed individually and manually by AV analysts so that they are able to produce a new detection signature and update their products. In addition to this fact, detection heuristics may need modifications to encompass a new variant of a known family; these changes should be carefully handled to avoid false-positives or, even worse, false negatives of previously known malware. False positives and negatives are another great difficulty regarding signature/heuristics generation: the erroneous identification of an inoffensive user application as malicious may lead to its deletion, blocking or quarantine, causing a “denial of service”, as the decrease of usability is not a desirable feature for AV vendors.

Another issue that must be taken into account is that malware developers are embedding self-defense mechanisms to their products, i.e. current malware can disable operating system native protection (e.g., firewall, AV, security plugins, updates), can verify if it is under some kind of analysis and do not present its malicious behavior (e.g., by modifying its execution on-the-fly), can be packed in a way that avoids analysis and detection (e.g., checking its integrity in memory), can disguise itself as a system application, a legitimate software or a fake antivirus, and so on. Therefore, the tasks of avoiding the disabling of security mechanisms, identifying malware in an efficient and accurate way without interfering with users experience, and extending detection capabilities without causing overheads are becoming harder and harder. The scenario is not the most favorable either: the malware variants, the social networking exposition and the variety of potentially malicious applications, the cell phones and tablets becoming closer to traditional computer system (in processing power and functionality), and the establishment of cloud computing-based systems contribute to the abundance of “attack points” that lead to malware compromise, dissemination and persistence.

Apart from those aforementioned issues, the AV developers’ community is still not tied to a common standard to classify detected malware samples. This slows the malware-related incident response procedures, turning them ineffective in certain cases. To illustrate this comment, lets think of AV engines that identify a program as being malicious based solely on their packer[1]. Packing features or even the identification of one specific facet of the evaluated binary may lead to wrong assumptions about the extent of its damages or about its attacking behavior. For instance, malware samples may be infected by another malware, and the AV will probably detect the “infector”, launching an alert and assigning a misleading label whereas the behavior of the “infected” file, i.e. the actual malware sample, can be missed [163]. Furthermore, diverging naming assignments can also confuse classification schemes, as the lack of a common standard makes each vendor to assign its own label in its own format. Latter in this thesis we are able to verify that the same malware sample can be identified as being from distinct classes, families and versions depending on the AV vendor, all of them formatted in a peculiar manner.

Thus, to guarantee that a countermeasure is effective, i.e. to remove the malicious code and its associated artifacts (objects and changes in the system), we need in most cases a manual intervention. This is required due to the fact that not all AV engines are able to undo the performed actions, as there is not a general procedure that embraces all malware samples in activity, and compromised systems are usually not trustworthy anymore. Aside from this, some more epidemic malware infections may have removal applications or automated routines available, which can be innocuous in face of variants. However, the success of achieving a correct and functional removal procedure (manual or automated) is unrelated to the label assigned by an AV engine, which can be the wrong one.

Although the usual method of operation of AV engines rely principally on signature-based

[1] In the scope of this thesis, a packer is a program used to encrypt or compress a malicious file to produce an obstacle to detection and analysis techniques. For more information about packers, we point to the work of Jacob et al. [73].

detection, there has been an increasing of interest in heuristics. A well-crafted heuristic may result in the substitution of dozens of signatures, taking some burden off the AV updating process. However, the generation of an adequate heuristic to identify a malware sample or class requires knowledge about the related behavior. This behavior consists of the actions performed on the target operating system, mainly those that denote abnormal, suspicious or dangerous activities. Thus, there are dynamic analysis systems that serve as a complementary option to the limited emulators present in AV engines. Those systems have been improved and refined in a constant basis, making them a viable option to extract malware behavior. Dynamic malware analysis systems leverage several advanced techniques to monitor system by running samples in a controlled environment so as to observe the infection while it is being performed. These systems are based on several monitoring methods, from the instrumentation of complex emulators to the capture of kernel system calls within the target operating system [49].

It is not unusual that AV vendors develop their own dynamic malware analysis systems to sell and to make internal use of them—a.k.a sandboxes—or provide financial support to third-party developers, but there are also publicly available solutions that can be accessed through the Internet, such as *Anubis*[2], *ThreatExpert*[3] and *CWSandbox*[4]. A sandbox is, within the scope of this thesis, a restrict and controlled environment that serves to the purpose of malware execution, and whose aim is to avoid damage to external systems through the use of network traffic filtering and blocking, and limiting the running time. In general, malware samples run for about four or five minutes inside the sandbox and then they are terminated. During the execution, the actions related to the sample under analysis are monitored, as well as the side effects of these actions. After the timeout, this kind of system provides an activity report that can be analyzed.

Though a malware activity report may provide insights about the infection process and the sample behavior, the analysis of this report is left in charge of the user who submitted the malware sample. Therefore, even in the presence of a summary of the relevant activities, average users can get lost easily in the analysis process, which is more a log of activities than an analysis report. In addition, dynamic malware analysis systems can be used to classify samples, but they do not do it, to the best of our knowledge. In addition, due to limitations regarding the monitoring techniques that are used internally in sandboxes, modern malware can easily evade the analysis by stalling [92], exhibiting unsuspicious behavior [13] or detecting the execution environment [31]. This can be worsened if a sample mutates its behavior during additional executions (context or environment-sensitive), misleading the efforts of an analyst that compares outputs from distinct analysis systems. Thus, without a proper classification scheme, an analyst or user may become confused at the point of a security decision taking (e.g., disinfection procedures, defensive applications), turning the analysis reports into useless information.

There are other worthy points regarding malware classification, in addition to the previously discussed lack of a general taxonomy: the available research works address very specific taxonomies (see Chapter 2), the classification algorithms are applied to already known families and

[2] `http://anubis.iseclab.org`

[3] `http://www.threatexpert.com`

[4] `http://mwanalysis.org`

on a limited amount of classes (usually based on AV labels with the only purpose to cluster samples and verify the precision of the clusters), the proposed previous taxonomies and malware naming schemes are meaningless when dealing with modern malware, as these were attempts to describe distinct classes tied to specific behaviors in a strict scope. Nowadays, the boundaries that divide a malware class from another do not exist anymore, as modern malware samples are usually built on functional modules, presenting thus, at the same time, the behavior expected from rootkits, Trojan horses, worms, viruses, flooders, spammers and botclients.

Thence, on the one hand, we depend on malware detection and classification to build defensive solutions or perform better incident response. On the other hand, there is neither an official "mandatory" standard nor a document widely adopted by the majority of the security community to handle the malware menace. This causes confusing identification (naming) schemes that become obstacles when addressing malware-related incidents. Since the lack of an objective standard is disrupting to users and organizations that rely on security products output to carry on their daily activities, handling incidents caused by malware attacks is an imperative task.

However, the adequate handling of this type of incident requires a systematic process, divided into well defined steps that address (i) the collection of malware samples, (ii) the extraction of the behavior that these samples present, (iii) the mining of suspicious features contained within a behavior and the characterization of the prevalent ones, (iv) the categorization of samples into families that represent their prevalent behavior and (v) the generation of detection procedures to identify malware according to the perceived behavioral profile.

1.2 Objectives

Accordingly to the afore discussed points, the main goal of this thesis is to study malware behavior (using a practical approach) and how to use the information acquired from the observed behaviors to develop detection techniques and a more meaningful classification scheme that handle the damage that a malware can cause to an infected system. To this end, forwarding this point, we present the research work done for this thesis as a proposal of addressing the aforementioned needs, aiming to respond to malware incidents in a useful, organized and understandable manner. This is accomplished through the extraction of malicious behavior using a dynamic analysis system that we have been developing, the proposal of a behavior-based taxonomy for classification, the addition of modules for the detection of specialized malware (bankers) and for the visualization of the malware execution trace, and, finally, the introduction of a clustering algorithm to address malware behavior at a lower level.

1.3 Contributions

The field of malware research is very broad and plenty of effort has been spent by the security community to address the several types of threats that malicious code poses to Internet-connected systems.

In this thesis, the focus is on malware behaviors and in what can be done with them concerning computer systems defense. Thus, in addition to defining dangerous behaviors and malware classes based on them, we also apply the behavioral profiling on other topics, such as detection, classification, clustering, code reuse identification, visualization and incident response. The main contributions of this thesis are:

- A brief review of the history of malicious programs, since their beginning up to date.
- A discussion about the current malware naming scheme issues and antivirus labeling.
- A survey on the diversity of malware taxonomies according to their types (e.g., worms, spyware, bots).
- A definition of the different types of behavior that a malicious program can present, the description of a set of dangerous activities gathered from actually analyzed samples and the proposal of a behavior-centric malware taxonomy.
- An approach to detect information stealing malware that leverages a subset of the defined behaviors and image processing techniques.
- Interactive visualization tools to help in identifying similar behavioral patterns.
- A heuristic to cluster malware based on their memory and registers writing values, as well as an application of this technique to identify code reuse among malware samples from different families.

Furthermore, the work concerning this thesis is not limited to the contents of this document: the Ph.D. candidate has co-advised three Master students in Computer Science and two Undergraduate Students in Computer Engineering. As a result of this effort, articles and technical production have been generated, and skills in advising people were gained.

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

In **Chapter 2**, there is a brief review of the history of malicious programs, from before their formal conception (as automata or science fiction subjects) to their use in disrupting critical infrastructures. Issues regarding the current malware naming schemes used by antivirus companies are discussed with practical examples. Finally, the chapter contains a survey on available malware taxonomies, all of them based on the current concept of a malicious program tied to a specific behavior (e.g., a virus attaches its code to another program, a worm propagates autonomously, a Trojan horse lures the user into thinking he/she is installing a legitimate program, and so on).

In **Chapter 3**, we provide the notion of different types of execution behavior, such as active, passive, suspicious and neutral. Regarding those behavioral facets, we have chosen to analyze malware according to behaviors that allow information stealing, malicious communications and activities that cause changes on the compromised system. Hence, we select and describe a set of suspicious or dangerous activities that compose the behavior commonly observed on actual malware. Next, we define malware classes whose main behavior can be defined in terms of those activities and, based on this, we propose a new taxonomic scheme centered on malware behavior.

In **Chapter 4**, we evaluate and validate the proposed taxonomy on a large set of malware samples collected from honeypots, spam crawling, colleagues and public databases. We also describe the architecture designed to analyze malware samples in order to obtain the behavioral profiles used along this thesis.

In **Chapter 5**, we present a novel approach to detect bankers, i.e. malware samples whose main purpose is to steal online banking credentials from infected users, often using social engineering tricks. We present BanDIT, a prototype system developed to detect bankers using image processing techniques, network traffic patterns and file system modifications related to Internet browsing.

In **Chapter 6**, we introduce the visualization of behavioral profiles to help in the analysis of malware infection. To this end, we developed two tools that allow the visual-aided analysis of malware samples and the observation of similar behavior among samples from the same family. This work is the first step towards the perception that although malware families overlap their behavioral features, their (combination of) classes can be linearly separable.

In **Chapter 7**, we propose a heuristic to cluster malware based on the values that are written into addresses in memory and into registers during the execution. To this end, we developed a prototype system that tracks these memory/registers values and produces execution traces that characterize the analyzed samples. We leverage an algorithm to approximate the LCS (longest common subsequence) computation and to group malware into clusters that represent their family. Moreover, the proposed concepts are also applied in a code reuse identification problem, allowing us to pinpoint excerpts of shared code among different malware families.

In **Chapter 8**, we conclude by discussing limitations of the previously presented approaches and future work that can be explored to advance the research in the field, taking advantage of the topics we have addressed. Finally, we briefly summarize the results obtained in this thesis.

It is worth to note that, since the chapters are based on published papers, the analyzed set of samples may be different (regarding the set's size and variety).

Chapter 2

Background and Related Work

2.1 Introduction

A malicious software, or malware, is a set of instructions that runs on a system to make it do arbitrary activities on behalf of an attacker [140]. Malware evolved from the harmless concept of self-replicating automata to multipurpose stealthy code that can destroy physical assets. However, despite all orientation changes, the research field of malware analysis suffers from inconsistent naming schemes and lack of a useful taxonomy. In this chapter we summarize the history of malware, discuss issues related to malware naming schemes, briefly present concepts and requirements regarding the field and present work related to security-oriented taxonomies, more specifically those regarding malware.

2.2 A Brief History on Malicious Software

Computer virus. It is still possible to hear (or read) this term associated to a malicious program attack. However, a computer virus is one among several other existing malware classes.

Fred Cohen formally coined the term "**computer virus**" in 1983, defining it as a program that can infect other programs, changing them to include a possibly evolved copy of itself [35]. This contagion method allows a computer virus to propagate to other systems through other programs or the network. In his Ph.D. thesis, Cohen wrote and demonstrated the execution of a computer virus aimed at Unix systems [34].

Years before the viruses appeared, another "malware" class was already being discussed: **worm**, a kind of program that resembles the self-replicant automata described by John von Neumann in 1949 [108]. John F. Shoch and Jon A. Hupp, from Xerox Palo Alto Research Center, are given the credit for the term being used in the computer field. They debate about the development of worm programs for use in distributed computing in their 1982 paper [137]. Shochs and Hupp's worms were inspired by the first noticed worm in activity, the Creeper, written by Bob Thomas in 1971 and spread through ARPANET [30].

Shoch and Hupp describe their worms as applications that perform inoffensive tasks, such as the dissemination of messages through the network or the splitting of tasks to accomplish real-time animation. Despite this benign intent, they raise some concerns about the propagation control and the maintenance of a stable execution along the time. Those worries were not in vain: in 1988 Robert Morris Jr. launched the “Internet Worm”, a malicious program that explored a vulnerability in the Unix’s Sendmail application [113]. This worm contained itself a programming bug that escaped from the control of its developer, causing quick spurts of local and remote propagations. The result was the denial of service of the majority of the Internet-connected systems of that time [81]. The “Internet Worm” established the genesis of the massive malware attacks and led us to the current situation of pervasive threat.

Trojan horses, or Trojans, are the elements of another major malware class. The first use of the term in the security scope is credited to Daniel Edwards, a NSA researcher, on the second volume of a U.S. Air Force report from 1972 [7]. Edwards describes a **Trojan horse** as a computer program that is useful to a system, i.e. attractive enough to be executed, but that also has hidden functions that disguise its actual motivation. Edwards exemplify this kind of behavior as a text editor whose hidden function copies a file being edited to an attacker, partially or totally. The mathematical foundations for Trojan horses date from the work of Leonard Adleman in 1990 [4]. Adleman characterizes Trojan horses as programs that are incapable of infecting other programs, but able to imitate or injure them, the latter occurring only if certain conditions are met.

While the 80’s brought us a new paradigm on computer security, the 90’s can be considered the rise of malware attacks. In 1995, a **macro virus** (a virus written using a macro programming language) started to receive public attention due to its potential to spread rapidly: Concept, which executed as soon as an MS Word application with enabled macros opened a document. Since then, a user could be infected through a simple e-mail reading [62]. Back to 1989, there already were reports related to macro viruses that attacked Lotus123 spreadsheets, as well as detection procedures [66]. In 1996, several macro attacks and prevention possibilities were discussed, including polymorphic macro viruses [24].

The polymorphism feature in malware, i.e. the ability to mutate their code to evade signature detection, forced the antivirus companies to develop a new technique, the detection by code emulation. This technique implies that a malware sample runs in a controlled environment, usually an instructions emulator, on a predefined amount of cycles [17]. The first virus that was considered to be polymorphic was “Chameleon”, from 1990. Its main functionality was to mutate its signature every time it infected another file [138].

In 1998, the first virus to abuse Java applets and applications, called “StrangeBrew”, was developed as a proof-of-concept. It was written using the Java programming language [107]. This year is also remarkable due to the dissemination of the “Back Orifice” tool, which was commonly installed as some Trojan payload and acted as a backdoor, allowing for the complete remote control of the target machine [41]. In 1999, the “Melissa” virus caused denial of service on e-mail servers and large financial losses due to disinfection procedures. This macro virus/worm had a massive spreading, infecting a user’s address book and sending itself (on behalf of the infected user) to the e-mail contacts found [59].

Following the “Melissa” infection scheme, the “ILOVEYOU” worm attacked 45 million computers and caused 80 million dollar losses in 2000 [162]. From this time on, malware became more dangerous, complex and ubiquitous. In 2001, the “Code Red” worm scanned the Internet searching for vulnerable Windows IIS Web servers and exploiting them as they were found [174]. According to Moore et al. [102], "Code Red" infected more than 359 thousand Internet-connected machines in less than 14 hours, causing losses of approximately 2.6 billions of dollars. Another critical worm from 2001 was "Nimda", which also exploited vulnerable Microsoft IIS Web servers. This worm's attack was based on a multi-vector approach to assure a high rate of success in its infection process. Among other methods, "Nimda" e-mailed itself as an attachment based on contacts found in the compromised machine, copied itself through open network shares and used backdoors opened by "Code Red" [143]. In 2002, "Sdbot" and its variants combined scanning, exploiting, hiding, system's information obtaining and peer-to-peer spreading capabilities in a single malware sample [105], beginning the Instant Relay Chat bots age.

Other malware samples that caused impact on the Internet stability, security and economy are described next. The "Blaster" worm of 2003, which exploited a buffer overrun vulnerability in the Windows RPC (Remote Procedure Call) interface—allowing for the execution of arbitrary code—launched denial of service attacks against the Windows Update site [11]; The "Slammer" worm, also in 2003, spread fast (it infected 90% of vulnerable systems in 10 minutes) by exploiting a buffer overflow vulnerability in Microsoft SQL Server [101]. In addition to causing denial of service attacks by disabling database servers, its massive scanning capabilities overloaded networks by consuming all their bandwidth. "Sasser", of 2004, scanned for buffer overflow-vulnerable Windows LSASS (Local Security Authority Subsystem Service) components and exploited them through TCP port 445 [88]. The worm usually crashed the LSASS component, leading the infected system to frequent shutdowns.

The war of the worms had its outbreak in 2004, bringing attention not only to the massive malware infection rate, but to malware interactions, i.e. a malicious program terminating another type of malicious program [151]. The main threats of 2004 were "Bagle", "MyDoom" and "Netsky". The three of them were mass-mailer worms and installed their own SMTP engines to propagate through spam; the first two also operated as backdoors whereas "Netsky" tried to deactivate "Mydoom" and "Bagle" [139]. 2005 was the year of the "Zotob" worm—which spread by exploiting a Microsoft Windows Plug and Play buffer overflow vulnerability [129]—and the "Zlob" trojan—which disguised itself as an ActiveX Codec [156]. In 2007, the "Storm" worm struck, infecting about 10 million computers and gathering them into a botnet [116]. Also in this year, the "Zeus" crimeware became popular, due to its toolkit approach that allowed criminals to generate variants using a user-friendly interface, to its low price in the underground and to its Internet banking credentials and sensitive information (such as credit cards) stealing capabilities [20].

Malicious botnets and attacks from multipurpose malware continued to grow and, in 2008, more sophisticated pieces of malware began to be observed. "Torpig" not only stole online banking credentials and credit card data, but also user's personal information. This malware is installed in the victim's machine as part of the "Mebroot" rootkit, which replaces the system's Master Boot Record (MBR) and launches a drive-by download attack to turn the infected machine into a bot. An estimative about the size of a "Torpig" botnet counted more than 182 thousand machines in a 10 days monitoring study [147]. Another interesting botnet that arose in 2008 was the "Koobface". It had a massive reach by infecting users mainly through Facebook, spreading to a victim's social network friends [154]. The “Conficker” worm/bot appeared in late 2008 and spread very fast by exploiting the port TCP 445 through a remote procedure call (RPC), allowing the execution of arbitrary code [117]. Its features include a domain generation algorithm and self-updates via Web and P2P, resulting in an estimative of about 25 millions of infected victims [135].

The situation got worse in 2010 with the rise of “Stuxnet”. This complex malware targeted industrial control systems (its aim was to reprogram them to take their control in a stealthy way) and the pack included zero-day exploits, a Windows and a programmable logic controller (PLC) rootkit, antivirus evasion, process injection and hooking techniques, network-based spreading, updates using P2P and a command and control interface [52]. “Stuxnet” caused a global proportion crisis as it compromised real-world infrastructure, specifically computers from an Iranian nuclear power station [16]. In 2011 Hungarian researches discovered “DuQu”, a malware whose features (modular structure, injection mechanisms and fake-signed driver) refers to “Stuxnet” [19]. However, the main purpose of “DuQu”, at least initially, was to gather information from the compromised system through a keylogger module. Moreover, “DuQu” has an execution delay of 15 minutes, avoiding the dynamic analysis process of most available systems. As of this writing (1Q/2012), malware trends include attacks using man-in-the-browser techniques, smartphones targeting samples and open-source kits for banking trojans development. Also, the threat of the moment is known by the name of “Citadel”, an open-source malware project based on the “Zeus” Trojan that alluded to the Software-as-a-Service model (SaaS) [84]. “Citadel” evolves based on its customers additions, which includes the use of AES cryptography, trackers detection avoidance mechanism, security vendors websites blacklist and even a video recording module [130].

Despite the fact that most of the “press-famous” malware are referred as virus, worms, Trojans or bots, other malware classes do exist. For instance, Bishop describes in [22] ”rabbits and bacteria” and ”logic bombs” as other forms of malware. He defines a **logic bomb** as a program that attacks when externally triggered, giving as example a Trojan horse that is activated when an employee identification is removed from the payment database. In this case, the trigger to the logic bomb is the removal and its effect is the execution of a malware. **Rabbits** or **bacteria** are programs that reproduce themselves in a rapid fashion to exhaust some system's resource (e.g., disk space, memory, processor). This kind of behavior can be also observed in worms; thus, instead of a new class, rabbits can be considered a worm's subclass, which is indeed done in Peter Szor's terminology [150]. Szor also describes, among other classes:

- **Spyware**, which inadvertently collects system's or user's information and sends it through the network.

- **Rootkit**, a set of tools to attack and maintain access to an infected machine in a long lasting, stealthy way.

- **Keylogger**, a program that captures keystrokes and is usually installed stealthily.

- **Flooder**, a kind of tool that generates plenty of network traffic directed to a target in order to cause a denial of service.

- **Exploit**, code that aims to automatically gain privileged access or run arbitrary code in a target machine.

- **Auto-Rooter**, whose goal is to remotely break into a target and to gain access to administrative privileges.

- **Downloader**, a piece of code whose intent is to install itself in an infected system to download and extract malicious content, while avoiding security mechanisms.

- **Dropper**, an installer-like program that "carries" hidden viral code inside of itself. A dropper "mutates" to install (or generate) its carried payload and then to replicate without the need of its original code.

- **Injector**, a special kind of dropper that injects/installs malware code in the target's memory.

Viruses, worms, Trojan horses and the other aforementioned categories form the basis for a simplistic taxonomy that can be used to classify malware according to their expected behavior. However, this is not so simple as it seems to be, as we discuss in the next sections.

2.3 Confusion in the Battlefield

The analysis of the malware definitions from Section 2.2 lead us to notice that some classes intersect whereas others are variations of a common behavior. For instance, auto-rooters, exploits and flooders can be grouped into one bigger class, as they are usually considered to be set of tools with different intents. Regarding exploits and auto-rooters, even the intent is almost the same. If we take notice to the classes downloader, dropper and injector, it is clear that the main behavior is shared and that they differ by the way they install the attacking piece of code. Szor himself defines an injector as a slightly different dropper.

There are also minor variations that specialize a class; for example, we can consider that a rabbit is a specialized worm and that a keylogger is a form of spyware. Back to the 1980's, Spafford discussed the misuse of malware terminology in his report about the Internet Worm [142]. He mentions the press using the term "virus" instead of "worm", misleading the readers about the behavior observed during the attack. His justification on denoting the Morris program as a worm is quoted below, as the "behavior" factor in classifying malware attacks is the stepping stone of this thesis.

> "*A worm is a program that can run by itself and can propagate a fully working version of itself to other machines. It is derived from the word tapeworm, a parasitic organism that lives inside a host and saps its resources to maintain itself. A virus is a piece of code that adds itself to other programs, including operating systems. It cannot run independently-it requires that its "host" program be run to activate it. As such, it has a clear analog to biological viruses - those viruses are not considered alive in the usual sense; instead, they invade host cells and corrupt them, causing them*

> *to produce new viruses. The program that was loosed on the Internet was clearly a worm.*"

Since then, there have been a plethora of efforts towards the standardization of procedures to name malicious programs, such as the CARO (Computer Antivirus Researchers Organization) Malware Naming Scheme [141], created in the beginning of the 1990's. Their website, which is unfinished up to the date of this writing, states that CARO Naming Scheme accepts as malware types "virus", "trojan", "dropper", "intended", "kit" and "garbage". Bontchev, one of the proposer of the original CARO Naming Scheme, presents some updates to the scheme [25] by adding "pws", "dialer", "backdoor", "exploit" and "tool" to the previously accepted malware types; the last replaces the "kit" definition. According to CARO, **kit/tool** is a program designed to generate variants from a malware family; **intended** is a buggy program whose initial intention was to be a virus, but its execution fails (or crashes); **garbage** is attributed to meaningless or buggy programs that are not intended to be malware; **pws** stands for password stealer programs; a **dialer** tries to force the establishment of a dial-up connection; **backdoors** are programs that allow unauthorized or illegitimate access to compromised systems. The other types have already been defined previously.

Although Bontchev discusses important issues, such as the lack of a naming standard and of reliability in the naming process, the difficulty to enforce its use and the current messy naming schemes, we can notice that even CARO's proposed scheme is confusing: they can group viruses and worms in the same type whereas they consider backdoors and Trojans distinct enough to be in separate classes. However, Bontchev poses an interesting discussion topic: if some user's antivirus (AV) scanner detects a program as `MyDoom.D`, the user is unaware about its malicious behavior at the same time he is helpless, as the AV analysts are unable to determine the actual malware's behavior without analyzing the sample. Another problem arises when more than one AV scanner is used to improve the detection of malware. The use of multiple AV scanners can make worse the understanding of an infection regarding a malware successfully detected, as the identifications are meaningless and not uniform. To illustrate this, we sent a malware sample to VirusTotal [67], a service that scans a user-submitted file with several AV engines and shows the identified detection labels, if any. Table 2.1 shows this submission's results. It is possible to notice the confusing naming scheme provided by each AV as well as the questionable utility of the label to the extent of the user's comprehension.

Table 2.1: AV scanner and respective identification label.

AV scanner	**Detection label**
McAfee	`PWS-OnlineGames.bu`
Symantec	`Infostealer.Gampass`
Norman	`W32/Packedi_Upack.h`
Kaspersky	`Trojan-GameThief.Win32.OnlineGames.akyb`
McAfee-GW-Ed.	`Heuristic.BehavesLike.Win32.Packed.A`
PCTools	`Rootkit.Order`
Ikarus	`Virus.Win32.Virut`
Panda	`Trj/Lineage.ISP`

The naming issue is also discussed in the work of Raiu [123]. Raiu, from the Kaspersky Antivirus Laboratories in Romania, claims that when the VBS/VBSWG.J malware appeared, the media called it Kournikova due to its disguise under a so-claimed picture of the named tennis player while most of the AV engines detected it as malicious program under a mysterious label: SST. The AV vendors disagreed on this malware sample name, some claiming that SST was shorter and easier to remember and others that VBSWG was not too hard to remember and pronounce for the average AV user.

Thus, considering the "AnnaKournikova" malware case and taking the AV labels and the names assigned to the threats along the years into account, there is still a major question, which is the subject of this thesis: *What does this malware sample do?*

To try to start answering this question, let's evoke Leonard Adleman again. In his seminal work about a theory for characterizing computer viruses [4], Adleman formally defines certain types of viruses, such as the "carrier", which is defined as unable to cause injury but able to infect other programs under the right conditions, and the previously mentioned "Trojan horse". Those formalisms have their basis on the behavior observed from malware execution, e.g., if a sample is "pathogenic", "contagious", "benignant", if it "injures", "infects" or "imitates" another program, and so on.

Adleman's behavioral models allow to distinguish among possible malware infection strategies and elegantly encompass several classes of the malware fauna. For instance, the "carrier" definition can be used to denote the behavior of droppers, downloaders and injectors. This perception contributes to stress out the importance of a taxonomy that clearly indicates the risk posed by a malicious program, i.e. its behavior on a compromised system, and emphasizes the obsolescence, contradiction and incompleteness of the current naming schemes, which have been defective since their inception.

2.4 Systematics

Systematics, in the biological scope, can be defined as the study of the living organisms diversity and their relationships through time. Those relationships are depicted as phylogenies or evolutionary trees [167].

Systematics is a field that encompasses several responsibilities, such as providing scientific names to organisms and their related descriptions, maintaining sample collections, their classification, identification and distribution [99]. These responsibilities can be used to define the terms "taxonomy" and "scientific classification". Taxonomy is more specific to the naming scheme, the identification and the description of organisms whereas scientific classification is focused on their separation into hierarchically-related groups.

At first in this thesis, we discuss features of taxonomies and how they can be used to help in the malware analysis process. The objective is to create a ruleset to identify malware, to understand their behavior during an attack, as well as to improve the counter measures to perform effective incident response. We leave the discussion about classification to Chapter 7.

Thus, taxonomy can be defined as the science of species identification and naming, as well as their organization into classification systems [110]. Although the history of taxonomy refers back to ancient China (3000 B.C) [96], an organized and widely adopted standard only arose after Linnaeus published his work, *Systema Naturae* [94].

However, as taxonomy is a set of rules to identify, describe and organize elements, their basic concepts can be applied to several knowledge fields. In the following sections, we discuss: the use of taxonomies in the field of computer security, their scope and requirements; a brief survey of taxonomies that are focused on malware classes; a proposal for a new taxonomic scheme that is extensible, flexible, understandable and more generic than the current ones, at the same time that it takes advantage of their strengths.

2.5 Security-related Taxonomies

A number of taxonomies have been proposed to cover diverse computer security topics. For example, the works of Aslam [8], Krsul [85] and Landwehr address system's vulnerabilities and attacks [89]. The Neumann-Parker taxonomy addresses intrusions [109], whereas other taxonomies related to intrusion or threats to computer security are summarized by Lindqvist and Jonsson [93]. Taxonomies related specifically to malware are discussed in Section 2.6, as they pertain to the scope of this thesis.

Many are the requirements [65] that a taxonomy has to meet in order to accomplish its goal, i.e. to be clear, adaptive and applicable. Howard and Longstaff [69] and Amoroso [6] mention important properties of good taxonomies, which are listed as follows:

- **Mutually exclusive**, to assure that a sample fits into only one category.

- **Exhaustive (or completeness)**, so that the predefined categories include all possibilities of the subject under analysis.

- **Unambiguous**, to eliminate uncertainty and to allow the taxonomy application to be a clear process.

- **Repeatable**, so that others can repeat the taxonomic process and get the same results.

- **Acceptable and useful**, to allow it to be used by the community, serving as a reference and source of knowledge in the field.

In addition, Bishop states that a taxonomy requires **well defined terms**, so that there is no confusion about the meaning of a term [21]. Another interesting property of a taxonomy is that it should be **comprehensible**, i.e. either a field's expert or a merely interested person are able to understand it [93]. Therefore, a taxonomy should provide the **discernment** ability to its "users". Thus, it is possible to clearly discern which features segregate the analyzed samples into distinguished classes and which are the features that make given classes to be closer to each other.

However, when dealing with malware, we may not fulfill all of the aforementioned requirements. This happens due to the complexity of malware behavior and we return to this issue.

2.6 Malware Taxonomies

There are several attempts reported in the literature concerning the creation of malware taxonomies. However, those taxonomies either address only one type of the existing malware set, e.g., a taxonomy specific for worms, or are closely tied to the standard classes and the current naming schemes. Although these efforts are valid and useful, we believe that there is still the lack of a generic and more comprehensible taxonomy obeying the requirements presented in Section 2.5. In this section, we present important research works that are focused on malware taxonomies and whose contributions led to the development of our proposal.

2.6.1 Viruses, Worms, Trojans

Filiol discusses the misuse of the term computer virus and the user's incomplete knowledge in the subject [54]. He defines malware as "computer infection programs" and proposes a taxonomy that divides them into two classes, **simple**, which encompasses logical bombs and Trojans, and **self-reproducing**, which embraces viruses and worms. Filiol shows that worms are particular viruses from an algorithmic point of view and that it is difficult to classify malware using the standard definitions (virus, worm, Trojan) due to the overlap of behaviors that characterize each class.

Weaver et al. state that "*to understand the worm threat, it is necessary to understand the various types of worms, payloads, and attackers*" [161]. They propose a taxonomy of computer worms whose attributes are the **target discovery technique**, the **propagation mechanism**, the **activation method**, the **payloads** and the **attackers' motivation**. As mentioned in their paper, the proposed taxonomy is incomplete, as new attacks and malware variants arise constantly. Table 2.2 summarizes this taxonomy of worms, which uses the *carrier*, the *activation* and the *payload* to describe a certain worm, as they are independent attributes.

Karresand proposes a taxonomy to separate viruses, worms and Trojans [79]. He discusses the fact that the malware developers started to combine different functions into a single sample, making harder the naming task. Also, he shows examples of malware that are categorized into multiple or different classes by antivirus vendors. He proposes TEBIT, a taxonomy of software weapons with 15 independent categories and one to four possible values for each category:

- **Type**: atomic or combined.
- **Violates**: confidentiality, integrity/parasitic, integrity/non-parasitic or availability.
- **Duration of effect**: temporary or permanent.
- **Targeting**: manual or autonomous.
- **Attack**: immediate or conditional.
- **Functional area**: local or remote.
- **Affected data**: stationary or in transfer.

Table 2.2: The taxonomy of computer worms proposed by Weaver et al., 2003.

Attribute	**Possible values**
Target Discovery	Scanning Pre-generated target lists Externally generated target lists Internal target lists Passive
Propagation Carrier	Self-carried Second channel Embedded
Activation	Human activation Human activity-based activation Scheduled process activation Self activation
Payloads	None/nonfunctional Internet remote control Spam-relays HTML-proxies Internet DoS Data collection Access for sale Data damage Physical-world remote control Physical-world DoS Physical-world reconnaissance Physical-world damage Worm maintenance
Motivations	Experimental curiosity Pride and power Commercial advantage Extortion and criminal gain Random protest Political protest Terrorism Cyber warfare

- **Used vulnerability**: CVE/CAN, other or none.
- **Topology of source**: single or distributed.
- **Target of attack**: single or multiple.
- **Platform dependency**: dependent or independent.
- **Signature of replicated code**: monomorphic, polymorphic or not replicating.
- **Signature of attack**: monomorphic or polymorphic.

- **Signature when passive**: visible or stealthy.

- **Signature when active**: visible or stealthy.

Although interesting in the sense of stressing once more the current malware naming schemes problems, TEBIT suffers from two major issues: it does not explain clearly the results of an attack to the victim's system and its categories heavily rely on technical descriptions provided by antivirus vendors (the same ones that still apply the criticized naming scheme). Moreover, TEBIT's categorization results show that it uses very few categories to separate viruses, worms and trojans, whereas the values of the remaining ones are unclear. This led to the conclusion that almost any malware is a Trojan horse; thus, there is no need for a classification process. Also, the few categories that clearly distinguish the classes fit the standard definitions, i.e. a virus is parasitic and performs local attacks, a worm is non-parasitic and propagates remotely.

2.6.2 Botnets

Bots are very remarkable species of malware as they present features that could include them in a large variety of classes. Bots can spread like worms, open ports or setup servers to act as backdoors, use rootkits to hide their presence, download other pieces of malware or components for self-updating, monitor keystrokes to steal information as keyloggers do and so on.

To evade detection and to accomplish their malicious intents, botnets are generated byusing different structures. Cooke et al. [38] identify three topologies related to the communication methods used by a botnet's infected machines—zombies or bots—and its controller—mothership, botmaster, botherder or C&C server: **centralized**, in which there is a central location to allow the exchange of messages among clients; **peer-to-peer** (P2P), which are structurally more complex and harder to take down due to the inexistence of a central point of failure; **random**, in which each bot knows only one other bot, passing encrypted messages after randomly scanning the Internet to find it.

Dagon et al. propose a taxonomy of botnet topologies to discuss their organization and utility for malicious activities [42]. They propose the effectiveness, the efficiency and the robustness as discriminators (or attributes) to measure botnets. The authors consider different types of graphs from complex networks to model the possible topologies of their taxonomy:

- **Erdös-Rényi random graph models**: denote randomly organized structures.

- **Watts-Strogatz small world models**: there are regional networks grouping local connections in a ring whose distant nodes are accessed through shortcuts, e.g., victim's lists.

- **Barbási-Albert scale free models**: usually organized in a hub architecture with a small amount of highly connected central nodes and a larger amount of less connected leaf nodes;

- **P2P models**: can be organized in structured or unstructured topologies.

Another way to develop a taxonomy of botnets is to consider their behavior once a bot compromises a victim's system. Rajab et al. [1] discuss the execution features that they have observed in IRC-based botclients, which they divided into the following classes: **AV/FW killer**, meaning that the bot tried to disable possible defense mechanisms running on the target; **Identd server**, when a bot run `identd` so that the IRC server can verify that this bot is allowed to join a specific channel; **System security monitor**, which indicates a bot's self-defense behavior to "secure" the compromised target against other malware attacks; **Registry monitor**, denoting the behavior of a bot that verifies for system's registry changes that may point to disabling attempts.

2.6.3 Spyware

Spyware is a malware class that aims to monitor the behavior of users and steal their data in a covert manner [48]. Saroiu et al. [128] refine the class into the following subclasses[1]:

- **Cookies and Web bugs**: can track the Web behavior of users in a passive way.
- **Browser hijackers**: attempt to change the browser settings usually by installing browser extensions to steal user data.
- **Keyloggers**: capture sensitive information, logs of visited sites, instant messaging sessions and so on.
- **Tracks**: the recorded information about the actions of a user that can be used by a malicious program.
- **Malware**: any malicious software.
- **Spybots**: a kind of spyware that monitors the user's behavior and transmit the collected information to a third party.
- **Adware**: software that displays advertisements based on the captured behavior of a monitored user.

We notice that the above classes can be somewhat confusing if we try to apply them to actual samples, i.e. most malware whose behavior is different from "spying and stealing" fit into the **Malware** class. Moreover, there are overlaps among some descriptions (mainly those related to user data monitoring). It is also worth to notice that all of the classes target the user's privacy.

Some harmful effects attributed to spyware are presented by Boldt et al. [23]. The authors mention four main categories of risks posed by spyware: **consumption of system capacity** (processing) and **consumption of bandwidth** can cause availability problems; **Security issues** and **privacy issues**, such as user's information (credentials, e-mail addresses) leakage through covert communications violate the principle of confidentiality.

[1] We can notice that *Tracks* and *Malware* are more a terminology than a spyware class.

2.6.4 Rootkits

Rootkits are often seen as stealth malware, as they aim to operate in an unsuspicious and hidden way. Rutkowska proposes a taxonomy to classify stealthy malware samples according to their interactions with the operating system [127]. For this purpose, she redefines the term that describes the general concept of malware to a new one that considers malware as a piece of code that modifies the behavior of the operating system kernel or some security application in an non-consenting, undetectable and undocumented way.

To embrace even the "ordinary" malware, i.e. those that are not stealthy in the sense she defined, her taxonomy describes four classes of malware, explained below:

- **Type 0 malware**: does not compromise the operating system nor the behavior of other applications of processes using undocumented methods. A type 0 malware can still compromise the integrity and the availability of user data (by modifying or deleting it) or the user's information confidentiality (by stealing sensitive documents or credentials and evading them through network connections), but this is done using valid APIs. This type of malware embraces the traditional classes already described in the previous sections.

- **Type I malware**: modifies operating system's constant resources, such as executable files and in-memory code sections of running processes or the kernel. This type describes rootkits that usually "trojanize" resources to turn communications stealthy or to hide processes, files and directories, for example.

- **Type II malware**: operates on dynamic resources that are designed to be modified, e.g., configuration files, registry keys and data sections of processes or the operating system kernel. Type II malware are those rootkits that apply dynamic hooking in kernel structures,

- **Type III malware**: a dangerous malware that can take the complete control of the operating systems without modifying the memory or any visible registers. Those rootkits leverage hardware virtualization techniques to run with greater privileges than the operating system.

2.6.5 Malware Taxonomies Panorama

As we can observe in the previous subsections, each of the presented taxonomies tries to classify specific malware types. Such classification schemes may be precise to describe those known malware types, as some of them delve into details such as the attacker's motivation, the vulnerability used during the infection, the propagation model and so on. However, this kind of approach may not be useful to users or to automated incident response procedures, since the analyst has to potentially analyze the sample manually to gather all the taxonomy's required information. Due to this fact, we propose a more general approach on the next chapter. Table 2.3 provides an overview of the taxonomies discussed in this chapter.

Table 2.3: Summary of the discussed malware taxonomies.

Malware Type	**Approach**	**Section**
Viruses, Worms and Trojans	Filiol [54] divides computer infection programs into *simple*—logical bombs, Trojans—and *self-reproducing*—viruses and worms—mentioning the malware behaviors potential overlap. Karresand et al. [79] also discusses the combination of different behaviors and proposes a taxonomy that is heavily dependent on AV descriptions. Weaver et al. [161] proposes a worms taxonomy that considers the attacker's motivation.	Section 2.6.1
Botnets	Cooke et al. [38] adresses the communication structures used between bots and their motherships. Dagon et al. [42] uses graphs to model botnets topologic organizations. Rajab et al. [1] considers the behaviors performed by bots during the infection procedure.	Section 2.6.2
Spyware	Saroiu et al. [128] defines (sometimes overlapping) subclasses to spyware that are based on their expected behavior. Boldt et al. [23] classifies spyware according to the risk they pose to the security principles of availability and confidentiality.	Section 2.6.3
Rootkits	Rutkowska [127] proposes a taxonomy based on the modifications that a malware sample performs on the infected operating system. It considers mainly stealth malware and the violation they can cause to the integrity of the target.	Section 2.6.4

2.7 Concluding Remarks

The lack of a consistent taxonomy that embraces most of the malware species is a clear conclusion of this chapter and the main motivation for this thesis. Furthermore, a more useful taxonomy to classify malware should include the ability to automate the classification process so as to help the security analyst responsible for handling a malware infection incident. Due to the level of detail and dependence upon external entities or specific knowledge about the inner workings and/or organization of some types of malware, we reasoned that automation can be hard for most of the presented taxonomies. We also presented in this chapter a brief history of malware, from just intelectual concepts to full-blown weapons of an ongoing cyberwar. In addition, we discussed the unaddressed issues of current malware naming schemes, introduced basic concepts on the field and finally summarized recent research available on malware taxonomies.

Chapter 3

A Behavior-Centric Malware Taxonomy

3.1 Introduction

Available malware taxonomies try to explain the behavior of a certain malware class by means of the attacks they launch, the kind of damage they cause or the way they are structurally organized to perform malicious activities. However, they are tied to a naming scheme that considers “viruses”, “worms”, “Trojans” and other identifiers as classes and attributes to them a static behavior, such as “appends itself to a file” or “autonomously propagates”.

In this chapter, we define the different aspects of “behavior” that a program can present, briefly discuss other research about malware behavior and, finally, propose a novel taxonomic scheme based solely on the malicious behavior of a program as it compromises a victim’s system. This way, we intend to effectively help incident response efforts as well as to provide a better understanding of a malware sample under analysis, fulfilling important requirements of a good taxonomic scheme.

3.2 Behavioral Aspects of Malware

Malicious programs behave most of the time similarly to benign programs. Therefore, to “analyze” a piece of software, we need to pinpoint aspects of its behavior that serve the purpose of characterizing malignity in it. We mention previous works on malware behavioral aspects and redefine “malware behavior” in the scope of this thesis.

3.2.1 Recent Work on Malware Behavior

Filiol et al. propose a theoretical model to perform behavior-based detection of infectious actions [56]. Their work presents a strong mathematical basis to define different types of behavior and is an extension to a previous work that was capable only of handling sequences of bytes [55]. The limiting factor is that their approach requires the malware’s source code, which is sometimes difficult to obtain.

Jacob et al. describe a malicious behavior model that is based on attribute grammars [74]. They propose an abstraction layer to bridge the semantic gap between the behavior-based detection of malware and subtleties of platforms and systems. They trace a malware sample behavior through its execution in a virtualized enviroment in order to monitor the sample's system calls. These system calls are then translated to their malicious behavior language. They examine and formally define four types of malware behavior.

Martignoni et al. propose to bridge the semantic gap using behavioral graphs that are manually built to correlate common actions found in malicious bots, such as e-mail sending and data leaking [97]. They evaluate seven kinds of behaviors related to bots; thus, their coverage is mostly based on network actions.

Bayer et al. provide a view on different behaviors presented by almost one million malware samples that were analyzed by Anubis over 2007 and 2008 [87]. They produced statistics and analyzed trends, showing the percentage of observed samples that performed a variety of actions, from a simple file creation to the installation of a Windows kernel driver. We took some of their observed behavior to compose our suspicious activity definitions but, instead of analyzing the overall scene, we tried to delve into behaviors we believe to pose risk to target systems.

3.3 Definitions

General Behavior. We consider the general behavior of a program as the set of actions performed during its execution by an operating system. We define "action" based on some attributes: source of action (usually the malware process), operation (create, delete, write, terminate and so on), object (file, process, network, registry, mutex, memory), target of action (the name or path to the object that is subject to one of the available operations), and arguments (additional parameters). Thus, an action "α_i" is a tuple composed by the values of the aforementioned attributes and, therefore, we define the general behavior as follows:

Definition 1 *Let B be the general behavior of a malware sample M_k, composed by the set of N actions $\alpha_1, \alpha_2, ..., \alpha_N$ performed during its execution, so that $B(M_k) = \{\alpha_1, \alpha_2, ..., \alpha_N\}$.*

The set of actions that compose a behavior can be divided into groups according to their nature: if an action interferes with the environment, i.e. changes the state of the system, it is part of an **active** subset of the behavior. This is the case of actions that involve a file write, delete or creation, for instance. Otherwise, the action is **passive**, meaning that it gathered a piece of information without modifying anything, for example, read, open or query something.

However, there is a subset of the general behavior that is **neutral**, i.e. the actions can be either active or passive, but they do not lead to a malign outcome. The neutral behavior contains common actions that are performed during a normal execution of any program, such as to load standard system libraries, to read or to configure registry keys and to create temporary files.

Suspicious Behavior. When a malicious program is executed, each of its actions can be considered suspicious. These actions constitute a suspicious behavior that, when analyzed, may reveal important details related to the attack. For instance, a malware sample that downloads another piece of malicious code and use it to spread itself has to perform a network connection, to write the file containing the malicious code in the compromised system and to launch the process of the downloaded file that will handle the spreading process.

Therefore, we are interested only in actions that modify the state of the compromised system (the active subset of the general behavior) at the same time that we want to ignore the actions that are considered normal to a program's execution (the neutral behavior). Hence, we define the suspicious behavior as follows:

Definition 2 *Let M_k be a malware sample whose general behavior $B(M_k)$ is divided into the active behavior $B_A(M_k)$ and the passive behavior $B_P(M_k)$. Then, $B(M_k) = B_A(M_k) \cup B_P(M_k)$. Let $B_N(M_k)$ be the malware sample's neutral behavior so that $B_N(M_k) \in B_A(M_k) \cup B_P(M_k)$. Thus, the suspicious behavior $B_S(M_k)$ is equal to $B_A(M_k) - B_N(M_k)$.*

The collection of suspicious behaviors and a few exceptions pertaining to passive behaviors allows us to identify the potentially dangerous behaviors discussed next.

3.3.1 Potentially Malicious Behaviors

During execution, a program interacts actively and passively with the operating system. Thus, benign software presents active behavior such as creating new registries, writing values to registry keys, creating other processes, accessing the network to send debug information or to search for updates, downloading and writing new files etc., which modify the system state in an authorized manner.

Therefore, as any piece of software does, malware interact with the operating system in the same way. However, malware interactions cause undesired or unauthorized changes in the operating system settings. The discovery of suspicious behaviors to categorize malware is a difficult task, as a legitimate or benign application may also present one or more of these behaviors, as mentioned above.

Despite this fact, as it is not expected that a malware sample performs only one of the (network and operating system) listed behaviors, we also consider in the categorization process those that do not pose a severe risk to the compromised system. Hence, less risky behaviors are included as we believe that they provide a better understanding about an analyzed malware sample, thus allowing better decisions related to incident responses or counter-measures procedures.

To the extent of this thesis, we gathered 24 suspicious behaviors that are listed below. They are initially grouped based on their venue (network or operating system features). On the one hand, we have tried to be as complete as our malware collection allowed us to be. On the other hand, the taxonomy proposed in this thesis has as a main objective to be extensible, so it should be easy to add new behaviors to our knowledge base.

Network Patterns

Network activity patterns that may point to suspicious behaviors are:

Information Stealing. Information related to the operating system or the user can be stolen through the network, such as the hostname, hardware data, network interface data, OS version and credentials. An adversary may use this information to choose targets for an attack, or to map her compromised machines (e.g., zombie computers that are part of a botnet). In a directed attack, sensitive documents may be stolen and transferred to an FTP server, for instance. Also, information can be stolen through a POST (HTTP method) performed on a compromised server, a FTP transfer, an SQL update query to a remote database, an IRC (Instant Relay Chat) communication or an e-mail message sent through an open SMTP server. We choose the following information as interesting ones to define this behavior:

1. Stealing of system/user data.

2. Stealing of user credentials or financial information.

Scanning. Malicious programs often perform scans to search for local or remote targets. These scans aim to discover information about the targeted systems to allow malware to disseminate, e.g., operating system and applications version, opened network ports, vulnerable services. Worm-like malware need to perform scans over the network to find possible targets for spreading. This involves the search for known vulnerable services or unprotected/open network applications. Apart from spreading, a malware sample may also perform scans to map a network topology or to find trampoline systems that could be used to launch attacks anonymously.

Downloading. Some types of malware consist of several pieces that execute specialized tasks. Thus, the first piece—the downloader—is responsible for downloading other components, such as libraries, configuration files, drivers or infected executable files. This compartmentalization is also used by malware developers to try to avoid antiviruses or other security mechanisms. This activity can also indicate a drive-by download, which is a download commonly performed during a user's Web browsing without his/her knowledge. We will consider executable files or libraries to specialize this behavior into:

1. Download of known malware.

2. Download of unknown file.

E-mail sending. A malicious program can communicate with its owner through e-mail to announce the success of an attack or to send out sensitive data from the compromised machine (see *evasion of information*). Also, a compromised machine can be used as an unsolicited e-mail server, sending thousands of spam on behalf of an attacker that is being paid for the service. The victim's machine is "rented" and acts as a provider of spam or phishing, aiming to distribute commercial messages or even malicious links or attached infected files [28]. Thus, it is interesting to "flag" when a program sends an e-mail message, as it may happen without the user's consent.

IRC/IM connection. If an attacked system becomes part of a botnet, it needs to "phone home", i.e. to contact a C&C[1] server to receive commands, updates etc. Botclients commonly connect to an Instant Relay Chat (IRC) server that acts as a C&C. To define this behavior, we consider connections to IRC or Instant Messaging (IM) servers as well as IRC commands passing as clear text within the network traffic:

1. Connection to known IRC/IM port.

2. IRC/IM unencrypted commands.

OS Patterns

Activity patterns related to operating system settings, services or applications that may point to suspicious behaviors are:

Shutdown defense mechanisms. Malware authors usually try to identify and disable security-related processes in order to evade detection when compromising a victim's system. This activity can be accomplished by turning off the system firewall or known antivirus engines, through the termination of their processes and/or removal of the associated registry keys. In addition, there are registry keys that are critical to the normal operation of a system, e.g., the one that allows initialization in secure mode. The removal of this kind of key can cause instability in the system and inconvenient obstacles during a disinfection procedure. We consider the following as suspicious activities when referring to this behavior:

1. Disable system firewall.

2. Disable system update notification.

3. Terminate known antivirus engine.

4. Removal of critical registry key.

Add object in sensitive area. This behavior is tied to the objects that a malware sample may create on a system to perform malicious tasks. These objects can be new libraries (DLLs in our case) or executable files, overwritten library/executable files so as to subvert their use, mutex elements to lock resources and avoid accesses (mutex names may be used either as part of detection signatures or by malware samples to avoid the reinfection of a system already compromised by a "member" from the same family), protecting malicious resources from termination or crashing. Thus, we identify the following suspicious activities:

1. Creation of new DLL or EXE.

2. Modification of existing (system) DLL or EXE.

3. Creation of unusual mutex.

[1] Command and Control server that manages the bots that belong to a botnet.

Subversion of Internet browsing. The subversion of Internet browsing is very dangerous as users are, in most cases, unaware of the changes that occurred in their systems. A Trojan-like malware sample can modify the network name resolution file to forward users to a compromised server and lure them into supplying their data. These can be credentials (e-mail, online banking, Web applications such as Facebook and Twitter) or financial information (credit card numbers). This kind of modification tricks users to access fake online banking sites as a direct cause of malware known as “bankers”. Another activity that is similar to the aforementioned one occurs when a malware sample loads a configuration file in the browser’s memory (when it is running) that changes the proxy on the fly, resulting in an automatic redirection of the user to a malicious site. Malware usually do it using PAC (proxy auto-config) files, which are effective on different operating systems and browsers. Moreover, it is possible to silently load a plugin such as a BHO (browser helper object) that modifies the browser behavior. We address three ways to accomplish this:

1. Modification of the name resolution file.
2. Modification of the browser proxy through PAC files.
3. Modification of the browser behavior via BHO loading.

Removal of evidence. Some malware disguise themselves as system processes to deceive security mechanisms or forensic analysis: they can “drop” a file that was embedded in a packed way inside their main file or download the actual malicious program from the Internet. In some cases, these droppers/downloaders remove the evidence of compromise, deleting the installation files after the attack. In addition, a malware sample that is able to identify it is under analysis may also remove itself from the system.

Driver loading. Drivers are kernel modules that access the most privileged level of a system. A driver makes the interface between the operating system and the hardware, such as network interfaces, graphic cards and other devices. However, drivers are also used by rootkits, a kernel-level kind of malware that can hide their processes, files and network connections in order to remain undetected.

Process hijacking. To disguise its presence and avoid monitoring, malware samples can write into the memory of another process (a benign one) and take control of it. Thus, the “hijacked” process can create a thread and perform malicious actions even if the original malware process is terminated.

Persistence. Once a system is infected, the motivation behind the malware sample may lead to the need of “survivability”, i.e. this sample should be able to survive a reboot and execute on a permanent basis. There are many reasons to this type of behavior, such as to keep a network port in an open state to allow the communication with an attacker (working as a backdoor), to perform a daily routine (collect files, credentials, logs), to connect to a remote system so as to receive commands (botnets) or even to wait for a specific date or event to occur.

Suspicious-Passive Patterns

Passive behaviors, which can be part of previous step leading to a malicious activity (e.g., some information related to the target that may cause a change of the expected behavior or a further exposure), should also be considered:

Language checks. Malware that have well-defined targets usually verify the system's installed language. This is a behavior common to Internet banking malware, as they are specially crafted to attack a specific type of user or application. In this case, the language checking serves two main purposes: to identify if the system has the appropriate language to run (if not, exit) and to avoid publicly available malware analysis systems (e.g., a malware sample targeting Brazilian banks that checks for the Portuguese language will terminate its execution if analyzed in an English or German analysis system).

System/user information reading. To perform the evasion of information related to the system or its users, malware samples need to obtain that information, usually from the registry. This type of information may also serve as management data to an attacker, i.e. a table of compromised machines and their peculiarities, such as username, hard disk identifier, operating system version, patching level etc.

Although finding these patterns does not indicate malignity, their combination with another behavioral feature can bring light to an analysis. For instance, if there is a language check in a behavioral trace that ends abruptly or that terminates without presenting other behaviors, it may be possible that a malware sample is targeted to a language different from the ones installed on our system.

3.4 The Proposed Taxonomy

Based on interesting attributes from other malware taxonomies (Section 2.6), the suspicious behaviors extracted from our malware collection and from the recent literature (Section 3.2.1), and aiming to meet the taxonomy requirements that make sense regarding our scope, we propose a novel taxonomic scheme with three dimensions, which is detailed below. The dimensions of the proposed taxonomy are:

- **Class**, an attribute that describes the type of behavior observed in a malware sample (Section 3.4.1).
- **Suspicious activity**, which specifies the potentially malicious (or at least dangerous) activity performed by a malware sample and that caused its assignment to a certain class (Section 3.4.2).
- **Violated security principle**, to make it better and simpler to understand the damage and risks that a malware sample poses to its victim.

The resulting scheme assigns a label to a malware sample that allows us to map it back to the taxonomic descriptions. Thus, a sample may be in several classes if it presents multiple behaviors.

3.4.1 Malware Classes

We mentioned in the previous chapter that it is very difficult to stick to some requirements for a taxonomy regarding malware. The main issue in our case is to meet the **mutual exclusivity** when assigning a class to a sample. Although a malicious program might exhibit only one behavior that predominantly characterizes it as pertaining to a certain class, there may be other behaviors that allow us to assign the sample to other classes also. Thus, we propose in this section a non exhaustive set of possible classes and state that a malware sample may be tied to more than one of them at the same time.

Stealer. A stealer is a piece of software that captures data (sensitive or not) from the compromised machine, potentially causing information exposure or identity theft. Stealers can disclosure confidential information from a system as well as from users, credentials (usernames, passwords, credit cards, Internet banking account and login information etc.), documents, keys pressed, opened windows and mouse clicks. Malware known as keyloggers, spyware, bankers and some botclients fit in this class.

Evader. Evasive malware samples try to bypass system security features or computer forensics procedures through the use of techniques that range from removing their own file after infections to stopping antiviruses engines and firewalls executing on the compromised system. There are malware samples that use packers to obfuscate themselves or leverage anti-analysis techniques to identify if they are being executed under a debugger, an emulated or a virtualized environment. Additionally, modern malware may launch several shadow processes to distribute their malicious actions [95] while evading detection mechanisms. Evasive malware can also be seen as deceptive programs. A deceptive program lure users by making them believe that it has a legitimate functionality or by trying to disguise itself as a system "resource". For instance, recently a malware started to execute as an antivirus engine to make users to provide their credit card numbers (so that users buy the fake antivirus to clean their infected machines) [144]. Other type of malware can disguise itself in the form of Internet bank protection mechanisms updates, tricking users to input their one-time password (OTP) tables and all online banking credentials. More common malware samples install and launch themselves using the same name (sometimes with minor modifications or typos) of known system's processes, or inject themselves to another processes. This type of malware may also need to operate stealthily. To operate in a stealthy manner, a program must run with administrative privileges or even greater, as in case of hardware-assisted virtual machine (HVM) rootkits [43]. Moreover, to get access to kernel resources, such as device drivers, userland[2] malware need to escalate privileges to assure the total control of the compromised system. Rootkits are a common type of privileged malware,

[2] Userland malware executes with limited access privileges, such as an ordinary user, i.e. non administrator.

as they usually load new drivers, modify system drivers and applications or change the system's flow of data while trying to remain unnoticed in the system. This class enables impersonation and process hijacking attacks.

Disrupter. Malicious programs can be used in directed attacks to disrupt computational resources to make them unavailable. Disrupters may be as simple as flooders that consume bandwidth or malware that self replicate until the exhaustion of processing power or memory, or they may launch massive and distributed denial of service attacks against major Internet sites with several motivations, e.g., protesting, politics, competition, etc. Even worse, this type of malware could inflict physical damages to infrastructures, e.g., Iran nuclear plant [16], or attempt against human lives, e.g., the shutdown of a hospital near Atlanta, Georgia, US [169]. However, their detection or their behavioral profiling can be difficult, as they are usually stealthy and/or dependent on specific frameworks, programs or devices to run properly. On the other hand, malware samples can connect to remote systems to wait for commands (e.g., scan a network, disseminate, attack a system). Although this type of behavior is not necessarily malicious while the sample is waiting the command, the result after an order is usually a scan, an exploit attempt, a flooding, a spam sending and so on. This way, we conclude that this type of malware can receive orders or act autonomously, disrupting their target environment functioning locally and/or remotely.

Modifier. To perform its malicious activity, a program may need to modify objects of the compromised system. These changes include the overwriting of processes, the removal of files, the modification of system settings (e.g., proxy, name resolution files, configuration directives to execute after a reboot), the substitution of a system library or binary, etc. Botclients, bankers, downloaders, droppers and Trojans pertain to the modifier class.

3.4.2 Observed Suspicious Behaviors and Related Classes

The set of behaviors described in Section 3.3.1 represents some of the possible suspicious activities that may compose a malware class. For instance, a sample whose type is "stealer" can accomplish its information stealing task in several manners, such as sending system data through HTTP methods, credentials through e-mail and so on. Table 3.1 presents the correspondence between previously defined suspicious behaviors and their related classes. Table 3.1 also contains a set of labels to identify each behavior within the malware class in a shorter manner. This led us to produce a reduced naming scheme that is based on the combination of the classes for which a malware sample pertains ($C = \{D, E, M, S\}$), associated with the specific subset of behaviors (B_C ={[VS, ES, IP, IC], [RE, RR, TA, TF, TU, LC], [NB, CB, UM, HC, PL, BI, Pe, DK, DU, DL], [IS, CS, IR, PH]}). Therefore, a malware sample can be classified based on the performed infection behavior and its label will be "assembled" according to it. Thus, the format of a general label is:

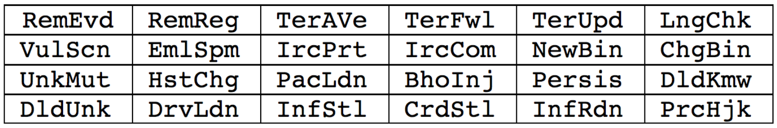
$$C1_{B_{C1}}.C2_{B_{C2}}.C3_{B_{C3}}.C4_{B_{C4}}$$

This way, a credential stealer that downloads an unknown file and injects a BHO to subvert the victim's browser would be represented as "$M_{[DU,BI]}.S_{[CS]}$".

Table 3.1: Proposed malware classes, suspicious behaviors and associated labels.

Class	**Behavior**	**Label**
Evader	Removal of Evidence	RemEvd [RE]
	Removal of Registries	RemReg [RR]
	AV Engine Termination	TerAVe [TA]
	Firewall Termination	TerFwl [TF]
	Notification of Updates Termination	TerUpd [TU]
	Language Checking	LngChk (Suspicious) [LC]
Disrupter	Scanning of Known-Vulnerable Service	VulScn [VS]
	E-mail Sending (Spam)	EmlSpm [ES]
	IRC/IM Known Port Connection	IrcPrt [IP]
	IRC/IM Unencrypted Commands	IrcCom [IC]
Modifier	Creation of New Binary	NewBin [NB]
	Modification of Existing System Binary	ChgBin [CB]
	Creation of Unusual Mutex	UnkMut [UM]
	Modification of the Name Resolution File	HstChg [HC]
	Modification of the Browser Proxy Settings	PacLdn [PL]
	Modification of the Browser Behavior	BhoInj [BI]
	Persistence	Persis [Pe]
	Download of Known Malware	DldKmw [DK]
	Download of Unknown File	DldUnk [DU]
	Driver Loading	DrvLdn [DL]
Stealer	Stealing of System/User Data	InfStl [IS]
	Stealing of Credentials or Financial Data	CrdStl [CS]
	System/user Information Reading	InfRdn (Suspicious) [IR]
	Process Hijacking	PrcHjk [PH]

We also propose a depictive matrix (to be included in a dynamic analysis report) that is based on our defined behaviors. This matrix is extensible, since new lines and/or columns can be easily inserted to reflect future additions to our taxonomy, and it aims to provide a quick guide to the report's reader. This matrix also allows for the visual identification of a sample as being similar (or not) to another previously analyzed sample in a fast and intuitive way. Figure 3.1 shows the overall current matrix disposition whereas Figure 3.2 shows the matrix for the previously mentioned example of malware sample ($M_{[DU,BI]}.S_{[CS]}$).



RemEvd	RemReg	TerAVe	TerFwl	TerUpd	LngChk
VulScn	EmlSpm	IrcPrt	IrcCom	NewBin	ChgBin
UnkMut	HstChg	PacLdn	BhoInj	Persis	DldKmw
DldUnk	DrvLdn	InfStl	CrdStl	InfRdn	PrcHjk

Figure 3.1: Behavioral matrix based on our taxonomy.

			BhoInj		
DldUnk			CrdStl		

Figure 3.2: Behavioral matrix for $M_{[DU,BI]}.S_{[CS]}$.

Furthermore, a behavior exhibited by a malware class may violate one or more security principles that affect users and the adequate functioning of the system. Quoting Matt Bishop [22], "*computer security rests on confidentiality, integrity, and availability.*" Thus, to the extent of this work, we consider these security principles when regarding the damage that a certain malicious behavior can cause. Briefly, they can be described as follows:

- **Integrity**: refers to the "credibility" of a subject content, i.e. the assurance that this subject—a file, an information, a piece of data—is not changed in an improper or unauthorized manner.

- **Confidentiality**: refers to the "need to know" principle, meaning that only those that allowed to do so should access an information or resource. The principle of confidentiality serves to the purpose of keeping the privacy and/or secrecy of a subject.

- **Availability**: refers to the possibility of access for use at any time that a resource or information is required, i.e. the subject must be available, providing its intended service.

To illustrate our proposed behavioral-centric taxonomy, we depict a mapping among the security principles that can be violated and the malware classes and their related behaviors in Figure 3.3.

3.5 Concluding Remarks

In this chapter we discussed malware behavior. We defined types of behaviors, focusing our research mainly on the active set of behaviors, i.e. activities that can cause changes to a compromised system. Moreover, we described some suspicious and dangerous activities that may be performed during the execution of a program. These activities are commonly observed in malware infections; thus, any sign of them should be observed carefully. Furthermore, we proposed a new taxonomic scheme that separates observed behaviors in classes (disruptor, evader, modifier, stealer), whose main goal is to be clear, understandable, easy to maintain and to update, and to overcome some of the inconsistencies seen on currently available naming schemes. Although the proposed taxonomy is not complete, as it is based on the authors observations about malware behavior along the last decade and their expertise related to malware analysis, it reflects most of the common malicious behaviors seen in the wild. Therefore, this taxonomy can be used for malware classification, detection and incident response procedures, generalizing the potentially dangerous behaviors that a malware sample can present, while at the same time clarifying the impact of the infection without cryptic naming schemes.

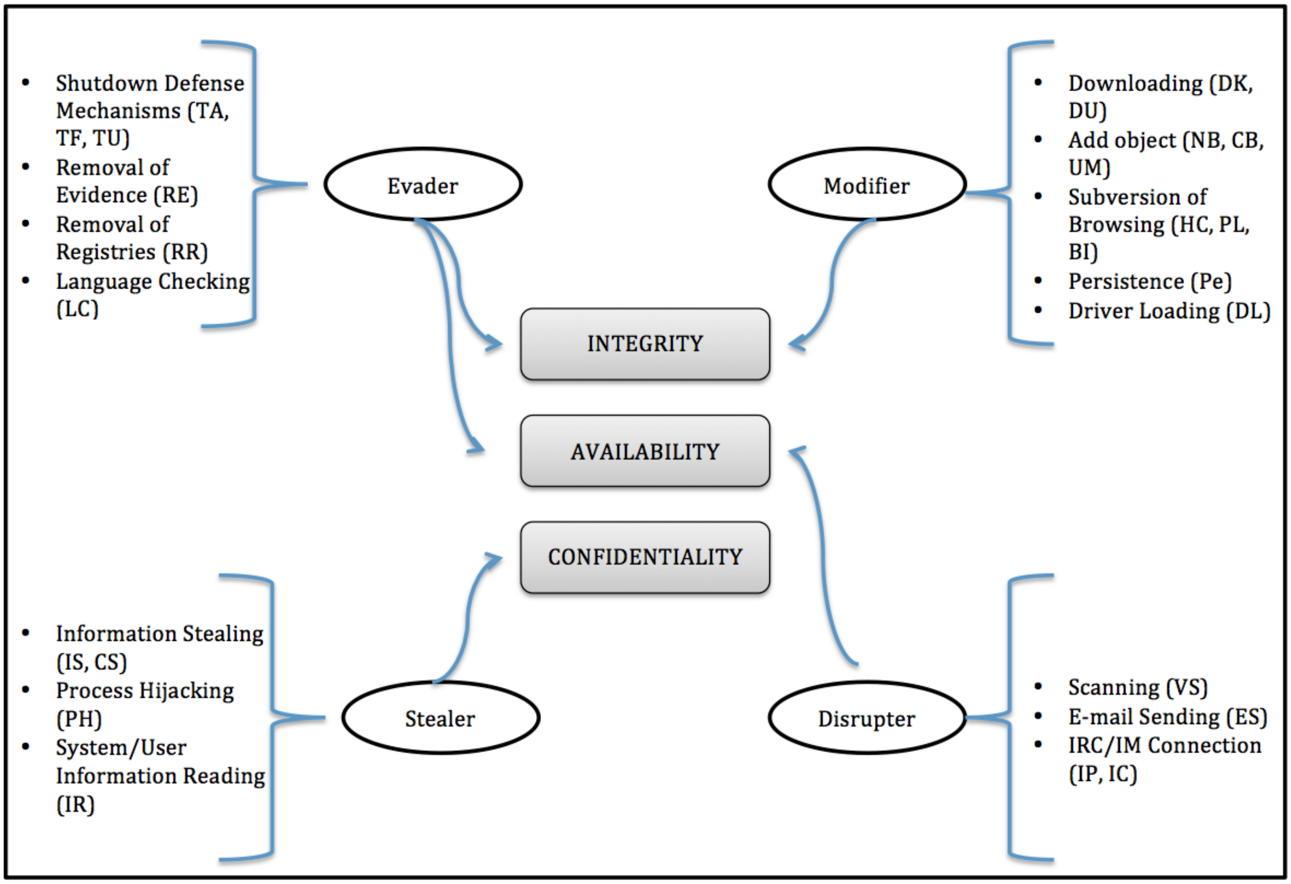


Figure 3.3: Mapping sets of behaviors to their classes and the security principles they violate.

Chapter 4

Evaluation of the Taxonomy

4.1 Introduction

In this chapter, we aim to evaluate and validate the taxonomy by collecting and dynamically analyzing malware samples in order to generate behavioral profiles for each one of them. These behavioral profiles will then be used to allow for the assignment of a sample into one or more of the taxonomic classes, consequently describing the malicious behaviors that it exhibits. To this end, we show the architecture proposed and developed that produces the behavioral profiles and analyze a large and diverse set of actual malware samples, providing a view on their malicious behaviors based on this taxonomic scheme and a discussion about the results.

4.2 Behavioral Profiling

To extract information about the behavior of a malware sample, we need to execute it in a controlled environment and observe its actions. We gather the behavioral profile of malware according to the defined set of behaviors and activities defined in Chapter 3.

Although the focus of this thesis relies on extracting malware behavior by running the executable, we prototyped a framework that is also able to analyze Web-related malware that perform attacks at the client-side of a system. To this end, we provide some additional information that is directly related to this thesis theme but that is not fully addressed on it to keep the scope tight. Therefore, we detail the framework from which we obtain the behavioral profiles, while at the same time emphasizing that not all Web malware detection techniques explained in the text were used in the present work.

Thus, we initially motivate the development of this framework by briefly discussing Web malware. Currently, the Web is the main vector to install malware in attacked systems. Web malware, i.e. malicious code inside Web pages, exploit the browser or some of its components and perform drive-by downloads that install malicious programs in the compromised system. They can also be used to steal personal cached information and even to control the browser.

Two methods are often used to make the victims' browser load malicious content, either by injecting malicious codes into benign pages and waiting for casual users to access it, or by sending phishing messages containing malicious files or links. The infection of benign sites is performed through the exploitation of some vulnerability in the Web application (server-side), followed by the insertion of the malicious content onto that server. The phishing messages are used to lure the victims to execute files or to access links (client-side) that lead them to the malicious content.

To develop and improve protection mechanisms deployed on the client-side, it is necessary to study and deeply understand these malicious pages and programs. There are several systems that perform this kind of analysis, but they are focused either on Web or operating system (OS) malware.

One of the major problems that makes the malware analysis process hard is the use of obfuscation techniques via packers. These packers change the code in a way that still keeps the main functionality, but turn the manual or static analysis into a very hard task. Moreover, some packers insert blocks of code inside the obfuscated file to verify whether it is being executed on an emulated or virtual environment, hiding its malicious behavior in such cases.

To analyze malware, we developed a framework that obtains URLs and files from a spam crawler and from malware collectors, and transparently performs their analysis. This framework is capable of analyzing both Web and OS-based malware and its modular design makes it possible to extend it to other file types or languages. We deployed a prototype and tested it against actual malware from our collected dataset, presenting results that show that our framework has some advantages over the existing systems that perform Web and OS-based malware analysis. The evaluation performed using Web malware shows that our detection rate is much better than that of other existing systems. Also, due to the way we capture the behavior of OS malware, we can deploy the monitoring system in emulated, virtual or real environments, providing an advantage over some of the existing malware analysis systems that are tied to specific virtualized [168], bare-metal [83] or emulated environments [86].

Before showing the proposed architecture, we present some related work related to Web and OS malware analysis.

4.2.1 Malware Analysis Systems

There are several analysis systems designed to monitor the behavior of Web or OS malware, as described in the following sections. However, each of them focuses solely on one of the mentioned malware types. Below, we present the main systems and techniques that are used to analyze malware, to produce informative reports about them and, in the case of Web malware analyzers, to tell whether the analyzed sample is malicious or benign.

OS Malware analysis

A malware behavior can be defined by the set of activities performed on the operating system. These activities include modifying files, writing registry values, establishing network connections, creating processes etc. Malware analysis is the procedure applied to a malicious program in order to extract features that can characterize it.

Malware analysis can be performed in a static way, i.e. without executing the sample, or dynamically, by monitoring its execution. The use of packers make the static analysis a very difficult and slow [103] process. Therefore, the most frequently used systems for malware analysis use the dynamic approach. Common techniques to dynamically extract malware behavior are: Virtual Machine Introspection (VMI), System Service Dispatch Table (SSDT) Hooking, and Application Programming Interface (API) Hooking.

In the case of VMI, a virtual environment is used to execute the malware and restore the system after the analysis. Monitoring is performed in an intermediary layer, called Virtual Machine Monitor (VMM), which is interposed between the virtual system and the real one. This approach allows for the extraction of low-level information, such as system calls and memory state. The down side is that this kind of analysis always needs a virtual environment. Moreover, some types of malware try to realize whether they are under analysis through the detection of a virtual environment. Thus, the malicious actions may not be executed [121] if the virtual environment is detected. VMI is used by the Anubis [72] system.

SSDT is a Windows kernel structure that contains the addresses of native functions. The SSDT hooking is performed at kernel level by a specially crafted driver that modifies some addresses of the SSDT to point to functions inside this driver. These functions can change the execution flow and the values returned to programs. This technique can be used in virtual, emulated or real environments as its flexibility is linked to the driver's mobility. However, there may be some issues related to rootkits' analysis, as they also operate at the kernel level and possess the same privileges of the monitoring driver. The framework presented in this text uses this technique to capture the malware behavior at the OS level.

Another technique to monitor malware execution behavior is the API hooking. It modifies the binary under analysis to force the execution of certain functions that are in the monitoring program, before calling the selected system APIs. On the one hand, this technique is deployed at a level that is closer to the analyzed sample, so it is possible to easily obtain higher-level information. On the other hand, this also makes it easy for a malware sample to detect the monitoring action through integrity checking. Furthermore, malware can issue system calls directly from memory-mapped addresses, evading this kind of monitoring. This approach is used by the CWSandbox [168].

Web malware analysis

Web malware analysis is usually performed through a component located at the operating system or at the browser. In both cases, the monitoring system verifies whether the analyzed Web page contains malicious codes or not and provides some information about the captured behavior. The three most used (and publicly available) systems are JSand, PhoneyC and Capture-HPC, which are described below.

JSand [40] is a low-interaction honeyclient that uses a browser emulator to obtain the behavior of the JavaScript code present in the Web page. Then, the system extracts some features from the obtained behavior and applies machine learning techniques to classify the analyzed page as benign, suspicious or malicious. The main problems related to this approach are its limitation to JavaScript-only analysis and the inability to detect attacks that steal information from the browser.

PhoneyC [106] is another low-interaction honeyclient that uses a browser emulator to process the analyzed Web page and is able to analyze JavaScript and VBScript codes. To detect exploits in those types of codes, it emulates certain vulnerable components. The limitations are the same as JSand's, except for the VBScript analysis.

Capture-HPC [132] is a high-interaction honeyclient that uses a full-featured browser and a kernel driver inside a virtual environment to extract the system calls performed by the browser as it accesses the analyzed page. Then, it performs a classification based on these system calls, in order to obtain a benign or malicious result. Regardless of the aforementioned systems, Capture-HPC can detect attacks independently of the script language used, but only those that generate anomalous system calls.

4.2.2 System Architecture

The architecture of our proposed framework can be seen in Figure 4.1, where the dark lines indicate the analysis from Web related files and URLs, and the dotted lines indicate the analysis of OS executable files. A spam crawler, malware collectors and manual input are the source of the samples, which are then forwarded to the *Selector* module. The *Selector* is responsible for the identification of the sample type and its sending to the appropriate module. Windows executable files—PE32 and DLL file formats—are sent to the OS module, whereas Web-related content, such as URLs, HTML and JavaScript files, are sent to the Web module. The OS module extracts the sample behavior and send it to the *Parser*, which processes it, extracts the high-level information and produces a report containing the analysis results. In contrast, the behavior extracted by the Web module serves as input to four different processors, each of them being responsible for one type of malicious behavior detection. The *General classifier* collects the results of these four processors and produces a summarized information report. When the Web module detects an executable file, for example, due to a drive-by download, this is forwarded to the OS module, which returns the sample's observed behavior.

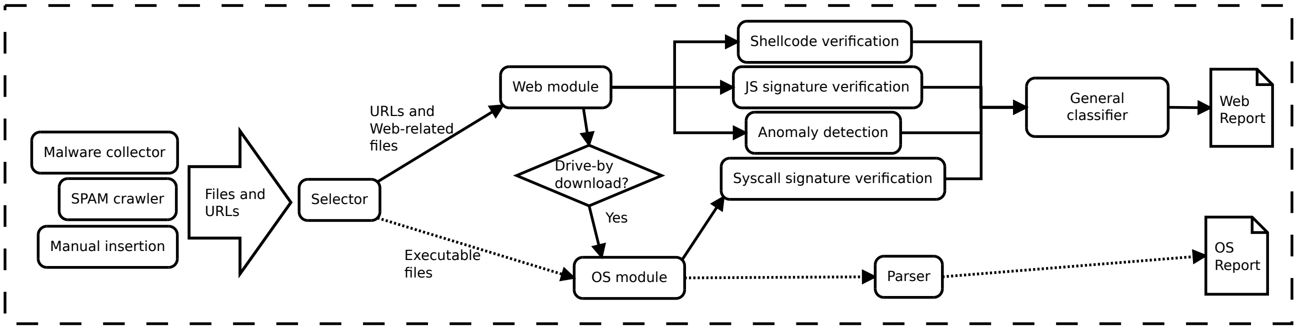


Figure 4.1: Framework's overall architecture.

Collection. Apart from manual insertion, malicious content is obtained by a spam crawler and by malware collectors. The spam crawler periodically fetches emails from an account created especially to receive spam. When the crawler finds a link or an attached file, that object is sent to the *Selector*. Malware collectors are low-interaction honeypots (systems that emulate some vulnerable services) that collect samples by downloading them after attempted exploits.

Web module. The Web module performs its monitoring process through a Windows library (DLL - Dynamic Link Library) that hooks some functions from libraries that are required by the Internet Explorer browser. When one of the monitored functions is called, the execution flow is changed to a function inside the monitoring DLL. It then logs all the needed information and redirects the execution flow back to the original function. We monitor actions that are executed by JavaScript code due to its wide usage in attacks as a client-side script language [47]. The monitored actions are related to string, ActiveX, decoding and array operations, DOM (Document Object Model) modifications, dynamic code execution and manipulation of personal information, such as cookies. The actions that the Web module captures are then sent to the four available detection modules, each one responsible for one type of detection. The windows executable files obtained during the monitoring process by the Web module are sent to the OS one, which in turn forwards the returned system calls to the detector that processes system call signatures. The detection results are used to classify the analyzed sample.

OS Module. The OS module is based on a Windows kernel driver and contains a pool of emulated and real environments. The SSDT hooking technique is used to monitor system calls performed by the analyzed sample and its children-processes. The captured actions are related to file, registry, sync (mutex), process, memory, driver loading and network operations. When it detects the use of some packer that is known to cause problems in emulated environments or when the analysis in the emulated environment finishes with error, the sample is then sent to be analyzed in a real system, i.e. neither emulated nor virtual.

Parser. The *Parser* processes the behavior extracted by the OS module and selects only the relevant actions to insert into the analysis report. An action is considered relevant if it causes a modification in the state of the system or if it incurs in sensitive data leakage. The parser not only screens out the passive and neutral activities that are not interesting within the scope of this thesis, but also applies the behavioral filters to the profile.

BehEMOT (OS Module)

The “OS module” described above consists of a dynamic analysis system designed and deployed to meet the requirements of malware behavior extraction. This system, BehEMOT (Behavioral Evaluation of Malicious Objects Tool), is able to monitor a running malware sample and to capture its operating system actions and network traffic, as well as the actions performed by launched child or hijacked processes. To obtain a malware sample’s profile, the behavioral filters described previously are applied to the BehEMOT output data[1]. An overview of these steps can be seen on Figure 4.2, which shows the data flow from binary files to class assignment (report).

[1] Appendix A shows an example of a BehEMOT analysis report.

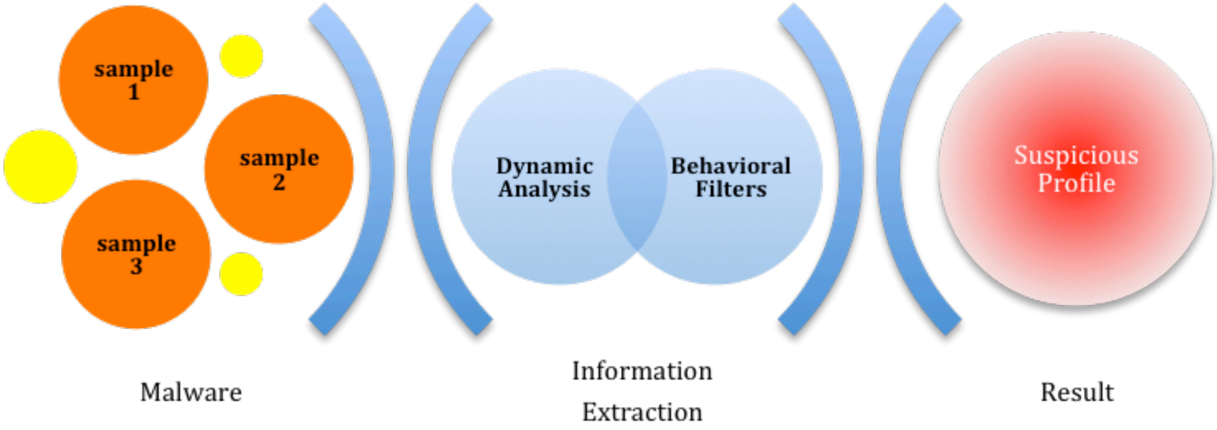


Figure 4.2: Steps to pinpoint suspicious behavior.

4.2.3 Considerations about the Framework

In this thesis, we used the OS module of the aforementioned described framework to dynamically analyze a set of samples and to apply the behavioral filters defined in the previous chapter to the extracted profiles. With these profiles, we were able to evaluate the proposed taxonomy. It is worth noting that the framework yielded promising results either in analyzing malware that may crash other approaches' systems or in detecting Web malware with better results than the most popular approaches. For a more detailed description and the complete results, we point the reader to the work of Afonso et al. [5].

4.3 Taxonomy Evaluation

To validate our proposed taxonomy, we obtained thousands of samples from different sources: spam and phishing e-mail messages, honeypots specially designed to collect malware [152],[153], private and public collections (specifically VxHeavens [160]). We extracted the general behavior of 12,579 unique[2] malware samples. We scanned these samples using three antivirus tools that use distinct engines: F-Prot[3], Avira[4] and AVG[5]. These AV tools provide identification labels in different formats, which then required normalization. For instance, for the well-known malware family dubbed "Allaple", we scanned one sample using the three AVs and obtained labels such as *Worm/Allaple.D*, *WORM/Allaple.Gen* and *W32/Allaple.A.gen!Eldorado*, which were then normalized to simply "allaple". The normalization process aims to provide the family name given to a certain sample and, in case no family name is assigned, we obtain at least its generic class (e.g., backdoor, worm), if available. The goal here is to obtain the distribution of the analyzed samples, as well as to further compare the AV family groups with the behavioral

[2] the uniqueness of the malware samples is based on the binary file's MD5 hash.

[3] http://www.f-prot.com

[4] http://www.avira.com

[5] http://www.avg.com

classes produced by our taxonomy.

4.3.1 Samples Distribution regarding AV Labels

The main objective of using three distinct AV engines is to assign a label to a malware sample, so that disagreements can potentially be solved. If at least two of them agree that a sample is from the same family, or in the case when only one AV detect the sample as malware, then this family label is assigned to that sample. Moreover, it is possible to observe (and analyze) situations in which the three AV engines disagree or they do not detect the sample as a known malware. We add an “UNDECIDED” tag to those samples that the three AV engines assigned distinct labels, and a “CLEAN” tag to those that are not detected by the three AV engines. Table 4.1 shows the distribution of the 12,579 samples in three categories: Undecided (each of the three AV engines providing a distinct label), Clean (the three having not detected the sample) and Labeled (at least two engines having agreed upon the same label or only one engine detected the sample).

Table 4.1: Distribution of malware samples in categories.

	Undecided	**Clean**	**Labeled**
%	44.2	21.0	34.8
Amount	5,559	2,640	4,378

Interestingly, the “UNDECIDED” samples correspond to almost half of the total samples, thus illustrating the confusion that AV engines may cause to a user. Further to that, the 5,559 samples tagged as “UNDECIDED” are distributed among 2,018 unique label combinations. The diversity of the top 15 combinations can be observed in Table 4.2. The samples assigned to these labels amount to just over a quarter of the total 5,559 “UNDECIDED” samples. In this table, we can verify that there are two occurrences of “allaple”, “rahack” and “virut” in a different ordering. It is worth to notice that “rahack” is another name used for “allaple”. Thus, if the AV vendors agreed to rely on a standard scheme to name their detected samples, those 217 (176 + 41) samples would not fall into the “UNDECIDED” group. Moreover, since the behavior of an “allaple” sample is the same of that presented by an “rahack” sample, the assigned labels are useless to explain the infection extension for a user. This corroborates our hypothesis that current AV labels are, in general, meaningless and confusing.

The analysis of the individual AV labels revealed that there are 648 unique identifiers. We show the 15 most frequent ones in Table 4.3, in which it is possible to notice that the three most frequent identifiers are related to the generic labels “gen”, “crypt” and “clean”. The first one means “generic”, a term that is usually assigned when the AV engine is not able to identify a proper family but detects the file as malicious. The second one refers to the type of packer used to obfuscate the malware sample, usually meaning that the file is encrypted. The last label means that the AV engine did not detect the file as malicious. Among the other identifiers, we can observe other generic type terms to describe malware, such as “backdoor”, “Trojan”, “worm” and “downloader”.

The samples tagged as “CLEAN” were not detected by any of the three AV engines used at

Table 4.2: Top 15 unique “UNDECIDED” labels assigned to 1,488 samples.

Label	Amount of Samples
autoit,gen,worm	306
gen,udr,backdoorx	282
allaple,virut,rahack	176
eesbin,gen,skintrim	112
allaple,crypt,rahack	102
gen,vb,worm	91
sheur,-,skintrim	66
clean,crypt,zbot	57
zlob,agent,socks	52
dialer,gen,skintrim	46
gen,crypt,downloader	42
sheur,crypt,fakealert	42
virut,allaple,rahack	41
bifrose,vb,downloader	41
sheur,crypt,downloader	32

Table 4.3: Top 15 unique “UNDECIDED” individual labels.

Individual Label	Amount of Samples
gen	2,831
crypt	1,231
clean	794
agent	709
downloader	680
Trojan	477
worm	436
backdoor	397
allaple	376
rahack	354
spy	334
autoit	322
sheur	317
backdoorx	304
virut	293

the time of the scanning. These 2,640 “CLEAN” samples represent over 20% of the entire sample set. The amount of undetected samples serves to show that it is hard to AV vendors to keep their products up to date, considering their collection and analysis process. Though they have teams to create the new signatures, each AV vendor has its own workflow, which invariably is supervised by humans to avoid the occurrence of false-positives when the update is deployed in a client’s system. We will return to the “CLEAN” subset later in this chapter, when discussing their behavioral classes according to our proposed taxonomy.

As for the “Labeled” set of 4,378 samples, we observed that they are divided into 160 distinct families. Table 4.4 shows the 15 most populated labeled families. Table 4.5 shows the remaining families and the amount of samples within each of them.

Table 4.4: Top 15 labeled malware families and their corresponding amount of samples.

Family	**Amount of Samples (%)**
allaple	1922 (43.9)
swizzor	281 (6.4)
agent	248 (5.7)
spy	140 (3.2)
atraps	139 (3.2)
crypt	124 (2.8)
downloader	121 (2.8)
parite	110 (2.5)
virut	105 (2.4)
generic	96 (2.2)
vb	92 (2.1)
hupigon	88 (2.0)
porndialer	59 (1.3)
expiro	58 (1.3)
banker	50 (1.1)

We notice that the “allaple” family is by far the most prevalent one, at over 40% of the entire dataset. This happens due to the fact that “allaple” samples are very common in the wild, as they scan vulnerable services in large network ranges in order to propagate, thus being more prone to be collected by a honeypot or to be present in malware databases.

4.3.2 Types of Behavior and Classes Found

Previously, we defined four basic malware classes based on a set of types of behavior they can present: “Disrupter”, “Evader”, “Modifier” and “Stealer”. Now, we define some “suspicious” behaviors that, although not malicious, can be indicative of something harmful about to happen (mainly in cases in which a suspicious behavior is followed by a dangerous activity). We include these suspicious (within our scope) behavior in our predefined classes: Language Checking is part of the Evader class and Information Reading of the Stealer one.

Hence, we applied the behavioral filters to the same dataset composed of 12,579 samples in order to classify them according to our taxonomy. First of all, we present an overview of the classes by showing the amount of samples that fell into each one of them. This overview is illustrated in Figure 4.3, in which we can notice that plenty of samples fall on more than one class. We also observe that the majority of malware samples from our dataset are Stealers, closely followed by the Modifiers. Also, 4.58% of samples did not present any of our predefined behaviors. The manual analysis of these samples, labeled as “None” (i.e. those samples that do not presented any of the defined network and filesystem behaviors), revealed that either they crashed during the dynamic analysis, or they performed only passive operations, or even tried to access an unavailable resource and exited. Possibly the main motives for these samples not

Table 4.5: Remaining labeled malware families and their amount.

Families	**# of Samples**
zperm, zorin, zapchast, yabinder, webhancer, vimes, viking, vesic, udef, tibick, threat-hllsi, suspiciouspe, startpage, sober, skintrim, sinowal, sheur, servu, ranky, radmin, qhost, pws, proxy, powerspy, popmon, poisonivy, pexvi, perfloger, pepbot, offend, obfuscate, nuj, newdotnet, netsky, mydoom, messen, mdh, maran, mantis, malum, mabezat, lookme, livetv, lineage, kolab, klone, killwin, istbar, im, hipak, happy, fc, etap, dyfuca, dudu, dsnx, dnschanger, deadcode, cmjspy, ciadoor, cdlmedia, buzy, buzus, bravesent, boxed, ardamax, apdoor, alemod, activitylog	1
clicker, oqa, mostofate, bladi, black, magania, adspy, admedia, casino, dropper, tool, genome, tibs, stration, sasser, nuclear, keenval, beastdoor, bancos, antinny	2
flooder, agobot, ircbot, vundo, packed, prorat, jevafus, zenosearch, passviewer, horse, toolbar, virtool, chir, banload, autorun	3
zhelatin, spybot, polip, gendal	4
korgo, induc, fiha, trojan, adrotator	5
polycrypt, koobface	6
fraudload, bobax, heuristic, malware	7
turkojan, socks, small, file	8
rbot, gobot, bagle	9
legendmir	10
dialer	11
pcclient	12
delphi, cinmus	13
bifrose	14
bho	15
delf	17
ldpinch	18
backdoor, sdbot, adware	20
fakealert	24
salite	25
aliser	27
autoit	29
zlob	35
luder	39
dh	42
onlinegames	45

presenting any suspicious behavior may be due to: i) a corrupted malware file; ii) the detection of the analysis environment by the malware; or iii) a malware that shows split personalities [13].

The distribution of samples among the possible combinations related to our defined malware

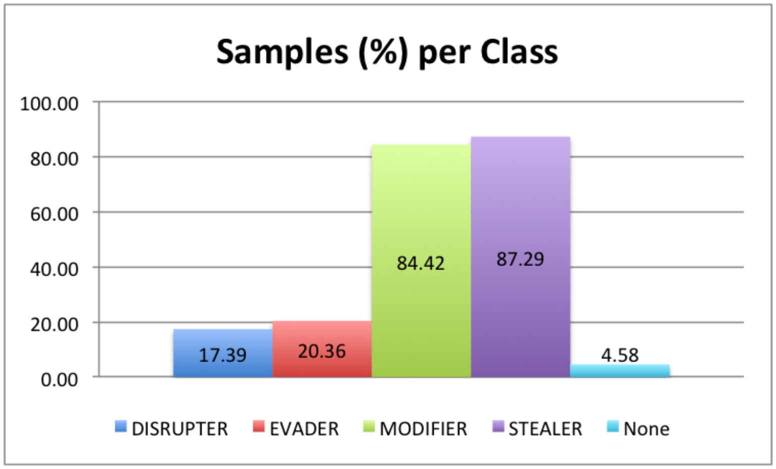


Figure 4.3: Overview of the samples distribution over the classes plus "None".

classes can be seen on the Venn diagram presented in Figure 4.4[6]. This diagram is useful to enrich our perception of how multifaceted current malware is, showing us that nowadays malware perform several malicious operations to accomplish their intent. Based on this Venn diagram, we produced the full multi-class distribution with the respective percentages (Figure 4.5), where "D" stands for "Disrupter", "E" for "Evader", "M" for "Modifier" and "S" for "Stealer". In this figure, it is possible to observe that the majority of our samples are Stealers and Modifiers (46% alone, 17% associated with Evader's behavior, 12% with Disrupter's).

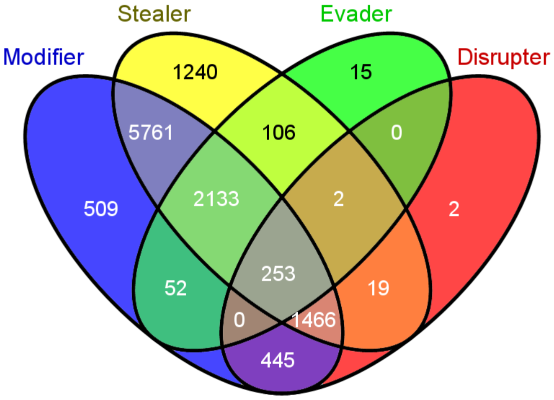


Figure 4.4: Distribution of samples among the possible combinations of classes.

If we go one step deeper within the classes, we can analyze the specific behaviors related to O.S. and network dangerous activities, and the amount of samples that performed each of them. Figure 4.6 illustrates the percentage of samples that presented an O.S. or network-related

[6] This diagram was generated using Olivero's tool [112].

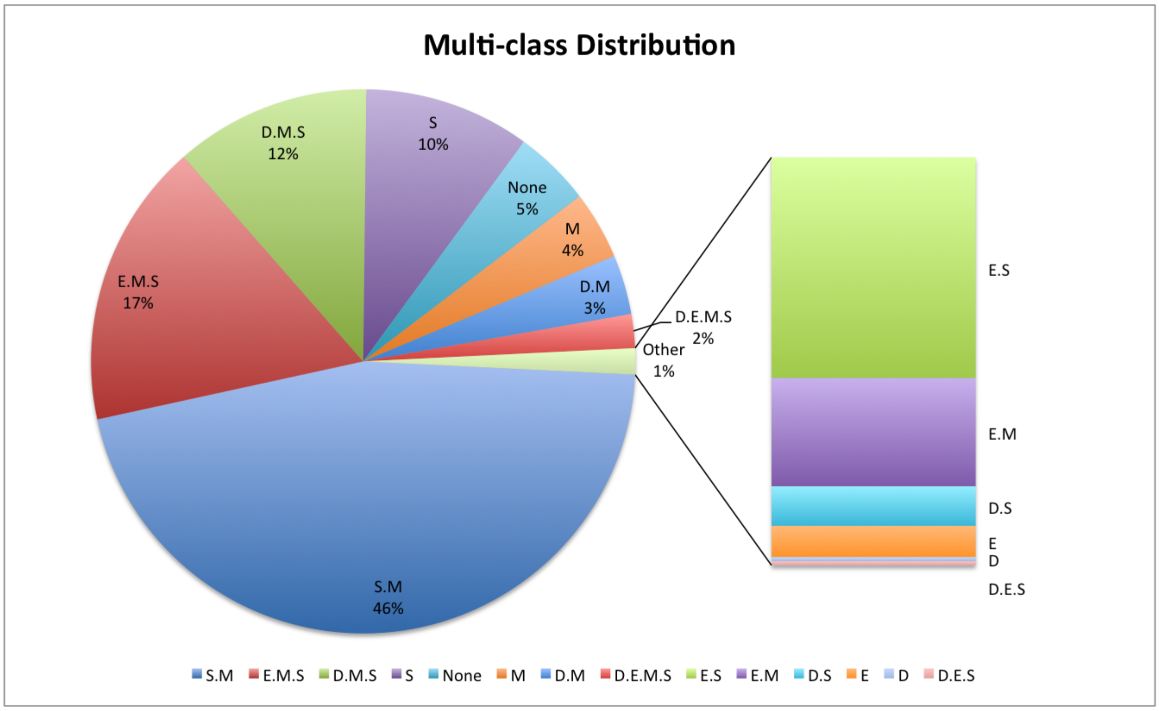


Figure 4.5: Amount of samples distributed over multiple classes (percentages rounded up).

behavior, with the respective class. In this case, the percentage is calculated over the 12,003 samples that presented any behavior, i.e. we excluded those labeled as "None".

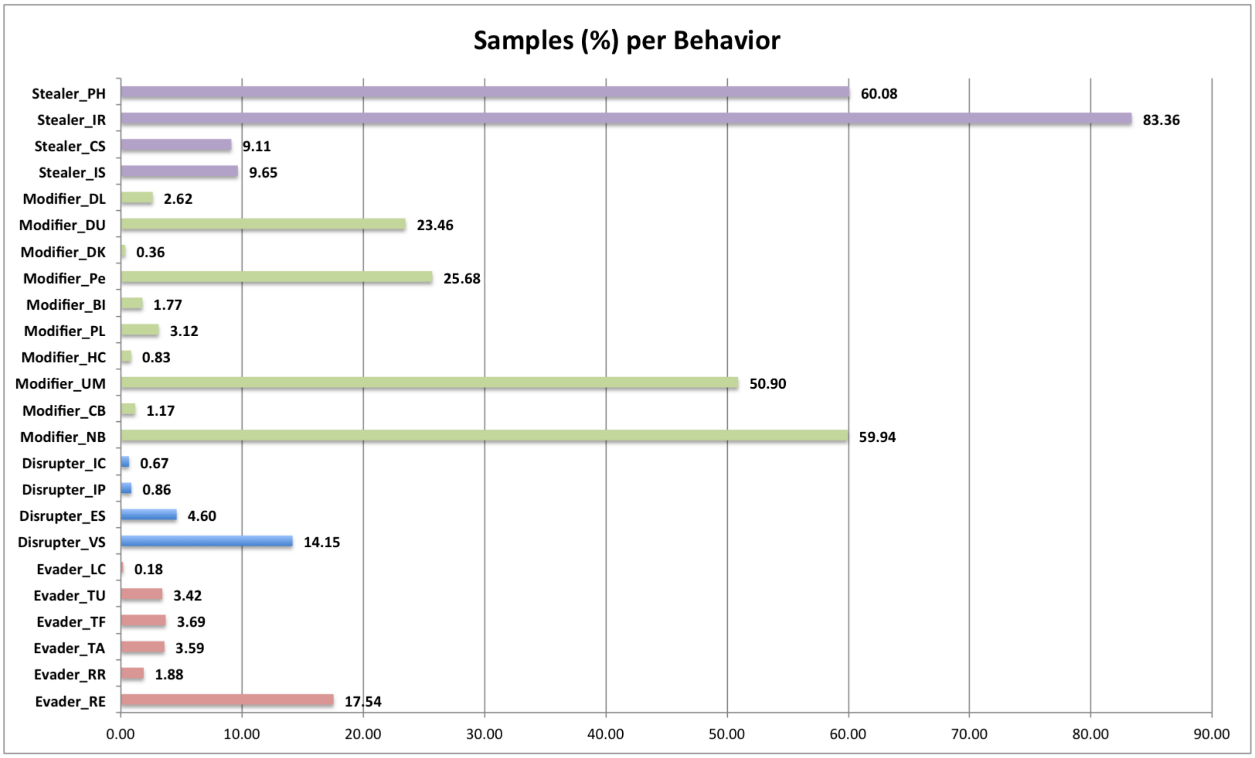


Figure 4.6: Percentage of samples that presented each of the compiled behaviors.

If we carefully analyze Figure 4.6, we notice that the most frequent behaviors (defined in Section 3.4.2) are the less dangerous ones, if found alone. For instance, 83.36% of samples performed "Information Reading (IR)"—a suspicious, passive action, yet very common even to legitimate programs. The same can be said about "Process Hijacking" (PH, 60.08%), "New Binary" (NB, 59.94%), and "Unknown Mutex" (UM, 50.90%). Though these activities may indicate behaviors that are often present in benign programs[7], the knowledge about their association with other more dangerous behaviors can shed a light on a malware-related security incident.

Therefore, an analyst or incident responder may envision the chain of events related to an infection (given a technical report), even in cases where actions that are only apparently not so suspicious are found. For instance, if a malware sample *reads some information* and, based on this, *downloads or drops an unknown file*, registers itself through the creation of an *unknown mutex* to avoid reinfections and, at the end, *hijacks a process*, we can track down the infection process using our proposed taxonomy (e.g., attributing the label "$M_{[DU,UM]}.S_{[IR,PH]}$" to the referred sample). Moreover, the provision of this behavioral aspects to a user depending on the analysis of an unknown file is very informative, since it is possible to know beforehand what that file is supposed to do.

4.4 Discussion

The most valuable contribution of a behavior-centric taxonomy is—besides the high-level information about the malicious intent of an executable file—the possibility of obtaining knowledge about unknown malware. Moreover, the use of our taxonomy allows analysts to be free of antivirus ambiguity. This advantage can be better acknowledged if illustrated with comparisons between samples classified by our taxonomy and by AV assignment.

4.4.1 Top AV Labeled Samples vs. Behavior-Centric Taxonomy

To find out which behaviors are performed by the most frequently detected malware samples, we analyzed them under the lens of our taxonomy. In this section, we will discuss the distinguishing behavior of the ten most populated AV-labeled families from Table 4.4, i.e. the behavior that characterizes most of the samples from those families. Since AV names are meaningless for us, we will not try to explain all of the families' expected behaviors using common AV technical descriptions. Actually, we are going to use the AV labels only to show that the assigned names may have little to no bearing on the samples' behaviors, which can be better described using our behavior-centric taxonomy.

Allaple. First of all, let us analyze the most popular family (according to the AV assignment step) of our samples dataset, the Allaple worm. Allaple's basic behavior includes copying itself to the system, launching its newly "dropped" file as a process and attempting to exploit other

[7] The definition of process hijacking encompasses the same legitimate activity of a program/process "calling" another program or a system process; a benign installer often downloads or drops a new binary; all mutex names that we did not include in our "white-list" are considered unknown. Despite the inconveniences that our definitions may cause in certain cases, these activities are important to enrich the understanding of complex malicious programs.

vulnerable systems across the network [100][131]. Dynamic analysis system's reports includes the creation of random mutexes, so as to guarantee that only one copy is running on the infected machine. According to our taxonomy, the discriminant set of behaviors of our Allaple samples is to write a new binary (**NB**), to launch its process (**PH**), to create an unknown mutex (**UM**), to start a scan of vulnerable network systems (**VS**), and to read system's information (**IR**). Different behaviors are also present in a few samples, as we can see in Table 4.6, but 37% of our Allaple samples exhibited the mentioned behavior, falling into the "$M_{[NB,UM]}.D_{[VS]}.S_{[PH,IR]}$" class. Thus, this may be the discriminant behavior for this family (Figure 4.7).

VulScn				NewBin	
UnkMut					
				InfRdn	PrcHjk

Figure 4.7: Discriminant behavior's matrix for Allaple.

Therefore, our result not only agrees with the family description, but also tells the user what the observed behavior is without those weird or peculiar names commonly found on AV analyses. Furthermore, we are able to search for other unknown samples/families embraced by the same behavior-centric produced class and group them together. While we have chosen to provide, with our taxonomy, higher-level information about the malware classes and their associated behaviors, there is additional, specific data (e.g., mutex and process names, scanned IP addresses or port numbers) that can be found in the report related to the analyzed malware sample. The information contained in this report serves as input to the step that classifies malware according to the proposed taxonomy. An extra 7% of Allaple-labeled samples presented the same previous behavior and, additionally, downloaded an unknown file from the Internet (**DU**), falling into the "$M_{[NB,UM]}.D_{[DU,VS]}.S_{[PH,IR]}$" class; 15.3% of them are "$M_{[UM]}.D_{[VS]}$", 10.6% are "$M_{[NB,UM]}.S_{[PH,IR]}$", and 10.0% are "$M_{[NB]}.S_{[PH,IR]}$".

Table 4.6: Allaple: AV-assigned family vs. behavior-centric taxonomy.

AV Family	**PH**	**IR**	**NB**	**UM**	**IP**	**VS**	**ES**	**IC**	**DU**
allaple	73.2%	73.3%	73.2%	87.8%	0.1%	70.1%	7.8%	0.3%	15.9%

Swizzor. Swizzor samples are divided into ten classes, 75.8% of them distributed over "$S_{[IR,PH]}$" (35.6%), "$M_{[UM,NB]}.S_{[IR,PH]}$" (26%), "$M_{[UM]}.S_{[IR,PH]}$" (10.3%) and "$M_{[DU]}$. $S_{[IR,PH]}$" (3.9%). The remaining samples are in classes that are variations of the same five behaviors present in those four more populated classes. Since IR and PH correspond to behaviors that, alone, are not too much informative about the malignity of a certain sample, we chose the discriminant behavior to be "$M_{[UM,NB]}.S_{[IR,PH]}$" (Figure 4.8). Notice that the discriminant behavior of Parite and Allaple differs only by the vulnerability scan exhibited in the latter.

Spy. We expect that samples whose assigned label is "Spy" present spyware behavior, trying to steal information if possible. When we applied our taxonomy to the available Spy samples, we obtained 54 distinct classes. The most populated classes are "$E_{[RE]}.M_{[Pe]}$" (10.9%), "$E_{[RE]}.M_{[DU,Pe]}.S_{[IS,CS]}$" (5.5%), "$S_{[IS,CS]}$" (5.5%), $S_{[IR]}$ (5.5%), "$M_{[Pe]}$" (4.7%), "$E_{[RE]}.M_{[Pe]}.S_{[IS,CS]}$"

				NewBin	
UnkMut					
				InfRdn	PrcHjk

Figure 4.8: Discriminant behavior's matrix for Swizzor.

(3.9%), "$E_{[RE]}.M_{[Pe]}.S_{[IR]}$" (3.9%) and "$S_{[IS,CS,PH]}$" (3.9%). Most of one-element classes exhibited behaviors related to information and credential stealing, as well as PAC loading and known malware downloading. We confirmed our hypothesis about the behavior of Spy samples by verifying that 57.0% of them performed information stealing wehereas 50.9% performed credential stealing.

Atraps. These samples produced 58 classes. 11.9% of Atraps samples fell in the "$M_{[DU]}.S_{[IS,CS]}$" class, which we chose as the discriminant behavior. In addition, 19.8% of Atraps samples exhibited the discrimant behavior along with other behaviors. Another 11.9% of samples fell in the "$S_{[IS,CS]}$" class, 5.5% in "$M_{[DU]}.S_{[IR]}$", 4.0% in "$M_{[DU]}.S_{[IS,CS,IR]}$", 4.0% in "$S_{[IR]}$", and so on. Figure 4.9 illustrates Atraps' discriminant behavior.

DldUnk		InfStl	CrdStl		

Figure 4.9: Discriminant behavior's matrix for Atraps.

Crypt. The discriminant behavior for Crypt samples is "$E_{[RE]}.M_{[NB,Pe]}.S_{[IS,CS,PH]}$" (Figure 4.10), which is observed in 8.6% of them. Other most populated classes include "$E_{[RE]}.M_{[Pe]}.S_{[IR]}$" (7.7%) and "$E_{[RE]}.M_{[NB,Pe]}.S_{[PH]}$" (6.9%). Crypt samples produced 63 classes, 44 of them with one element. The behaviors found in these one-element classes vary from loading PAC files (**PL**) and drivers (**DL**) to changing the "hosts" file (**HC**) and sending e-mail (**ES**).

RemEvd					
				NewBin	
				Persis	
		InfStl	CrdStl		PrcHjk

Figure 4.10: Discriminant behavior's matrix for Crypt.

Downloader. We identified that 33.9% of these samples performed downloads of known malware or unknown files. However, 49.6% of them created new files on the system, indicating that some of these samples effectively "dropped" these files instead of downloading them. The most populated of the 51 produced classes are "$E_{[RE]}.M_{[NB]}.S_{[IR,PH]}$" (9.6%), "$S_{[IS,CS,IR]}$" (7.8%), "$E_{[RE]}.M_{[NB]}.S_{[IS,CS,IR,PH]}$" (6.9%) and "$S_{[IR]}$" (6.9%). Downloader samples produced 32 one-element classes that exhibited, among other behaviors, vulnerability scanning (**VS**), known IRC ports communication (**IP**), persistence to survive to reboots (**Pe**) and e-mail sending (**ES**).

Parite. These samples produced 24 distinct classes, 16 of them with only one element. 35.4% of Parite samples fell in the "$S_{[IR]}$" class, 9.1% in "$M_{[UM]}.S_{[IR]}$", 7.3% in "$M_{[NB]}.S_{[IR,PH]}$", and so on. More interesting are those classes that contain only one sample, since they are a clear example of the nonsense regarding AV labeling. To properly name the Parite samples while still showing the richness of their behavioral diversity, the items below show each of their one-element class:

- $D_{[ES]}.E_{[RE]}.M_{[UM,NB,Pe]}.S_{[IR,PH]}$
- $D_{[IP,IC]}.S_{[IR,PH]}$
- $M_{[DU]}$
- $M_{[DU,UM]}$
- $E_{[RE]}.M_{[DL,DU,UM,NB,Pe]}.S_{[IR,PH]}$
- $M_{[DL,DU,UM,NB,Pe]}.S_{[IR,PH]}$
- $M_{[DU,UM]}.S_{[IR]}$
- $M_{[NB,HC,Pe]}.S_{[IR]}$
- $E_{[RE]}.M_{[NB,Pe]}.S_{[IR,PH]}$
- $M_{[NB,Pe]}.S_{[IR,PH]}$
- $M_{[Pe]}$
- $M_{[Pe]}.S_{[PH]}$
- $E_{[RE]}.M_{[UM,NB]}.S_{[IR,PH]}$
- $E_{[RE]}.M_{[NB]}.S_{[IS,IR,PH]}$
- $E_{[RE]}.M_{[NB,Pe]}.S_{[CS,IS,IR,PH]}$
- $S_{[PH]}$

This behavioral diversity makes it difficult to define a discriminant behavior for Parite samples without a deeper analysis of the samples.

Virut. Virut samples produced 43 classes, 24 of them with one element. Again, the one-element classes exhibit a more diverse set of behaviors, such as IRC communication and known port access, e-mail sending, "hosts" file modification, vulnerability scanning and download of already known malware. However, the most populated classes are "$M_{[DU,UM,NB,HC]}.S_{[IR,PH]}$" (14.5%), "$D_{[VS]}.M_{[UM]}$" (10.0%), "$D_{[VS]}.M_{[UM,NB]}.S_{[IR,PH]}$" (4.5%), "$M_{[UM,NB,HC]}.S_{[IR,PH]}$" (4.5%), and "$D_{[VS]}.M_{[DU,UM]}$" (4.5%). We consider the first one as the discriminant behavior of Virut (Figure 4.11).

Agent. These samples were distributed among 37 classes of our behavior-centric taxonomy, 21 of them with a single element. Similarly to Parite and Virut, the most varied Agent samples (regarding the behavior's diversity) are not associated with the most populated classes. Class "$S_{[IR,PH]}$" encompasses 17.0% of the total Agent samples and class "$S_{[IR]}$", 15.9%. One-element classes include:

				NewBin	
UnkMut	HstChg				
DldUnk				InfRdn	PrcHjk

Figure 4.11: Discriminant behavior's matrix for Virut.

- $M_{[DU,NB,Pe]}.S_{[PH]}$
- $M_{[DU,NB]}.S_{[IR]}$
- $M_{[DU,NB]}.S_{[IR,PH]}$
- $M_{[DU,NB,CB]}.S_{[IR,PH]}$
- $M_{[DU,UM,NB,BI]}$
- $M_{[DU,UM,NB]}.S_{[PH]}$
- $M_{[DU,UM]}.S_{[IR]}$
- $E_{[RE]}.M_{[NB]}$
- $M_{[NB,Pe,BI]}.S_{[IR,PH]}$
- $M_{[UM,CB]}.S_{[IR,PH]}$
- $E_{[RE]}.M_{[UM,NB]}.S_{[PH]}$
- $M_{[UM,NB,BI]}$
- $M_{[UM,NB]}.S_{[IR]}$
- $D_{[VS]}.M_{[UM,NB]}.S_{[IR]}$
- $M_{[UM,NB,Pe]}.S_{[IR]}$
- $M_{[UM,DL]}.S_{[IR]}$
- $M_{[UM,DL]}.S_{[IR,PH]}$
- $M_{[UM,Pe]}.S_{[IR]}$
- $M_{[Pe]}.S_{[IR]}$
- $E_{[RE]}.M_{[UM,Pe]}.S_{[IS,IR]}$
- $S_{[IS,IR,PH]}$

Generic. Generic samples produced 41 distinct classes, 25 of them with one element. 10% of the total samples were classified as "$S_{[IR]}$", 8.9% as "$M_{[NB]}.S_{[IR,PH]}$", 7.8% as "$D_{[ES]}.E_{[RE]}.M_{[PL]}.S_{[IS,CS,IR,PH]}$", and so on. From the 24 defined behaviors, only 10 did not show up in Generic samples (**TA, TU, TF, CB, HC, DK, VS, IP, IC, LC**).

VB. Similarly to the Generic samples, the two most populated VB classes are "$S_{[IR]}$" (8.6%) and "$M_{[NB]}.S_{[IR,PH]}$" (8.6%). These samples produced 47 classes, 33 of them with one element. The diversity of behaviors exhibited by the observed classes includes a sample that tried to terminate security mechanisms—assigned to the "$E_{[TA,TF,TU]}$" class—and a sample that presented half of the possible behaviors—assigned to the "$E_{[TA,TF,RE]}.D_{[ES]}.\ M_{[PL,UM,NB,Pe]}.S_{[IS,CS,IR,PH]}$" class.

Hupigon. 41.3% of Hupigon samples were classified as "$M_{[UM,NB]}.S_{[IR,PH]}$", 18.4% as "$M_{[UM]}.S_{[IR,PH]}$", 6.9% as "$E_{[RE]}.M_{[UM,NB]}.S_{[IR,PH]}$", 5.7% as "$M_{[NB]}.S_{[IR,PH]}$", and 4.6% as "$M_{[DU,UM,NB]}.S_{[IR,PH]}$". On the group of the single-element classes, we observed the odd behavior of loading a driver in one sample, and the stealing of system or user information through the network in another one. Considering only the high-level exhibited behavior, we can notice that the discriminant behavior of Hupigon samples (Figure 4.12) is the same as Swizzor samples' (Figure 4.8).

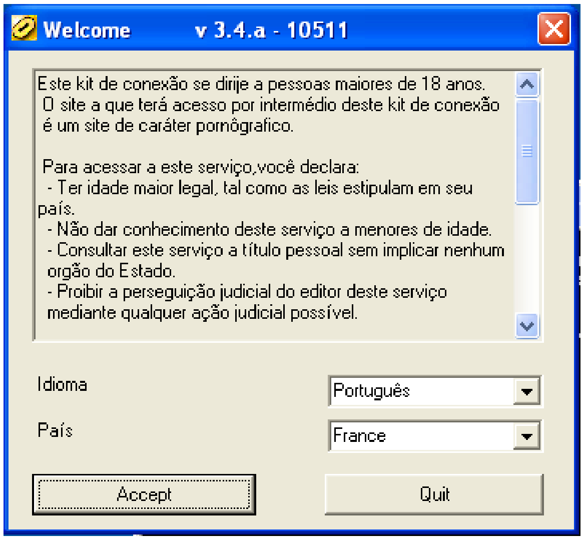
				NewBin	
UnkMut					
				InfRdn	PrcHjk

Figure 4.12: Discriminant behavior's matrix for Hupigon.

Porndialer. These samples were grouped into only two classes: 83% in "$M_{[UM,NB]}.S_{[IR]}$" and 17% in "$M_{[DU,UM,NB]}.S_{[IR]}$", behaving like droppers and downloaders, respectively. It makes sense, since this type of malware is a program that executes (or downloads and executes) a software that intends to dial telephone numbers whose call will be charged to the victim. Figure 4.13 shows an image captured during BehEMOT's dynamic analysis step.

Expiro. Expiro samples' discriminant behavior is "$M_{[UM,CB,NB]}.S_{[IR]}$" (Figure 4.14), which is present in 24.1% of the samples. This warns us about the fact that some of these samples modify system programs. 22.4% of samples are "$M_{[UM]}.S_{[IR]}$", 10.3% of them are "$D_{[VS]}.M_{[UM,NB]}.S_{[IR,PH]}$", another 10.3% are "$M_{[UM,DU,CB,NB]}.S_{[IR]}$", 8.6% are "$M_{[UM,NB]}.S_{[IR,PH]}$", 5.2% are "$M_{[UM,CB,NB]}.S_{[IR,PH]}$", one sample tried to send e-mail from the victim's machine and to perform a vulnerability scanning (among other behaviors), and another one tried to download an unknown file and to perform a scan for vulnerable services. The remaining samples performed a combination of the behaviors already exhibited by this family.

Banker. The name "banker" is usually assigned to a special type of malware whose main purpose is to steal Internet banking credentials from infected victims. Samples labeled as bankers, however, behaved in 25 distinct ways. 10.9% exhibited the behaviors from the "$M_{[NB,Pe]}.S_{[IR,PH]}$" class, 8.7% from "$E_{[RE]}.M_{[UM,NB,Pe,BI]}.S_{[IR,PH]}$", 6.5% from "$M_{[NB,Pe]}.S_{[IR]}$", and another 6.5% from "$D_{[ES]}.M_{[NB,Pe]}.S_{[IR,PH]}$". Interestingly, we were able to pinpoint the credential stealing behavior in only one of those AV-labeled bankers, and over 80% of them tried to persist in the infected system after a reboot. We address bankers in more detail in Chapter 5.



Welcome v 3.4.a - 10511

Este kit de conexão se dirije a pessoas maiores de 18 anos.
O site a que terá acesso por intermédio deste kit de conexão é um site de caráter pornôgrafico.

Para acessar a este serviço,você declara:
- Ter idade maior legal, tal como as leis estipulam em seu país.
- Não dar conhecimento deste serviço a menores de idade.
- Consultar este serviço a título pessoal sem implicar nenhum orgão do Estado.
- Proibir a perseguição judicial do editor deste serviço mediante qualquer ação judicial possível.

Idioma: Português

País: France

Accept

Quit

Figure 4.13: Screenshot of a Porndialer sample execution.

				NewBin	ChgBin
UnkMut					
				InfRdn	

Figure 4.14: Discriminant behavior's matrix for Expiro.

4.4.2 Behavior of Unknown Malware

Another useful feature of a behavior-centric taxonomy is to group samples that the AV engines did not detect as malicious together with samples that exhibits the same behavior. Therefore, those "undecided" samples can also be associated to their rightful pairs. For instance, the class $M_{[UM,Pe,NB]}.S_{[IR,PH]}$ is composed of 358 "undecided", 2 "clean", and 34 samples with variable labels (e.g., agent, backdoor, bagle, banker, bifrose, chir, fakealert, koobface, legendmir, nuclear, sdbot, skintrin, vb). This turns the AV label unnecessary, since we know that, among other behaviors, those samples try to persist on the infected system and they create a new binary file (apparently a "drop" behavior).

We found 463 unique combinations of classes and behaviors for the 5,559 "undecided" (regarding AV labels) samples. Table 4.7 shows the 15 most frequent labels for the samples in which the AVs did not agree, according to our behavior-centric taxonomy. Figure 4.15 shows the distribution of behaviors among those "undecided" samples.

The undetected samples ("clean") are even more important, since they pose a major threat to AV-powered users. We were able to extract the behavior of 91.7% of the samples labeled as "clean". Then, we found 332 unique combinations of classes and behaviors for these 2,422 "clean" samples. We show the 15 most frequent labels in Table 4.8, and the overall distribution of behaviors in Figure 4.16. It is worth noting the type and frequency of behaviors exhibited by those "clean" samples. For instance, about a quarter of them performed information and credential stealing; over 40.0% of them downloaded unknown files; over 10.0% of them loaded a PAC file to change the browser's proxy. Therefore, the knowledge about these potentially

Table 4.7: Top 15 Behavioral-Centric Taxonomy labels for AV-“UNDECIDED” samples.

Behavioral Label	**% of Samples**
$M_{[UM,NB,Pe]}.S_{[IR,PH]}$	6.4
$M_{[NB]}.S_{[IR,PH]}$	6.2
$S_{[IR]}$	5.2
$M_{[UM,NB]}.S_{[IR,PH]}$	5.0
$D_{[VS]}.M_{[UM,NB]}.S_{[IR,PH]}$	3.3
$M_{[UM,NB]}.S_{[IR]}$	3.1
$M_{[NB]}.S_{[IR]}$	3.1
$M_{[UM]}.S_{[IR]}$	2.6
$S_{[IR,PH]}$	2.5
$M_{[NB,Pe]}.S_{[IR]}$	2.2
$M_{[NB,Pe]}.S_{[IR,PH]}$	2.1
$E_{[TA,TF,TU,RR]}.M_{[NB,Pe]}.S_{[IR,PH]}$	2.1
$E_{[RE]}.M_{[NB]}.S_{[IR,PH]}$	1.8
$E_{[RE]}.M_{[UM,NB,Pe]}.S_{[IR,PH]}$	1.6
$E_{[RE]}.M_{[UM,NB]}.S_{[IR,PH]}$	1.6

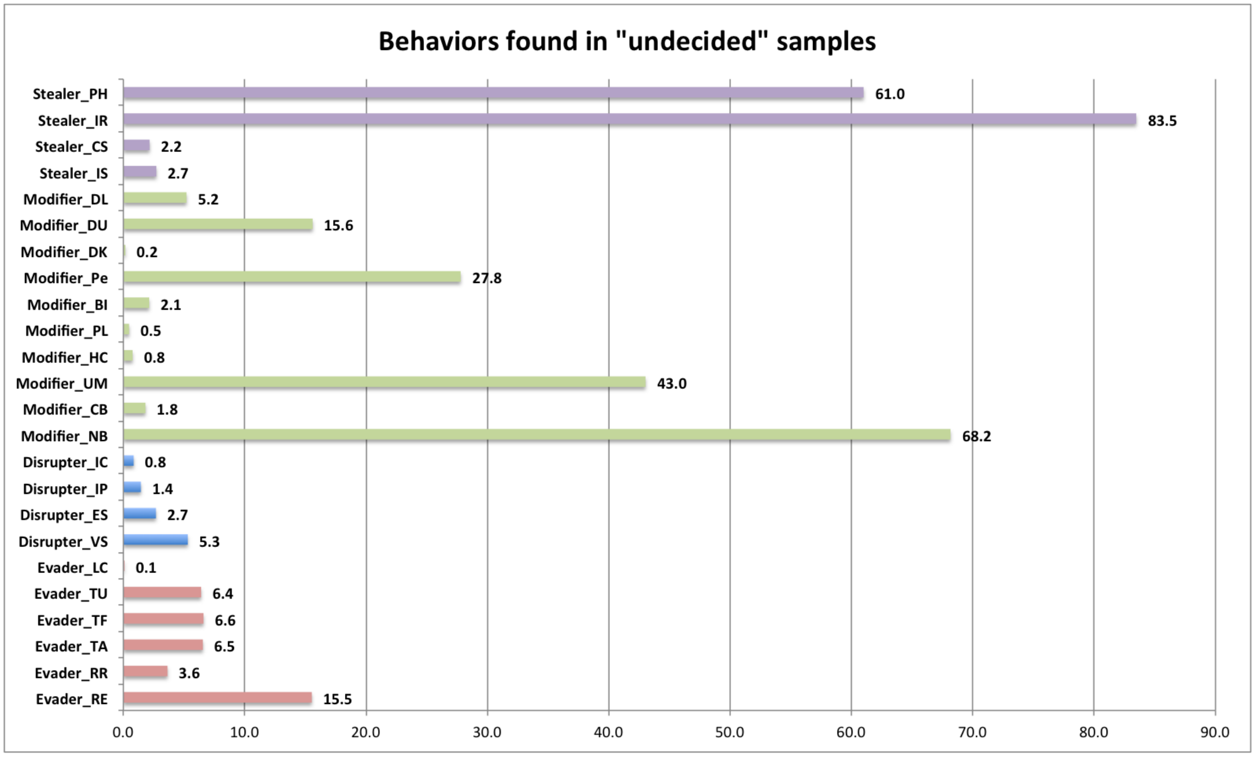


Figure 4.15: Behaviors exhibited by AV-“UNDECIDED” samples.

dangerous behaviors show that our proposed taxonomy can effectively help users to decide whether a program that their AV considered as clean will run or not.

Table 4.8: Top 15 Behavioral-Centric Taxonomy labels for AV-“CLEAN” samples.

Behavioral Label	% of Samples
$M_{[DU,UM,Pe]}.S_{[IR]}$	16.4
$M_{[NB]}.S_{[IR,PH]}$	8.4
$S_{[IR]}$	8.3
$M_{[DU,UM,Pe]}.S_{[IR,PH]}$	5.8
$E_{[RE]}.M_{[DU,UM,NB,Pe]}.S_{[IR,PH]}$	4.2
$M_{[DU,NB]}.S_{[IR,PH]}$	2.5
$M_{[UM]}.S_{[IR]}$	1.9
$M_{[UM,Pe]}.S_{[IR,PH]}$	1.7
$M_{[UM]}.S_{[IR,PH]}$	1.6
$E_{[RE]}.M_{[UM,NB]}.S_{[IR,PH]}$	1.5
$M_{[DU]}.S_{[CS,IS]}$	1.4
$M_{[DU]}.S_{[IR]}$	1.4
$S_{[CS,IS]}$	1.4
$D_{[ES]}.M_{[PL]}.S_{[CS,IS]}$	1.2
$M_{[DU]}.S_{[CS,IS,IR]}$	1.2

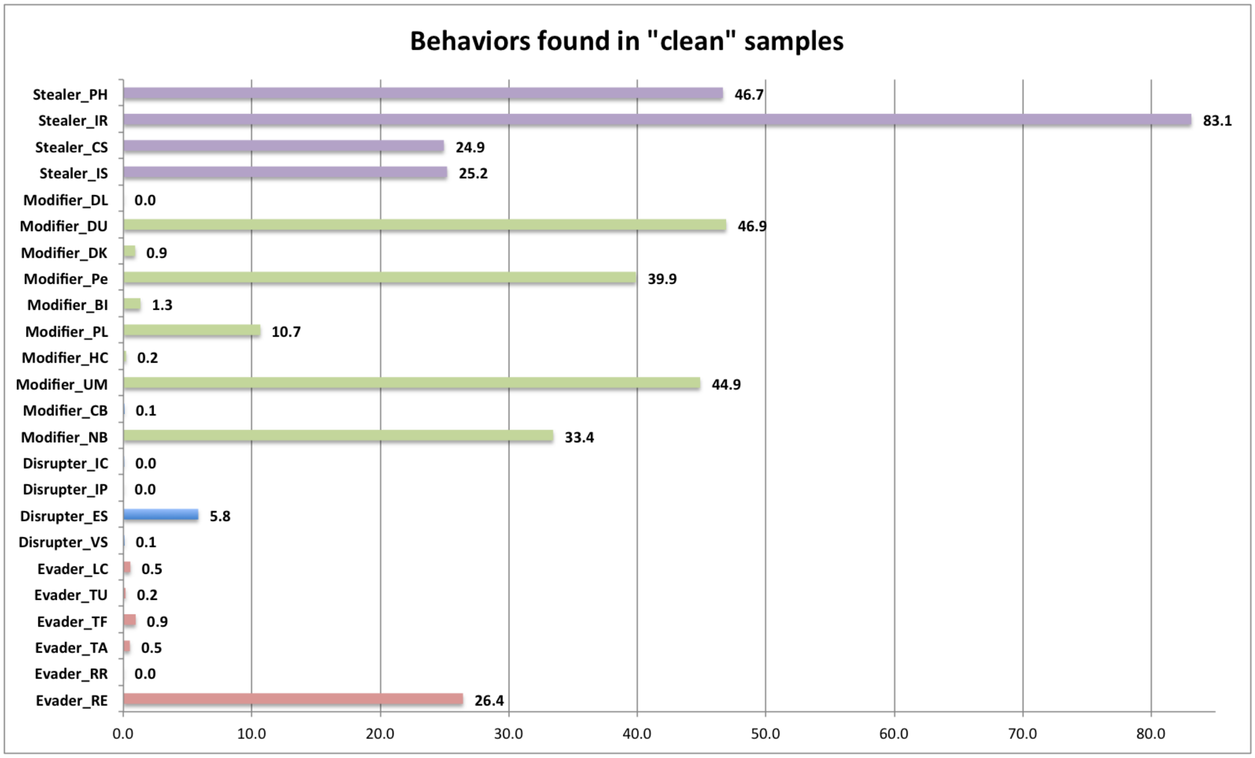


Figure 4.16: Behaviors exhibited by AV-“CLEAN” samples.

4.5 Concluding Remarks

In this Chapter, we presented an evaluation of our proposed behavior-centric taxonomy. Initially, we discussed dynamic malware analysis and introduced our framework that extracts behavioral profiles. We also presented the distribution of our samples regarding AV labels, obtaining the amount of samples for which the AV engines did not agree while assigning a label, the undetected ones and those that were assigned to a family. Then, we presented the results of our taxonomy's application to our sample dataset, grouping those samples (not exclusively) in four classes: Disrupter, Evader, Modifier and Stealer. This evaluation corroborates the fact that current malware are very complex and multi-purposeful. Finally, we discussed our results and showed that our taxonomy can guide users and professionals due to its informative orientation.

The discussion showed us, on one hand, that our taxonomy describes these samples according to their dangerous activities, such as persistence on the victim's system, as well as qualifies them as capable of data stealing and modificaton during an infection. On the other hand, we are limited to describe only samples that fit into our scheme, that is, those that exhibit behaviors that belong to our taxonomy, and even so, only if the behavior extraction is successfully accomplished, that is, the dynamic analyzer that was used being able to capture the monitored malware behavior. Nonetheless, these problems can be technically solved, either by adding more behaviors to the taxonomy, or by changing the monitoring techniques used in the dynamic analysis system.

Chapter 5

Malware Detection

5.1 Introduction

Since one of the main objectives of this thesis is to provide a taxonomy that may be easily extended and to show that our taxonomy can also be used in restricted scopes to, for instance, identify specific types of malware, we introduce BanDIT (the Banker Detection and Infection Tracking system), a specialization of our system that intends to address malicious Internet Banking software (a.k.a bankers). In this chapter, we motivate the study of bankers' behavior, briefly present an Internet Banking background, describe BanDIT's architecture and leverage bankers' identification results using an extended, yet specialized, version of our taxonomy. Therefore, this chapter can be considered a case-study about bankers.

5.2 Motivation

Crimeware, phishing Trojan or banking Trojan are terms used to refer to a special type of malware whose main objective is to obtain online banking credentials in order to steal money [51]. These malware samples, commonly known as "bankers", make use of social engineering techniques and phishing e-mail messages to trick users into providing their Internet banking credentials, credit or debit card numbers, and security token values to take over the victim's online bank account or to resell these pieces of information in underground markets. The high success rate of such attacks results in billions of dollars in losses worldwide [158][71].

Recent examples of widespread bankers are ZeuS [20] and Spyeye [37], which infect users by modifying files and system libraries, and by injecting their code into system processes. Conversely, Brazilian bankers usually infect users' machines by falsely warning them that they need to update their Internet banking defense mechanisms or to confirm their authentication data. This type of banker has been seen in the Brazilian Internet space for at least ten years: in late 2006, they amounted to 30.7% of the observed bankers in [51] (the remaining ones targeted mainly European, Australian and North American banks). More recently, a report from Kaspersky [80] pointed Brazil as the country most frequently targeted by bankers.

We believe that this happens due to two main reasons: Brazil has about 28 million online bank users and there are no specific laws (until very recently) regarding computer-based crimes, turning the scenario into a favorable one for cyber criminals. This situation created an underground commerce in which attackers sell not only valid credit card numbers, but do-it-yourself kits—to compile Delphi or Visual Basic source codes offered at social networks, such as Orkut—that allows for a quick production of customized variants. In spite of the focus of this chapter lying on Brazilian bankers, mainly because our samples were obtained from phishing or collectors in Brazilian cyberspace, the detection techniques proposed here are applicable to any other banker that acts in the same way, only requiring updates to the bankers detection pattern lists and images database.

Despite previous works [57][27][51] regarding the detection of bankers, there is a lack of initiatives to automatically identify this type of malware during the dynamic analysis in publicly available systems (e.g., Anubis [86], CWSandbox [168], ThreatExpert [155]). However, these systems are largely used by incident response teams around the world as a first step to provide information about a malware incident during the response procedures. Hence, the identification of a banker right on the dynamic analysis step should yield a more efficient development of protective measures, such as sending alerts to ISPs, deeper investigation of the banker's file and execution behavior to find out information about the possible damage extent, production and deployment of block lists, and early warning to law enforcement authorities. To this end, we propose BanDIT, a dynamic analysis system that provides identification of bankers and tracking of servers used in banker infections.

In spite of some of the techniques presented on this chapter not being exactly novel, we are not aware of any other work that (i) yields the combined use of these techniques, (ii) applies them to identify bankers as a complement to dynamic analysis system reports, and (iii) focuses on the identification of Brazilian bankers based on their traits. Thus, the main purpose of BanDIT is to aggregate value to the dynamic analysis of malware by identifying bankers that behave like Brazilian ones, making it possible to screen them out for further analysis. The main contributions of this chapter are as follows:

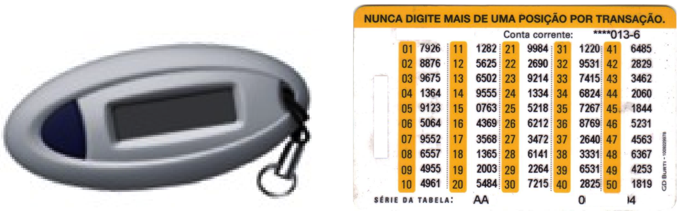
- We proposed a scheme to analyze and identify malware samples that act as bankers through a combination of visual similarity identification, network signature matching and file system change monitoring.

- We implemented a prototype of our proposed system (BanDIT) and analyzed over 15 hundred unique malware samples, mostly from phishing e-mail messages. Furthermore, we leveraged an empirical analysis of the execution behavior presented by the bankers that were identified through BanDIT.

- We reported all the e-mail and IP addresses/domains that were involved in bankers infections (used for sending/receiving stolen credentials, or as a download base for other malware pieces). We also developed a Web site for tracking infected IP addresses related to malware samples that attack Internet banking users.

5.3 Background: Internet Banking and Bankers

Internet banking access brought convenience to users, allowing them to perform money transfers, payments and account status verification without going physically to a bank agency. However, Internet access is also convenient to attackers, as they can subtract money funds anonymously while avoiding getting shot or arrested. To minimize risks, banks deploy security mechanisms intended to protect online transactions, which range from mailed password tables to hardware tokens (Figure 5.1). Together with the bank account card passwords, these mechanisms provide additional authentication factors, making it harder to forge a valid transaction.



NUNCA DIGITE MAIS DE UMA POSIÇÃO POR TRANSAÇÃO.

Conta corrente: ****013-6

01	7926	11	1282	21	9984	31	1220	41	6485
02	8876	12	5625	22	2690	32	9531	42	2829
03	9675	13	6502	23	9214	33	7415	43	3462
04	1364	14	9555	24	1334	34	6824	44	2060
05	9123	15	0763	25	5218	35	7267	45	1844
06	5064	16	4369	26	6212	36	8769	46	5231
07	9552	17	3568	27	3472	37	2640	47	4563
08	6557	18	1365	28	6141	38	3331	48	6367
09	4955	19	2003	29	2264	39	6531	49	4253
10	4961	20	5484	30	7215	40	2825	50	1819

SÉRIE DA TABELA: AA 0 4

(a) Hardware token (b) Password table

Figure 5.1: Additional Internet banking authentication factors

5.3.1 Hardware Tokens

Some banks provide hardware (security) tokens to their clients so that they have a second authentication factor. Thus, every time a client performs an online financial operation, he/she must provide the number that is currently showing in the device's display, or else the operation is not completed. Moreover, this kind of authentication has a short timeout, forcing the client to input several different numbers if performing several operations within anything but a short time frame. This approach is useful to secure the clients online bank accounts even when their access credentials are compromised. However, due to the complexity (and cost) to deploy and manage millions of users/devices and synchronize their security tokens, few banks are using this type of device and those which are usually provide it only to enterprise users.

5.3.2 Password Tables

To increase the security of an online financial transaction while promoting another authentication (cheaper than a hardware token), some banks provide password tables to their clients. When a client performs an online financial operation, the Internet banking server requests the value of some index from this table to validate that operation. Differently from hardware tokens, password tables' indexes can be reused, as the table may not be changed frequently [1]. In addition, it is common that the same password table index is used during an entire Internet banking session, no matter how many financial operations are performed. Security is in fact achieved by setting a very low number of failed attempts at a transaction (usually three to five) before a full lock out of the account, after which the user is forced to use yet another set of

[1] I used to have a password table that did not change for about two years.

authentication factors and communication channels to restore online operation. This may also involve a visit to a bank branch.

5.3.3 Anatomy of a Banker Infection

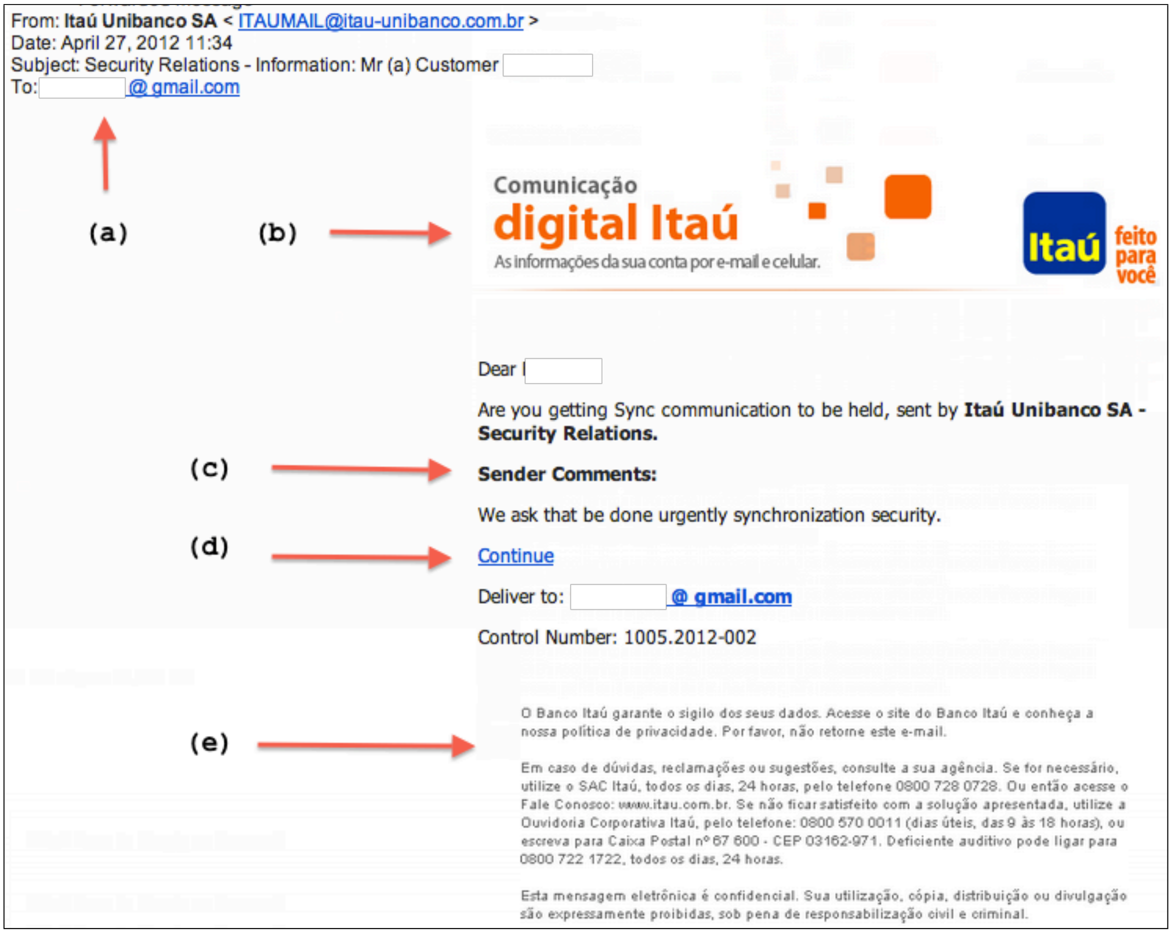
Banker attacks affect both server and client sides. On the server side, possible actions that attackers can take are: i) to register "mistyped" bank domains and deploy sites that clone the legitimate Internet banking site, in order to deceive careless users and ii) to compromise vulnerable servers/sites, so they can host a clone of the targeted Internet banking site or the banker executable. Furthermore, attackers may spread phishing messages containing attached banker executable files, or links to fake/compromised sites (to directly grab information using cloned interfaces or to download the banker).

On the client side, bankers leverage a myriad of techniques to mislead users, such as the overlapping of Internet banking form fields, the interception of network communications, the modification of name resolution and proxy configuration files, the sending of e-mail messages warning about updates to the Internet banking security solution installed on the victim's systems, the use of keyloggers that filter Internet banking content to log only important data and so on. In the last few years, we have been observing that most of the Brazilian bankers infect users through sending of phishing content that leads to the download of an executable. Once downloaded and installed, this executable presents a GUI (which may simulate a browser) that "guides" the victim into providing sensitive information (e.g., bank agency, account number, credit card number and verification code, Internet banking password, token values, password table values etc.).

This trend may be due to the fact that users can easily realize the fraud in infections where the attacker supplies a (weird) link to a cloned site, such as `http://something-in- another-country/http/www.mybank/index.php`. Apparently, it makes more sense to the user to download and execute files named `<bankname>-iToken.exe` or `<bankname>- security-update.exe`. Although Brazilian banks claim that they do not send messages regarding either sensitive data and security mechanisms, or asking for their clients' information, banker attacks have been and are still being performed successfully on a steady pace.

To pretend legitimacy, a successful phisher has to insert some items in the message to convince users to install the banker and inadvertently compromise their machines and online banking credentials. A typical phishing message is illustrated in Figure 5.2—(a) the "from" field contains an address that seems to come from the bank, (b) the body header has the bank logos, (c) the body message request a security update (token synchronization) and provides a control number, (d) the link points the user to a banker download (disguised as a "continue" for the updating process) and, finally, the message ends with (e) the bank disclaimer and phone numbers.

After downloaded and executed, the banker actually steals the user's information. This can happen in two ways, online or offline. In an online stealing, the attacker subverts the hardware token usage by distracting the user during the slice of time in which the token value is valid (about 3 minutes). After asking for the user to input the credentials required for the bank site access, the banker sends this information to the attacker (through HTTP methods, e-mail,



From: **Itaú Unibanco SA** < ITAUMAIL@itau-unibanco.com.br >
Date: April 27, 2012 11:34
Subject: Security Relations - Information: Mr (a) Customer [redacted]
To: [redacted] @ gmail.com

Dear [redacted]

Are you getting Sync communication to be held, sent by **Itaú Unibanco SA - Security Relations.**

Sender Comments:

We ask that be done urgently synchronization security.

Continue

Deliver to: [redacted] **@ gmail.com**

Control Number: 1005.2012-002

O Banco Itaú garante o sigilo dos seus dados. Acesse o site do Banco Itaú e conheça a nossa política de privacidade. Por favor, não retorne este e-mail.

Em caso de dúvidas, reclamações ou sugestões, consulte a sua agência. Se for necessário, utilize o SAC Itaú, todos os dias, 24 horas, pelo telefone 0800 728 0728. Ou então acesse o Fale Conosco: www.itau.com.br. Se não ficar satisfeito com a solução apresentada, utilize a Ouvidoria Corporativa Itaú, pelo telefone: 0800 570 0011 (dias úteis, das 9 às 18 horas), ou escreva para Caixa Postal nº 67 600 - CEP 03162-971. Deficiente auditivo pode ligar para 0800 722 1722, todos os dias, 24 horas.

Esta mensagem eletrônica é confidencial. Sua utilização, cópia, distribuição ou divulgação são expressamente proibidas, sob pena de responsabilização civil e criminal.

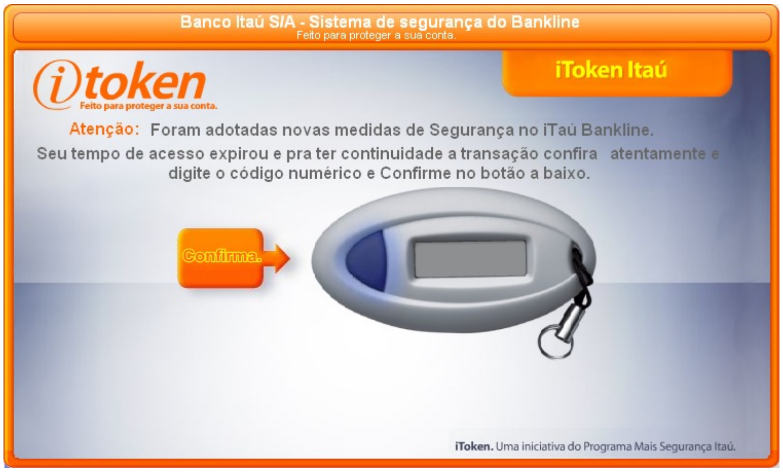
Figure 5.2: A successful phishing message.

remote database or FTP server), shows a screen requesting the code that is currently on the hardware token display (Figure 5.3) and then usually shows a dynamic bar that mimics an update (Figure 5.4), ensuring that the user waits for the completion of the "updating" process. Meanwhile, the attacker uses the stolen credentials and valid token code to perform online transactions.

This attack is difficult to perform as it demands that the attacker be online as the user is performing a financial operation. In addition, it is not scalable, as one attacker is not able to perform more than one attack of this type at a time.

A similar way to steal online bank credentials is to persuade the user into inserting the entire password table. Thus, the attacker receives all required information in an unmanned way and thus can later perform a variety of financial operations without a time constraint. The *modus operandi* is the same as above but, after asking for the users bank site credentials, the banker shows a screen requiring the values from a password table. Some bankers are even better developed than others and check if the user is not inserting repeated values among the positions. An example illustrating different banker screens to steal the entire password table is depicted in Figure 5.5.

Other ways to inadvertently redirect the user to a cloned banking site may involve either the modification of the system "hosts" file, or the loading of a PAC (proxy auto-config) file. The



Banco Itaú S/A - Sistema de segurança do Bankline
Feito para proteger a sua conta.

itoken
Feito para proteger a sua conta.

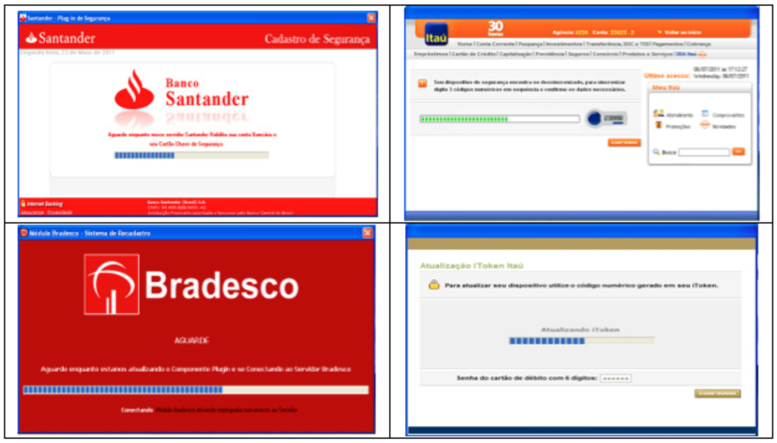
iToken Itaú

Atenção: Foram adotadas novas medidas de Segurança no iTaú Bankline.
Seu tempo de acesso expirou e pra ter continuidade a transação confira atentamente e digite o código numérico e Confirme no botão a baixo.

Confirma.

iToken. Uma iniciativa do Programa Mais Segurança Itaú.

Figure 5.3: Banker screen to steal token value.



Aguarde enquanto estamos atualizando o Componente Plugin e se Conectando ao Servidor Bradesco

Atualização iToken Itaú

Para atualizar seu dispositivo utilize o código numérico gerado em seu iToken.

Atualizando iToken

Senha do cartão de débito com 6 dígitos:

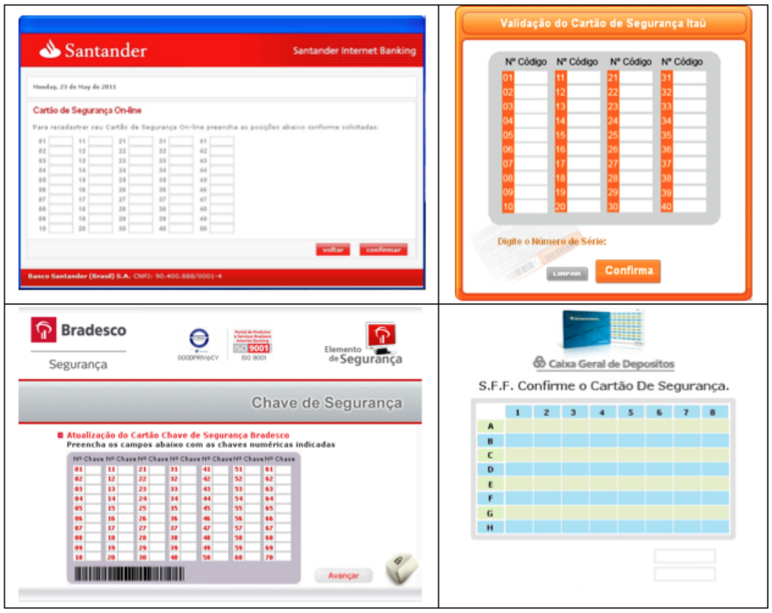
Figure 5.4: Distraction screens to allow an attacker to steal online banks while the user waits.

"hosts" file maps hostnames to IP addresses, being the starting point to resolve a hostname query on the victim's system. Malware can modify this file to map known Internet banking URLs to IP addresses that are hosting cloned sites containing forms to steal users' credentials. The other method forces the browser to load a PAC file, which defines the proxy that a Web browser should use to access a given URL through the JavaScript function `FindProxyForURL(url, host)`. A PAC file created on the victim's system provides IP addresses that might lead to cloned sites, thus allowing us to identify owned Internet servers.

Based on the observation of the bankers' common behavior, we defined a (non-exhaustive) set of potentially dangerous activities that may be used to identify them. These activities are briefly described below:

- Stealing of user credentials or financial information (**CS**), and of system or user data (**IS**), such as username, machine name, hard disk serial number, O.S. version etc.

- E-mail sending (**ES**) from the infected system, to send out sensitive data or to deliver spam to extend the attack by further propagation.



Monday, 23 de May de 2011

Cartão de Segurança On-line

Para recadastrar seu Cartão de Segurança On-line preencha as posições abaixo conforme solicitadas.

Banco Santander (Brasil) S.A. CNPJ: 90.400.888/0001-4

Atualização do Cartão Chave de Segurança Bradesco

Preencha os campos abaixo com as chaves numéricas indicadas

Digite o Número de Série:

S.F.F. Confirme o Cartão De Segurança.

Figure 5.5: Banker screen to steal all values from the user's password table.

- Subversion of Internet browsing through the modification of the name resolution file (hosts) (**HC**) or the browser proxy using PAC files (**PL**).

- Presence of images related to Internet banking sites, as banks do not send executable files to their clients (at least in Brazil), as well as an autonomous program not being supposed to have embedded bank images.

Hence, in addition to the already existing aforementioned behaviors (CS, IS, ES, HC, PL), we include the Internet Banking forgery (**BF**) due to the presence of this type of image during the execution or within the binary file.

5.3.4 Other Works about Bankers

The financial motivation behind malware infections justifies (from the standpoint of a cybercriminal) the increasing spread of bankers. Next, we present some research works regarding banker detection.

In [51], the author discusses the behavior of bankers targeting German, Swedish and Brazilian Internet banks, and the mechanisms employed by banks to protect their customer accounts, such as one-time-passwords (OTP) and transaction numbers (TAN). He proposes a tool called "Mstrings", which searches the memory of an analysis system (during a malware sample's execution) for known Internet banking-related strings. Although this scheme is effective against bankers that act as keyloggers when the user is accessing the Internet bank site, it may be ineffective against bankers that rely on social engineering and that do not depend on the user access to the Internet bank site.

Buescher et al. [27] present a banker detection scheme that checks for hooks on Internet Explorer to steal user's information. To do so, the authors developed BankSafe, a component that verifies for API changes during Internet Explorer execution. After a malware sample runs in a controlled environment, BankSafe checks for hooking fingerprints to detect if some type of hook was installed on Internet Explorer during the analysis time. The authors claim that BankSafe produces very good detection results, however it is limited to detecting bankers that install hooks.

In [98], the authors present an approach to detect phishing Web pages, which is based on a visual detection scheme that lets the user know beforehand that his/her data is being intercepted by an attacker. Phishing detection is based on three features extracted from the analyzed Web pages: the text portion, the images and the visual appearance of the page when it is rendered in the browser. To detect the phishing attempt, they need a database of legitimate Web pages to be compared to suspicious pages. To avoid overloading the browser due to the heavy comparisons, the authors propose to work together with other phishing detection tools (AntiPhish and DOM AntiPhish), which do not compare Web page contents. Those tools verify the user's input data and the domain where it is used to alert if the same data serve as input for two different domains. The combination of approaches allows for a more accurate phishing detection, less prone to false positives. Their limitation is that they only detect phishing attempts that are based on fake Web pages during a user's browsing session, missing bankers that forge an Internet bank site through windows from their own executable file (such as Brazilian ones).

Botzilla [126] uses a network signature detection scheme that is closely related to part of our work, since we also perform our network traffic detection step based on invariant content patterns. Botzilla's goal is to detect malware "phoning home", i.e. contacting an attacker controlled site to send information. Botzilla's signatures address some malware classes whose samples were collected from a large university network, yielding a detection rate of about 94.5%.

In [68], Holz et al. analyze the underground economy of stolen credentials and the phenomena of keyloggers that communicate with criminals through dropzones, i.e. publicly writable directories on attacker-owned Internet servers, which serve as exchange points for the keylogger's collected data. Their analysis is focused on Limbo (a Browser Helper Object for Internet Explorer) and ZeuS (a keylogger attached to spam) samples. To automate the keylogger sample analysis, they developed *SimUser*, a tool based on *AutoIt* that simulates arbitrary user behavior—keystrokes, mouse movement, manipulation of windows—according to predefined templates.

5.4 System Overview

In this section we present BanDIT, the Banker Detection and Infection Tracking system that we have developed to identify bankers and their resulting compromised assets (IP addresses, e-mail accounts).

5.4.1 Architecture

We developed BanDIT on the top of our own dynamic analysis system, BehEMOT (see Chapter 4), which monitors the execution of a malware sample and the processes it spawns (or writes into the memory of). Inside the monitoring environment, we also developed a component based on *AutoIt*[2] to interact with screens and error popups, and to take screenshots. Moreover, we capture the network traffic off the analysis environment, so that payloads are reliably obtained. We also use *Foremost*[3] to extract the images or screens that are embedded in a banker binary file. An overview of BanDIT is shown in Figure 5.6.

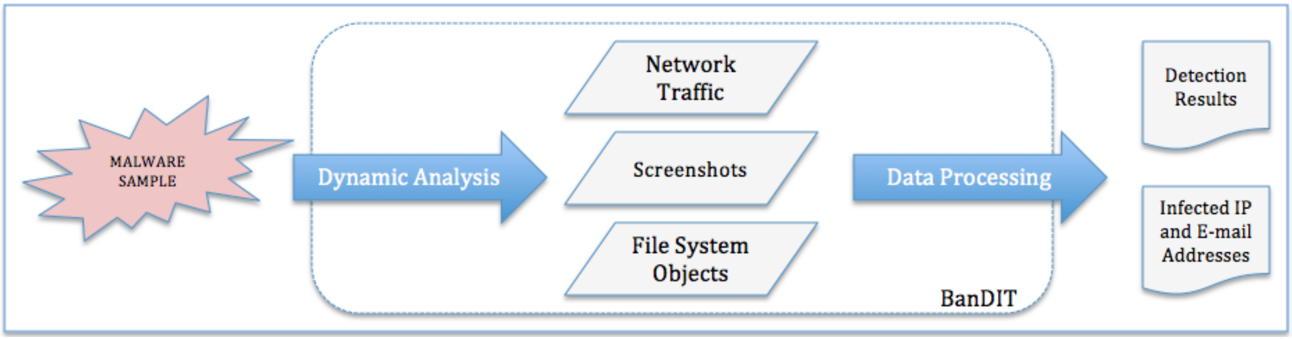


Figure 5.6: Overview of BanDIT.

BanDIT's identification process relies on three steps, which are explained as follows:

Visual Similarity. In this step, we perform the recognition of known Internet banking logos, capitalizing on the fact that attackers are restricted on the number of variations in the bank site patterns, or else phishing is bound to fail. Thus, it is important that an attacker keeps the logo positions and colors to successfully lure the user. To accomplish the visual identification, we obtain figures from the banker binary file, from network connections that download images (even from the legitimate Internet banking site), and from windows presented by the banker during its execution. Gathering images from these sources increases our chances to obtain them if the banker avoids screenshots, ciphers the traffic or its binary is specially packed. To verify if a malware extracted image contains bank logos, we use JavaCV[4], an interface to the OpenCV[5] library that contains a class to search for an object inside an image using the Speed-Up Robust Features algorithm [14]. To do so, we developed a component that searches for the logos of the five major Brazilian banks within the extracted images. Moreover, we submit all images to Tesseract[6], an Optical Character Recognition engine. We then search the resulting text for keywords present in our specially crafted banker-related terms dictionary.

Network Traffic. We have been observing that, within our samples, bankers use to send stolen data from the victim's system through HTTP requests. The analysis of these requests revealed some patterns that allowed us to develop "network signatures" to verify for banker-related activities in the network traffic that is captured during the dynamic malware analysis. Listing 5.1 exemplifies the contents of a POST request related to a banker infection. We verified that some bankers send this type of request as soon as they are executed, followed by a similar one containing Internet banking information provided by a victim.

[2] http://www.autoitscript.com
[3] http://foremost.sourceforge.net/
[4] http://code.google.com/p/javacv/
[5] http://opencv.org/
[6] http://code.google.com/p/tesseract-ocr/

Listing 5.1: A banker POST request.

```
praquem=XXXXXXX@gmail.com
&titulo=NOME_DA_MAQUINA
Vitima da Key Revoltado
&texto=
------------------------------------------------
infectizinho
------------------------------------------------
Computador Nome..: NOME_DA_MAQUINA
Mac..: XX-XX-XX-XX-XX-XX
Numero Serie HD..: xxxxxxxx
------------------------------------------------
Data..: 4/24/2012
Hora..: 8:35:34 AM
------------------------------------------------
```

The network traffic analysis step also allows us to obtain PAC files, images, URLs, e-mail and IP addresses of abused servers, which are used in our identification process and reported to competent authorities.

File System. To detect changes that bankers commonly perform in infected systems, such as the inclusion of PAC files, modifications to the “hosts” file, and attempts to disable security mechanisms (AVs, firewall, automatic updates), we monitor malware execution using our SSDT hooking driver. We are able to obtain PAC files if bankers drop them on the victim’s system or by downloading them from URLs that are set as values in the registry `Internet Settings\AutoConfigURL` key. To evaluate the obtained PAC files and to check for redirection of bank site URLs, we execute the PAC code using a JavaScript processor based on Rhino [104] and call the produced *FindProxyForURL* function using Internet banking domains as parameters. Thus, even if the JavaScript code found inside the PAC file is obfuscated, it will eventually deobfuscate itself and *FindProxyForURL* will be called. The result of the analysis of the PAC presented in Listings 5.2 and 5.3 is shown in Listing 5.4. Furthermore, we obtain the “hosts” file from the infected system after the dynamic analysis is done, which is then searched for Internet banking domains.

Listing 5.2: Example of an obfuscated PAC.

```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!’’.replace
    (/^/,String)){while(c−−){d[c]=k[c]||c}k=[function(e){return d[e
    ]}];e=function(){return’\\w+’};c=1}; while(c−−){if(k[c]) {p=p.
    replace(new RegExp(’\\b’+e(c)+’\\b’,’g’),k[c])}}return p}(’25
    17(14,11){7 4=\’4\’,1=\’1\’,2=\’2\’,5=\’5\’,3=\’3\’;7 8=16
    15(”*.”+2+”12.*”,1+”10.*”, ”*.”+1+”10.*”, ”*.”+4+2+1+5+”.*”,
    4+2+1+5+”.*”, 1+1+”.*”, ”*.”+1+1+”.*”, 2+”12.*”, 2+”13”+3+”.*”,
    ”*.”+2+”13”+3+”.*” );29(7 6=0;6<8.26;6++){27(28(11,8[6])){9”24
    23.19.20.21:22”}}9”18”}’,10,30,’|b|s|l|h|c|pa1s|var|naxz|return|
    radesco|host|antander|antanderempresaria|url|Array|new|
    FindProxyForURL|DIRECT|254|63|133|80|65|PROXY|function|length|if|
    shExpMatch|for’.split(’|’),0,{}))
```

Listing 5.3: Example of a deobfuscated PAC.

```
function FindProxyForURL(url,host){
        var h=’h’,b=’b’,s=’s’,c=’c’,l=’l’;
        var naxz=new Array(”*.”+s+”antander.*”,b+”radesco.*”,”*.”+b
           +”radesco.*”,”*.”+h+s+b+c+”.*”,h+s+b+c+”.*”,b+b+”.*”,”*.”+
           b+b+”.*”,s+”antander.*”,s+”antanderempresaria”+l
           +”.*”,”*.”+s+”antanderempresaria”+l+”.*”);
        for(var pa1s=0;pa1s<naxz.length;pa1s++){
                if(shExpMatch(host,naxz[pa1s])){
                        return ”PROXY 65.254.63.133:80”
                } } return ”DIRECT” }
```

Listing 5.4: Results of the analysis of the PAC in the previous example.

```
bb.com.br -> 65.254.63.133:80
bradesco.com.br -> 65.254.63.133:80
hsbc.com.br -> 65.254.63.133:80
santander.com.br -> 65.254.63.133:80
santanderempresarial.com.br -> 65.254.63.133:80
```

5.4.2 Data Extraction and Dynamic Execution

The previous process of obtaining logos, creating a banker-related terms dictionary, and developing network/file system signatures involved the manual analysis of 1,194 malware samples obtained from phishing (and analyzed) between August, 2010 and July, 2011.

For our current tests, we selected 1,653 malware samples mainly from phishing messages and partners/contributors, all of them obtained in 2012[7]. We submitted these samples to BanDIT, which runs on a virtualized Windows XP with Service Pack 3 for four minutes. This process includes the external capture of network traffic and the extraction of images from the binary file using Foremost. Moreover, we scanned all samples with the Avira antivirus engine to obtain their assigned labels. We normalized these labels to focus on the type/family of the malware sample. Table 5.1 shows the distribution of samples among malware families.

5.5 Empirical Evaluation

From Table 5.1, $\approx$ 23% of the 1,653 samples are not detected as malware by the antivirus engine (ID = 15), and $\approx$ 10% of the samples (ID = 16) are distributed among 63 distinct families. The remaining of the samples is divided into the 14 families with highest incidence. Below, we discuss the behaviors found in the samples, map their families and leverage BanDIT identification results.

[7] These 1,653 malware samples belong to the full dataset, which was used to the evaluation of our taxonomy in the previous chapter.

Table 5.1: Distribution of malware samples among Avira-assigned families.

AV ID	AV Label	Total
01	Agent	2.2%
02	Atraps	6.4%
03	Banker	17.7%
04	Crypt	9.1%
05	Delf	2.7%
06	Delphi	5.8%
07	Downloader	1.1%
08	Gen	3.3%
09	Graftor	2.5%
10	Offend	3.0%
11	Spy	5.4%
12	Vb	4.4%
13	Virtool	1.0%
14	Zusy	2.0%
15	NOT DETECTED	23.2%
16	Other families	10.2%

5.5.1 Observed Behavior

Before testing BanDIT, we manually analyzed our samples and searched for the set of behaviors defined in Section 5.3.3. We found 1,520 samples that behaved like bankers. Figure 5.7 shows the amount of samples that presented each behavior. As also observed in Chapter 4, information stealing is the most frequent behavior found in our reduced dataset, followed by PAC loading, e-mail sending and the presence of bank-related images.

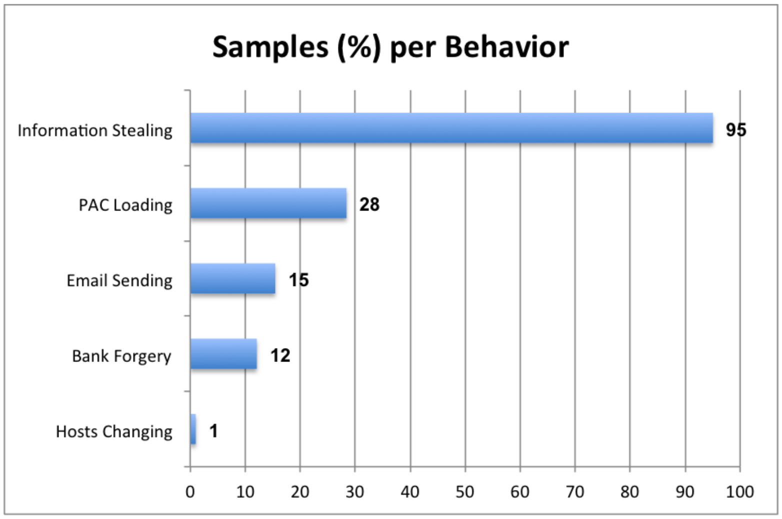


Figure 5.7: Percentage of banker samples per presented behavior.

Table 5.2 associates the malware family IDs to each observed behavior[8]. One can see that

[8] IS = Information Stealing (Credential + System/User); ES = Email Sending; HC = Hosts Changing; PL = PAC Loading; BF = Bank Forgery (Internet banking images).

samples labeled as "bankers" presented all of the selected behaviors: stealing credentials and system data, sending e-mail messages, loading PAC files, interfering with the "hosts" file and carrying bank logos, as expected. However it is worth mentioning that, except for "Hosts Changing" and "Bank Forgery", all other banker-related behaviors were presented by all AV-assigned families, making the AV classification rather dull. Interestingly, bank-related images found in samples do not correlate to the specific "banker" AV label.

Table 5.2: Behaviors presented (columns) by malware families (rows) under manual inspection.

	IS	**ES**	**HC**	**PL**	**BF**
01	✓	✓		✓	✓
02	✓	✓		✓	✓
03	✓	✓	✓	✓	✓
04	✓	✓	✓	✓	✓
05	✓	✓		✓	✓
06	✓	✓		✓	✓
07	✓	✓		✓	
08	✓	✓		✓	✓
09	✓	✓		✓	✓
10	✓	✓		✓	✓
11	✓	✓		✓	✓
12	✓	✓		✓	✓
13	✓	✓		✓	
14	✓	✓		✓	✓
15	✓	✓		✓	✓
16	✓	✓	✓	✓	✓

5.5.2 BanDIT Results

To conclude our study about banker behavior, we evaluate BanDIT's results and compare our findings to the manual analysis previously provided. The main advantage of BanDIT is to aggregate different techniques into one component so as to identify banker-related behavior. Figure 5.8 shows how many of our 1,520 manually labeled samples (as bankers) were identified by each of the BanDIT's techniques. BanDIT was able to identify banker-related behavior in 1,502 (98.8%) samples. Network pattern matching was the most effective, but filesystem modification monitoring and image analysis were also important, since that both (combined) identified 539 (35.5%) samples.

5.5.3 Infection Tracking

During BanDIT's evaluation, we were able to identify 88 e-mail addresses that were compromised or specifically created to receive stolen data from banker's infected victims. Furthermore, we pinpointed 183 IP addresses of attacker-owned/compromised servers that have been used to host malware pieces, such as PAC files, or to act as fake bank sites. For the purpose of information leverage, we deployed a Web site (inspired on `https://zeustracker.abuse.ch` and

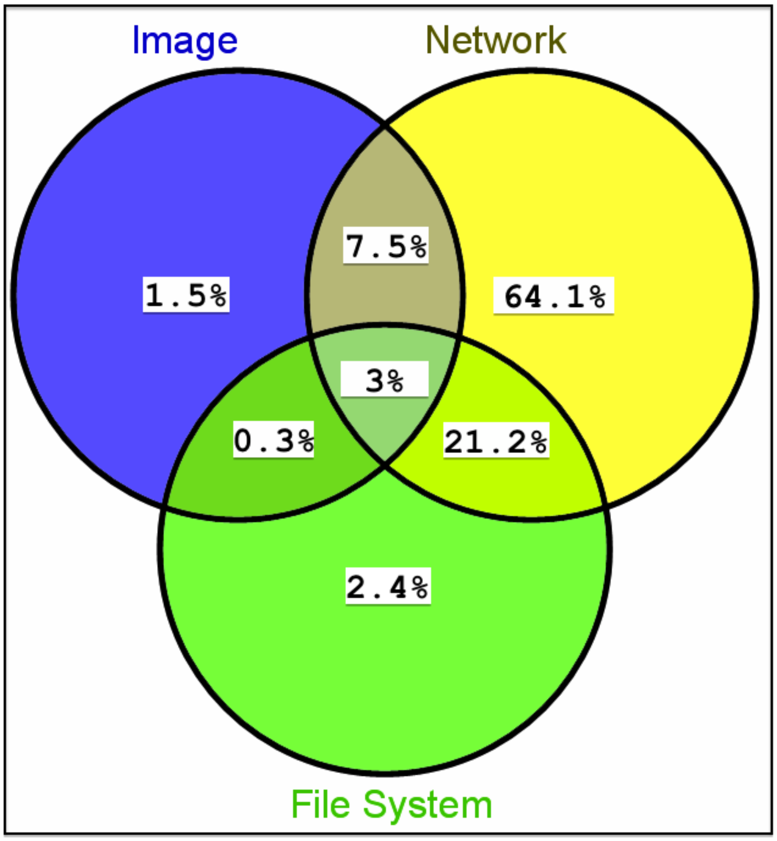


Figure 5.8: BanDIT's identification results

`https://spyeyetracker.abuse.ch`) based on Google Maps API [61] that tracks IP addresses contacted by bankers and associates them with the type of infection (through a hosts or PAC file, or by evading information via POST method). A screenshot of the Web site's main page is in Figure 5.9, where each marker provides geographical information about the compromised IP addresses and the infection type.

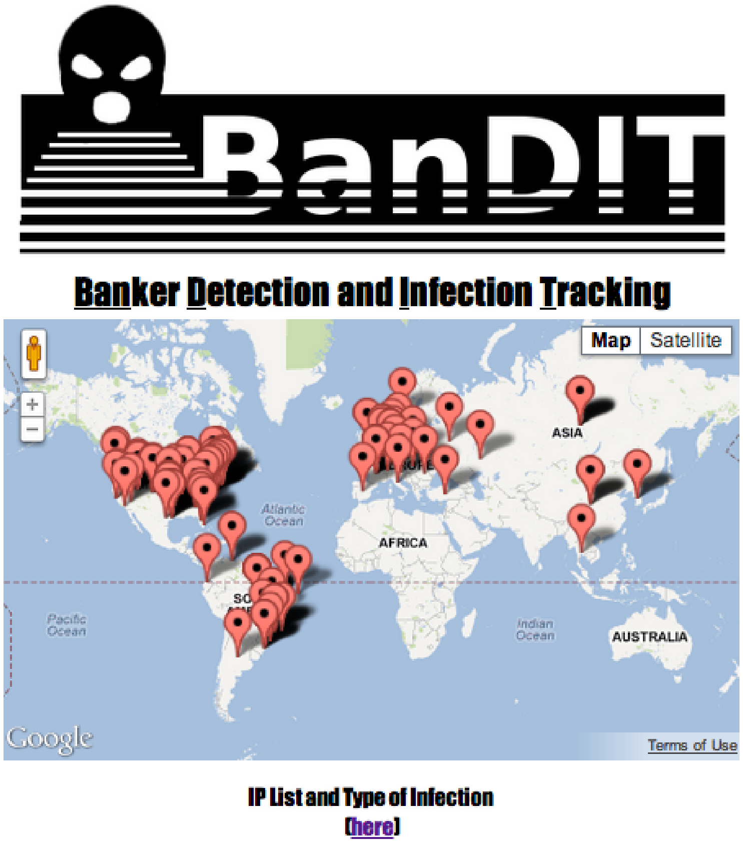


Figure 5.9: Screenshot of BanDIT's Web site.

We found that 77.1% of the samples that evaded information through the network ended up contacting IP addresses from Brazil, United States and Germany. Additionally, 80.7% of the IP addresses used in PAC files and "hosts" files are from United States, Brazil and France. The e-mail accounts used by the bankers were notified to the corresponding e-mail providers and the IP addresses were notified to the competent authorities.

5.5.4 Discussion

In our analysis, we considered that banks do not send executable files through e-mail to their clients, as major Brazilian banks claim on their Internet banking sites. The comparison of our manually obtained results to those produced by BanDIT shows that our system identified information stealing patterns on 94.6% of the samples, which stands pretty well to the 95.1% obtained by the manual process. This difference may be due to user interactions that might be required by the sample but were not done during the BanDIT analysis. Regarding the visual similarity searching, BanDIT found Internet banking related images on 12.1% of samples, the same amount that we found manually. This shows that the combination of logo searching and keyword matching in the text obtained by an OCR was very effective. BanDIT's filesystem monitoring led to the identification of changes in the "hosts" file or loading of PAC files in 26.6% of our samples; in contrast, manual inspection identified this type of behavior in 29.1% of the samples (PL + HC). In this case, the difference may be caused by problems during the sample execution, such as PAC file unavailable, analysis timeout or the need for human interaction. Finally, our manual search for e-mail sending allowed us to find compromised servers and e-mail addresses.

Although we chose not to address ZeuS and SpyEye samples, since there already are related work addressing them (e.g., [27], [57]), we also tested the `deBank` tool[9]. It is intended to detect banker variants from some known families, such as SpyEye and ZeuS through matching of certain patterns found in the infected system's memory. We executed, one at a time, around 70 SpyEye and ZeuS samples obtained from [2] and [3], and then launched the tool. Unfortunately, none of them was detected by `deBank`, maybe because they are newer than the last available version of the tool. This test was interesting as it corroborated the importance of screening out banker samples for further analysis, since traditional detection tools can be easily bypassed. We conclude that behavior-based approaches, such as BanDIT, may be useful to help develop counter-measures, although they are not intended for real-time detection. In addition, BanDIT identification components can be easily linked to available dynamic analysis systems.

5.6 Limitations and Future Work

As happens with every detection scheme, once the signature generation process is exposed, attackers are prone to change their malware. However, security researchers live an arms race against cybercriminals. Hence, we must remain alert for new malicious trends and techniques in order to develop new defensive countermeasures. Every signature-based scheme is prone to be bypassed once attackers find out the signature generation process. This causes an arms race between security people and cybercriminals, requiring this type of system to be constantly updated. Another limitation that our system suffers is related to evasion techniques that affect all dynamic malware analysis systems, (e.g., waiting for a certain number of reboots before presenting the malicious behavior). However, since we are able to extract images from the sample file, our visual similarity technique may identify a banker right away. Due to our observations that most Brazilian bankers send the stolen information unencrypted, we currently do not handle encrypted network traffic. Thus, the current implementation would not be able to identify stolen data sent by an encrypted channel.

However, the detection steps based on images and file system modifications could do this job. Furthermore, we can improve the system to monitor the network traffic before it is encrypted, solving this issue. The image recognition process may fail when malware employ overlays to present the bank images, because the traditional screenshot capture the contents presented in the overlay. But, to evade all image gathering techniques employed by our system, the malware

[9] `http://www.fitsec.com/tools/DeBank.exe`

also have to use encryption when obtaining the images from the network and keep it obfuscated when it is stored inside the malware file. Also, rootkit-based malware could potentially subvert our filesystem monitoring component, however only two samples of our dataset loaded drivers and one of them was identified by BanDIT as a banker.

Therefore, the use of more than one detection method is important because it allows the system to identify a malicious behavior even when one or two of the employed methods fail. Future works include making the infection tracking system available to the public, addressing encrypted network traffic and developing dissectors to extend our network protocol coverage. Finally, we are working on the relationship of samples to malware development kits so as to provide information to law enforcement agencies.

5.7 Concluding Remarks

As more sophisticated bankers appear, the need for counter-measures and fast response increases. Thus, it is important to identify and extract relevant information about bankers so that they can be screened out during dynamic analysis for further and deeper investigation. In this chapter, we introduced BanDIT, a system that dynamically analyzes malware in order to identify bankers. BanDIT's identification process combines visual similarity searching, network traffic pattern matching and filesystem modification monitoring. BanDIT's evaluation shows that it is effective to screen bankers out (98.8% of correct identifications) of a dataset of general malware. Moreover, BanDIT helps incident responders to warn victims in a timely manner, to notify infected ISPs and servers and to create blacklists that can be quickly deployed to avoid more banker infections to take place. Finally, we showed that our behavior-centric taxonomy can be extended to address specific types of malware and to help in detection procedures.

Chapter 6

Malware Visualization

6.1 Introduction

Another possible use for malware behavioral profiles is to serve as input to visualization tools. Visual analysis of malware behavior can facilitate the work of incident responders, as well as help in the search for unusual patterns. In this chapter, we show how visualization can improve the analysis of malware-related incidents. To this end, we developed two tools that handle and present samples' extracted execution behavior, the behavioral spiral and the malicious timeline. These tools take advantage of our dynamic malware analysis system, BehEMOT, to generate interactive and visual environments regarding a malware sample execution. A great advantage of visually analysing malware behavior is the possibility to verify the "similarity" among families, as well as the separability among them.

6.2 Interactive, Visual-Aided Tools to Analyze Malware Behavior

We saw previously that malware samples usually disseminate through the Internet and can cause incidents with severe damage to confidentiality, integrity or availability of systems and data. Most malware are not target-oriented, attacking as many systems as they can, so that an attacker can gain control and use of the victim's resources or steal sensitive data. However, there are cases in which malware have specific targets and are thoroughly designed to delude the victim, talking the unsuspecting user into supplying confidential information, as it happens in attacks directed to a government infrastructure. In both situations, severe incidents caused by malware can disrupt an entire network of systems by disseminating through headquarters and then to branch offices or can also cause irreparable damage by exposing confidential information.

When attacks succeed in breaking into a computer system, forensic procedures can be followed in order to find out:

- How the attack was perpetrated;
- Where the points of collection of information are (downloading of tools and sending of data);

- What happened during the attack to the system.

In the case of malware attacks, it is important to collect the binary that infected the system or was downloaded after the target system was compromised. This binary may then provide some clues leading to a deeper understanding of the techniques used by the attacker and the purpose of the attack. This can be done by running this malware in a controlled environment and monitoring all the filesystem and network activities to compose a specific behavioral trace, as we have been doing for this thesis results.

Malicious behavioral traces are in essence a log of the events performed by a malware on a compromised system, but this amounts to large chunks of data. Such logs are difficult to analyze as we have to stress out interesting segments of behavior (main malicious actions) while simultaneously having to obtain a general overview of the extension of the damage. However, the information obtained from this analysis is paramount to provide adequate incident response and mitigation procedures. In those cases of massive amounts of textual data to analyze, we can apply visualization techniques that can greatly enhance the analysis of logs and allow us to quickly spot important actions performed by a specific malware and to better understand the chain of malicious events that led to the compromise of the target system.

The main contributions of this section are:

- We developed two visualization tools—the behavioral spiral and the malicious timeline—to aid security analysts to observe the behavior that a malicious software presents during an attack. Those tools are interactive and they allow a user to walk through the behavior while performing zoom, rotate, and gathering of detailed information for each malware action.

- We discuss visual classification of malware families and show that our tool can be used to visually identify an unknown malware sample based on its comparison to previously known malware, and that is a step towards a visual dictionary of malicious code.

- We distribute online a beta version of our prototype, so that the community can benefit from it and use it freely.

6.2.1 Related Work

There are several research works that use visualization tools to overcome the plenty of data provided by textual logs related to security. Some of them are not open to the public, others are neither intuitive to use nor interactive, and there are those whose results are more difficult to be visually interpreted by security analysts than if they search manually through the log files.

Quist and Liebrock [120] applied visualization techniques to understand the behavior of compiled executables. Their VERA (Visualization of Executables for Reversing and Analysis) framework helps the analysts to have a better understanding of the execution flow of the executable, making the reverse engineering process faster.

Conti et al. [36] developed a system that helps the context-independent analysis of binary and data files, providing a quick view of the big picture context and internal structure of files through visualization. In a forensic context it is especially helpful when analyzing files with undocumented formats and searching for hidden text messages in binary files.

Trinius et al. [157] used visualization to enhance the comprehension of malicious software behavior. They used treemaps and thread graphs to display the actions of the executable and to help the analyst identify and classify malicious behavior. While their thread graphs can confuse a human analyst with lots of overlapped information and lack of interactivity, our timeline (Section 6.2.3) allows a walk-through over the chain of events performed by different processes created and related to the execution of a certain malware sample, as well as the magnification of interesting regions, information gathering about selected actions and annotation. Furthermore, our behavioral spiral represents temporal action, whereas their proposed treemaps consist of the actions' distribution frequency. Again, there is a lack of interactivity and excessive data, as we handle only actions that can cause changes on the target system. However, similarly to our work, it is not possible to visually classify every malware family, as variant samples can pertain to a class while presenting completely different behavior regarding the order or nature of the performed actions.

Finally, reviewing logs from intrusion detection systems is an important task to identify network attacks and understand these attacks after they happened. There are several tools that use visualization for this purpose; each one has its own approach and is better than others at specific situations. To take advantage of those tools the DEViSE (Data Exchange for Visualizing Security Events) framework [124] provides the analysts a way to pass data through different tools, obtaining a better understanding of the data by aggregating more extracted information.

6.2.2 Data Gathering

To visualize malware behavioral data, we first need to collect malware samples that have been currently seen in the wild and then analyze them to extract the actions they would perform in an attack to a target system. In the sections below, we discuss our approaches to malware collection and behavior extraction.

Malware Collection

To collect malware samples that spread through the Internet, we use our automated collection architecture [64], which uses mixed honeypots technology (low and medium interaction) [119] to capture malicious binaries for MS Windows systems. Honeypots are systems that are deployed to be compromised so as to lure attackers to reveal their methods and tools by the compromise of a highly controlled environment. The collection architecture consists of a `Honeyd` [118] node to forward attacks against certain vulnerable ports to a `Dionaea` [153] system, which emulates vulnerable MS Windows services in those forwarded ports and actually downloads the malware sample. During 2010 we captured more than 400 unique samples, which are used as our test dataset in this chapter.

Behavior Extraction

To extract the behavior of a malicious software, we run it in a controlled environment and monitor the actions it performs during the execution in the target system. Those actions are based on the system calls executed that are relevant to security, i.e. if they modify the status of the system or access sensitive information, such as file writing, process creation, changes in registry values or keys, network connections, mutex creation and so on. The dynamic analysis environment used for behavior extraction is BehEMOT [63], our system that was presented on Chapter 4. It produces logs in which each line means one action performed by the monitored malware sample or a child process and is in the form "*timestamp, source, operation, type, target*". For instance, lets suppose that a malware sample "*mw.exe*" created a process entitled "*downloader.exe*", which connects to port 80 of an Internet IP address $X.Y.W.Z$ to download a file *a.jpg* to a temporary location *TEMP*. The log file produced by BehEMOT would have the following three lines:

```
ts1, mw.exe, CREATE, PROCESS, downloader.exe
ts2, downloader.exe, CONNECT, NET, X.Y.Z.W:80
ts3, downloader.exe, WRITE, FILE, TEMP/a.jpg
```

Therefore, we use the BehEMOT behavior format to input the textual data to our visualization tools, which are described in the next section. It is worth noting that logs produced from malware sample execution can be thousands of lines long, motivating the use of visual tools to aid the process of human analysis.

6.2.3 Interactive Visualization Tools for Behavioral Analysis

The behavior of a malicious program can be interpreted as a chain of sequential actions performed on a system (as seen in the previous section), which involves operating system interactions and network connections. The analysis of those operations can provide the steps that were performed in an attack to understand the incident as a whole, as well as detailed information about what was changed in a system, such as libraries that were overwritten, infected files, downloaded data and even evaded information.

Usually, antivirus developers analyze unknown malware samples to create signatures or heuristics for detection. This is an overwhelming process and involves plenty of manual work, as a human analyst has to search for pieces of data that characterize the sample as part of an already known malware class or create a new identifier to it. Actually, this process is worse due to the increasing amount of new malware made from 'do-it-yourself' kits and variants of older ones. Sometimes, a malware is assigned to a family (and detected by an antivirus engine) based on the value of a mutex it creates, or on a specific process that it launches with a particular name, or on the kind of information it sends to the network.

Our motivation to develop visual tools is to make it easy to process and pinpoint very specialized information and then to help a human analyst to focus in the interesting actions performed by a malware sample. This way, it is possible to quickly analyze new malware by visualizing their overall behavior and expanding only those actions that an experienced analyst considers suspicious or important. Also, public available dynamic analysis systems (e.g. [72], [155], [26]) provide textual reports full of information that would be easier to be interpreted if an analyst could visually manipulate it. To fill this gap, we developed two tools that transform a textual report from a dynamic analysis system into an interactive and visual behavior, which are explained below.

Timeline and Magnifier

Malware time series events can also be visualized in simple x-y plots, where the x axis represents the time and the y axis some information about the event. The time information on the x axis can be any of the following: i) the absolute time of occurrence of an event; ii) the relative time of occurrence (counted from a particular initial value); or iii) a simple sequence that implies the order of occurrence of the events.

The event information can be plotted using several different methods, often specifically tailored to a particular purpose. The height on the y axis can be used to represent the frequency of occurrence, severity or intensity of a given event, if such information is known, or a discrete representation of event types. Decorations such as different marks for additional event characteristics can also be used to allow representation of more data dimensions than just two. Additional graphical elements may also be used, but one should always take care not to overload the plot with too much graphical content, which may confuse the user and hinder his/her ability to draw quick conclusions from the plot.

An example x-y plot used to represent malware time series events is given in Figures 6.1 and 6.2 (events selected by the tool's user to be a search pattern are underlined in light red; automatically matched events are underlined in dark red). They show the corresponding tool in action, as it parses a malicious behavior file with malware events ordered by action timestamp and plots the result using a simple, interactive interface. The tool draws the whole time series in two panels: on the top panel all points on the series are plotted, with the x axis representing the order in which events occurred and the y axis representing the action associated with an event (which are not in any particular order and can be rearranged). Also, events are plotted as dots of different colors, according to the process id that performed such actions. For example, if the malware associated process created two child processes and also required service from an already running process, we would have four different colors in the graphic: malware, first child, second child and the running process). Not all possible monitored actions have to be performed by a malware, so the y axis varies accordingly to what was present in the captured behavioral trace.

Plots created by this tool are also interactive. Since there are many events on the top section of the plot, it is hard to see exactly which one follows which one, so a region of interest (highlighted under a translucent yellow region on the plot) can be selected by the user. Selection is done by dragging the region with the mouse, which also causes the region of interest to be shown enlarged on the bottom panel of the plot. The x and y axes and plot colors follows the ones in the top section. The bottom part of the plot also conveys information about the diversity and variability of operation types in its gray background: darker backgrounds suggest a higher diversity, whereas lighter backgrounds point to higher similarity.

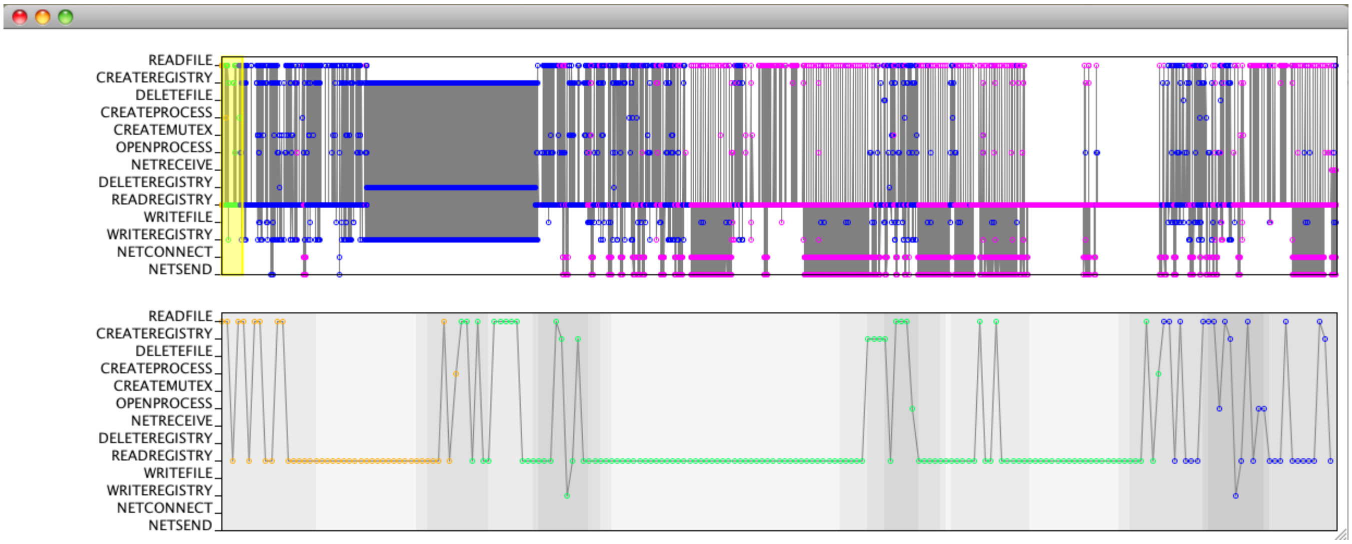


Figure 6.1: Timeline and Magnifier tool representing malicious events.

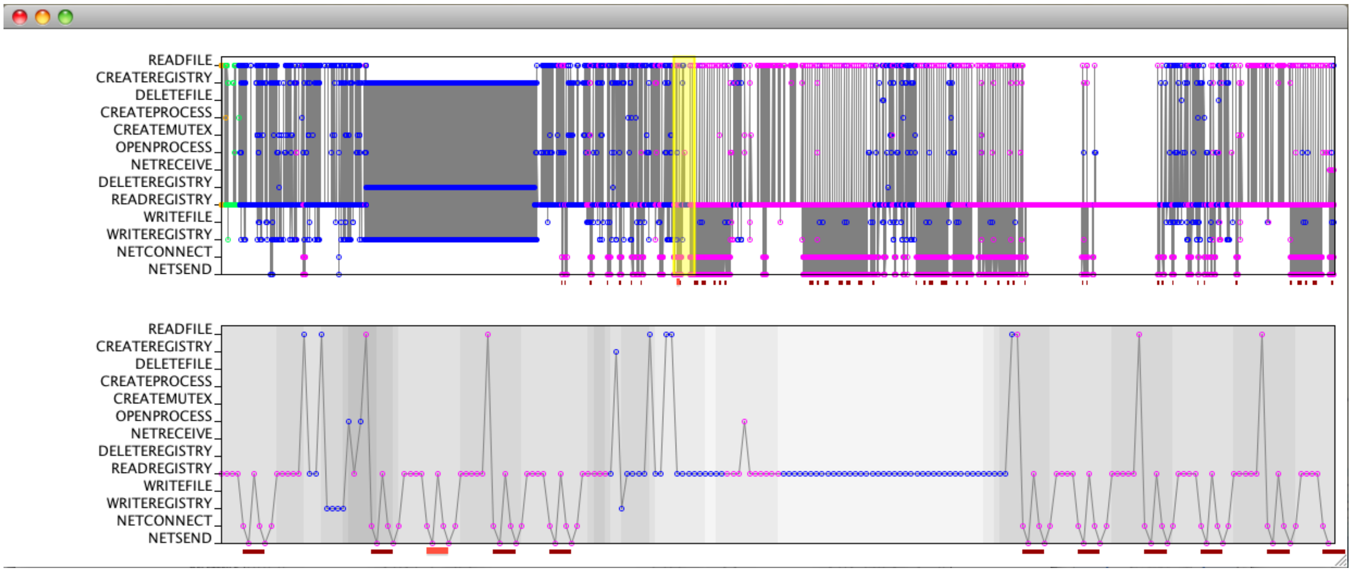


Figure 6.2: Sequence of selected events (user and automatic).

Malicious Spiral

The goal of this tool is to present an ordered sequence of all malicious actions of an attack in a spiral format, using an iconic representation. The spiral representation is useful to show the big picture of a malware sample behavior and also to allow quick visual comparisons among different malware samples even in the presence of variants. Instead of using a straight line broken in columns [50], the spiral format is less prone to confusion as small variances present in the behavior of malware from the same class usually keep the general visual appearance that models a family.

By exploiting the viewing abilities available, an investigator can zoom in and out, turn, tilt, select behavior slices, view the logged action in textual form and compare it with other behavioral data, if available. Thus, we provide not only an overview of the attack, but also the possibility of identifying certain behavioral patterns that could help in the classification of malware samples (Section 6.2.4).

The operations and types that are monitored and used to produce a behavior log are shown in Table 6.1, as well as all icons that are used to represent them. We divided the table lines in four groups of operations with similar purpose on each subsystem type, e.g., a CONNECT operation in a NET type has the same effect as that of a CREATE REGISTRY, i.e. they prepare the environment for an active operation that will represent a registry value being written or a network connection being opened that later might send data. In the same way, a READ FILE or REGISTRY, a RECEIVE NET and a QUERY MUTEX are all passive operations, and so on.

Table 6.1: Icons for monitored operations and types.

Action / Type	MUTEX	FILE	PROC	REG	NET
READ		■		■	
QUERY	■				
RECEIVE					■
WRITE		▲		▲	
SEND					▲
CONNECT					●
CREATE	●	●	●	●	
DISCONNECT					✳
DELETE		✳		✳	
TERMINATE			✳		
RELEASE	✳				

6.2.4 Tests and Analysis of Results

Although we have developed the timeline and the spiral tools based on the same kind of log format, they have different usage. The timeline and magnifier tool can be used by any user to do a "behavioral walk-through", while verifying how many processes were launched, what actions they performed and from what kind, etc. On the other hand, the spiral tool can be used to visually identify malware from the same family, while simultaneously depicting an iconic overview of the behavior that can be manipulated to show more detailed information (present in the log file). In this section, we show how the spiral tool serves as a visual dictionary and how it can help classify malware from the same family. To the extent of our tests, we extracted the execution behavior from 425 malware samples collected by the system described in Section 6.2.2 and dynamically analyzed them with the system described in Section 6.2.2.

All malware samples from our dataset are currently found "in the wild" and together constitute variants from 31 families. Scanning them with the up to date ClamAV antivirus engine [33] reveals 94 unidentified samples. By observing the generated spirals, we realized that it is possible to group some of them by their visual behavior. Also, malware samples from different families present similar behavioral patterns among samples from their own class, while at the same time keeping a dissimilarity from other classes' samples. This differentiation factor present in the visual patterns is an important indicative that clustering algorithms, artificial intelligence and data mining techniques can be applied to our logs to classify malware based on behavioral similarities.

In Figure 6.3, we chose two trojan families—Pincav and Zbot—and selected three samples of each one to be printed side-by-side. Notice that even when the malware samples from a family perform variant operations, they still keep a behavioral pattern that can be used to characterize them in the same class.

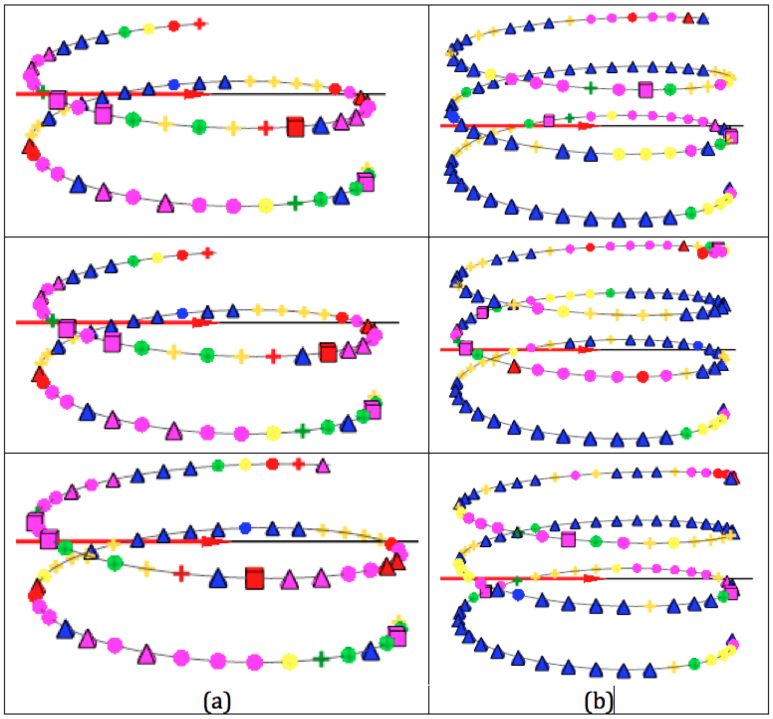


Figure 6.3: Behavioral spirals for 'Pincav' (a) and 'Zbot' (b).

Another interesting fact is that if a malware sample tried to do more network connections than another one from the same family (e.g., malicious scans) or if it performed fewer actions or crashed, it is possible to clearly notice the similarities between an incomplete behavior and a larger one, as shown in Figure 6.4, which contains three samples of the *Allaple* family ((a) a sample that could not connect to the network and stopped its activities; (b) a sample that performed a short network scan; (c) a sample performing a massive scan, where each red sphere means a connection to a different IP address).

In Figure 6.5, we depict the behavioral spirals from four different malware families—the worms Palevo and Autorun; the Trojans Buzus and FakeSSH. Notice that we can visualize the separability of the classes with minor variances among behaviors from the same family for Palevo, Autorun and FakeSSH. In the case of Buzus, one can observe that the first two behaviors significantly differ from the last ones, putting those samples apart from each other

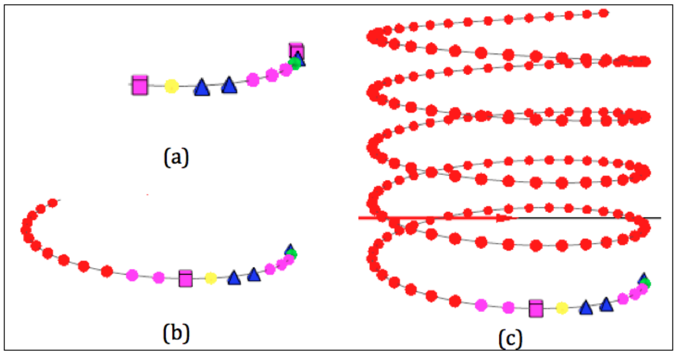


Figure 6.4: Three 'Allaple' worm variants.

is an automated classification scheme. However, in Figure 6.6, one can also realize that a sample whose AV assigned label is "UNKNOWN"—i.e. an unidentified sample—presents a visual behavior that is quite similar to a sample from the trojan family "Inject".

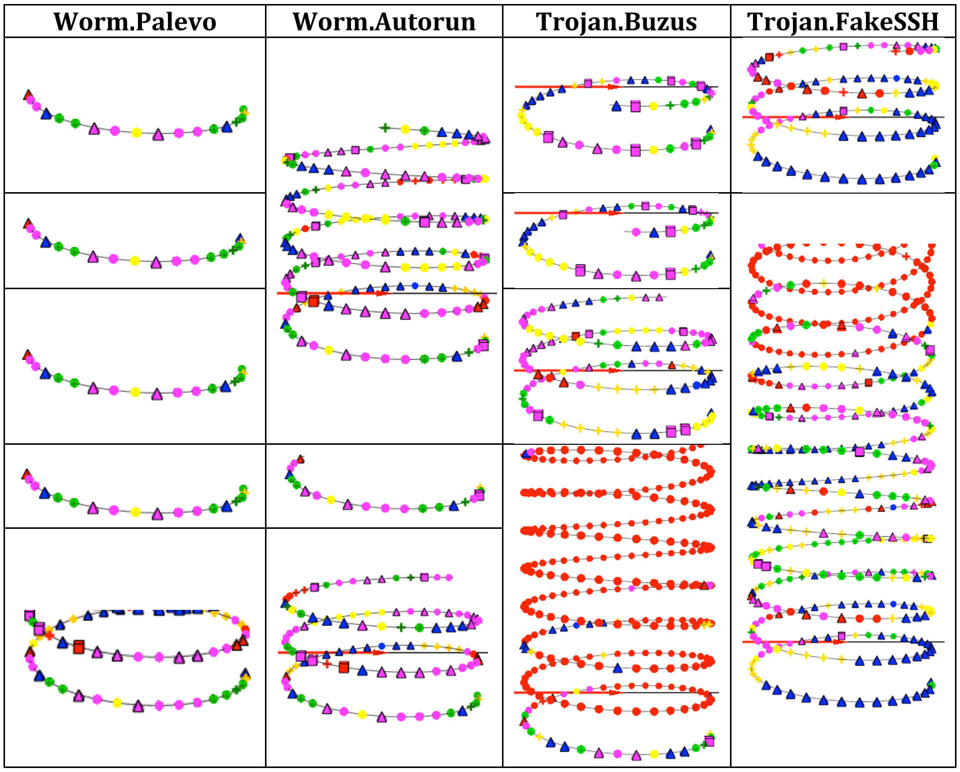


Figure 6.5: Visual behavior extracted from four malware families.

6.3 Concluding Remarks

In this chapter we proposed two interactive, visual-aided tools to increase the efficiency in malware analysis, which allow security analysts to overview malicious behaviors. Moreover, our tools allow walking through over the logs, annotation and highlighting of interesting events and actions, searching for patterns, deeper understanding of the damage performed on a target

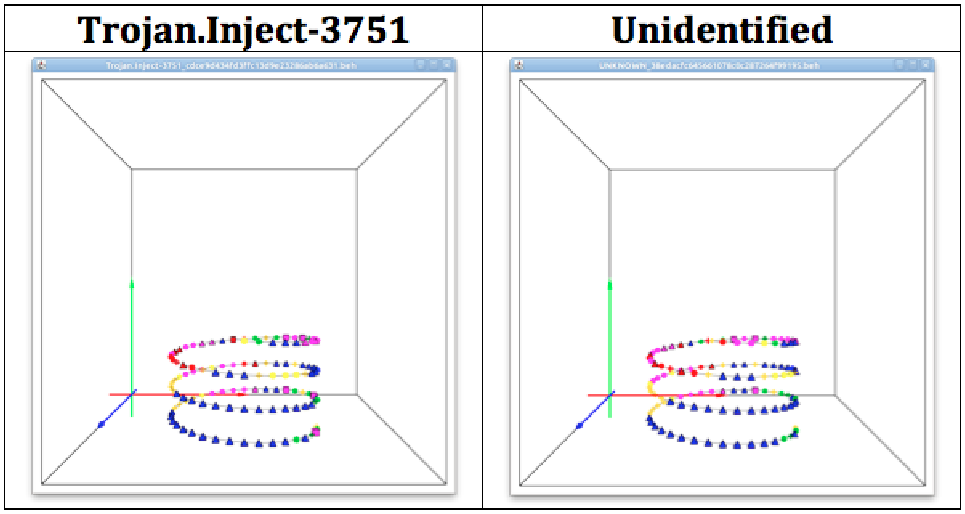


Figure 6.6: Unidentified malware (right) sample visually classified as a known threat (left).

system and visual comparison among malware samples. Hence, the possibility of visual family distinction suggests that we can apply, in the future, other automated techniques to classify, cluster or mine behavioral data. It is also possible to visualize which parts of a malware sample behavior are similar to another one's, indicating the same functionality or even code reuse.

Chapter 7

Malware Classification

7.1 Introduction

In this chapter, we introduce another way of extracting behavioral profiles, this time through a lower level tracing of the malware execution to capture performed instructions and their written values. Using these traces, we developed a two-step approach to group malware samples that are quite similar according to an empirically established threshold. We show that this type of behavior can be effectively used to cluster malware samples into families with good results by comparing our produced clustering with three distinct reference clustering sets. We also map the value traces back to the original instructions, thus showing that it is possible to search for similar blocks of code between two malware samples. The achieved results indicate that it is possible to improve the proposed techniques to find code reuse present in malware variants or samples that are produced using malicious development kits.

7.2 Motivation

Malicious software (malware) is a significant threat for cyber security. It has evolved from programs used to infect other programs and do damage to a personal computer, to political weapons and pathways to organized crime. Current malware operations vary from stealing sensitive data (secret documents, credentials, credit card numbers and other financial information) to attacking critical infrastructure and SCADA systems, as seen with the Stuxnet incident [52]. Today's malware employs many different ways to propagate, including social engineering techniques to deceive a user to click on e-mail attachments [39], as well as drive-by download attacks that exploit web browsers and their plug-ins [172].

Besides the diversity of purposes and propagation vectors that can be observed in current malware, cyber criminals have started to sell "do-it-yourself" kits that allow anyone to create customized malicious code. Moreover, malware developers usually apply obfuscation techniques (e.g., polymorphism and packing) to avoid detection by the security mechanisms installed on a victim's system. The result is a plethora of new samples that are discovered every day. This massive amount of malware variants overwhelms security analysts, and it increases the difficulty for anti-virus (AV) software providers to update their product's databases in a timely fashion [44].

Obfuscation techniques are a powerful tool to render static malware analysis approaches ineffective and to defeat those security tools (such as anti-virus scanners) that rely on signatures. To address this problem, researchers have proposed dynamic analysis systems, which rely on the observed runtime activities (behavior) for detection and classification. With dynamic analysis, a malware sample is run inside an instrumented environment, and the sample's actions are monitored. To capture and model the behavior of malicious code, dynamic analysis systems typically rely on system calls. The underlying assumption is that system calls capture the interactions of a program with its environment, and they can be used to monitor all persistent changes to the operating system. Also, system calls are relatively easy to capture (e.g., by hooking the system call table), and user-mode malware has no way to bypass the system call interface.

Unfortunately, system calls are not the perfect solution. They treat the program as a black box and capture activity at a relatively high level. For example, two programs might be very different "inside" but might yield the same, visible effect to the "outside" by invoking the same system calls. While this might not be an immediate problem for malware detectors, it makes it hard to distinguish between different malware families. More problematic, a malware author has a lot of flexibility in disguising behavior that is recorded at the system call level. For example, independent system calls can be scheduled in different order. Moreover, a particular change to the operating system can often be achieved through different system calls. Previous research [60] has demonstrated that system-call-based intrusion detection systems can be evaded by replacing one sequence of system calls with a semantically-equivalent but different one. Finally, researchers have introduced third-generation mutation engines that provide functional polymorphism: program activity at the system call level is automatically obfuscated [75].

To address the limitations of system-call-based detection and classification, we propose a novel approach to capture and model program (malware) behavior. Our approach is based on monitoring the data values that a program produces while it is performing its computations. More precisely, while the program is executing, we record a trace that contains all the values that (a certain subset of) instructions write. These writes can go either to a destination register or a memory location. By looking at the intermediate data values that a computation produces, we analyze the execution of a program at a much finer level of granularity than by simply observing system calls. The main intuition is that by using the data values, we can produce a very detailed profile that captures the activity of individual functions. Also, data values are tied very closely to the purpose (semantics) of a computation, and, hence, are not as easy to disguise as the code that performs the computation. Malware authors have introduced many ways in which code can be altered so that syntactically different instructions implement the same algorithm (e.g., dead code insertion, register renaming, instruction substitution). However, when an algorithm computes something, we would expect that, at certain points, the results (and temporary values) for this computation hold specific values. Our goal is to leverage these values to identify (possibly different) code that "computes the same thing."

We introduce a mechanism to record a trace of values that are written by "interesting" instructions. Leveraging these traces (one for each program execution), we have built two techniques that implement tasks that are important for malware analysis. First, we propose an efficient clustering technique to find malware samples that perform similar activity (implement similar computation), and, hence, are very likely to belong to the same malware family. Second, we show how traces can be used to find code reuse between samples that are different. That is, we look for similar subsets in traces that are mostly different. This allows us to find code snippets that are shared between malware samples of different families.

The main contributions presented on this chapter are the following:

- We propose a novel approach to capture and model behavior from dynamically analyzed malware. This approach is based on the sequence of values that a program writes (to memory or registers).
- We describe an efficient, two-step decision procedure to decide whether two value traces (produced by the execution of two programs) are similar. This is important because value traces can be very large.
- We leverage the aforementioned decision procedure to implement malware clustering and code reuse identification.
- We implemented a prototype system and collected the execution traces for more than 16 thousand malware samples. We demonstrate that the clustering results produced by our system have a good precision, and we show that we can find code reuse in unrelated malware families.

7.3 Related Work

Malware analysis, detection, and classification remain as open problems in the security field, as there is no holy grail that completely solves all these issues. In addition, malware authors are always developing new ways to subvert or evade security mechanisms. There are several previous systems that use assembly instructions to address malware identification/classification, but, to the best of our knowledge, there are no approaches that leverage memory manipulation (write) operations and values. In addition, most of the works that used assembly instructions collected them through binary static analysis techniques, which is provably limited for malware detection [103].

Dynamic malware analyzers such as [86], [168], [111] and [26] operate at the system call level and currently do not log the low-level values of an execution (memory and registers). VirtICE [122] could obtain this information from a malware sample by monitoring it from outside the emulated execution environment, but it does not collect this information in an automated way and it is not publicly available. Ether [45] performs both instruction and system call tracing to analyze malware in a transparent way by using hardware virtualization extensions, but it has several of prerequisites on the type of operating system, architecture and platform, which can limit its use.

Indeed, we can divide malware classification techniques according to how the traces were obtained — i.e. through either static or dynamic analysis — and to the type of behavior gathered

— i.e. either lower-level or assembly-related data or higher-level or system call information. Below, we discuss recent research based on static and dynamic analysis.

7.3.1 Static Analysis Approaches

Shankarapani et al. [134] propose two detection methods to recognize known malware variants without the need of new AV signatures: SAVE, which generates signatures based on a malware sample API calls sequence through the static analysis of its executable, and MEDiC, in which the signature of a malware sample is part of its disassembled code. SAVE performs an optimal alignment algorithm before applying similarity measure functions (cosine, extended Jaccard, and Pearson correlation). This kind of algorithm does not scale well if there are too many sequences or if the sequences are very large. For executable files whose size is among ≈500 Bytes to ≈1000 Bytes, the detection time can be in a range of few seconds (considering just one executable). They claim that MEDiC is more accurate in detection due to the different data used (assembly instructions) and then test it against a very limited set of known families — labeled by AVs — with few variants of each family to demonstrate their detection capabilities.

Kinable and Kostakis [82] performed a study of malware classification based on call graph clustering, where they measured the similarity of call graphs that were extracted from malicious binaries through matches that try to minimize the edit distance between a pair of graphs. In this context, a call graph is built after obtaining the function names from a sample through static analysis. The authors conclude that it is difficult to partition malware samples in well-defined clusters using k-means based algorithms, and chose to use the DBSCAN algorithm to cluster some sets, the larger having 1,050 samples. They state that this larger set had 72% correct clusters.

Zhang and Reeves [173] propose a method to detect malware variants that uses automated static analysis to extract the executable file semantics. These semantic templates are characterized based on the system calls executed by a malware sample and used in a weighted pattern matching algorithm that computes the degree of similarity between two code fragments. To generate a semantic template, they disassemble a malware sample and identify instructions that affect the values in memory or registers related to the target address of the `call` instruction. Those instructions are considered the code patterns of the templates and are input to a maximum weighted matching algorithm that calculates the similarity degree based on the mean of the similarity scores from matched pairs of elements. The authors use a few samples to test the proposed approach (less than a thousand), and they claim that it is possible to identify code reuse among malware variants by using the similarity coefficient calculation. In our work, we identify which instructions are being reused (and in which part of the trace), and our templates are based on instructions (arithmetic and logic) that modify values in memory and registers.

7.3.2 Dynamic Analysis Approaches

Park et al. [114] present a classification method that uses a directed behavioral graph extracted from system calls through dynamic analysis. The generated directed graphs from two malware samples are compared computing the maximal common subgraph between them and the size of this subgraph is used as the similarity metric. They evaluated their method on 300 malware samples divided into 6 families, and 80 benign Windows applications.

Bailey et al. [12] developed a method based on the behavior extracted from a malware sample after executing it in a virtualized environment. This behavior is considered the malware's fingerprint and represents the set of actions that changed the system state, such as files written, processes created, registry keys modified, and network connection attempts. They discuss the AV labeling inconsistency problems, which is also a motivation for our work. The tests were performed on a dataset of 3,700 samples from which the fingerprints were extracted. The normalized compression distance (NCD) was used as a similarity metric, and the pairwise single-linkage hierarchical clustering algorithm was used for classification purposes. However, as the distance matrix computation is an $O(N^2)$ operation, as it requires the calculation of the distances among every each pair, the process cannot scale well to larger datasets.

Rieck et al. [125] propose the use of machine learning techniques on malware behaviors composed of changes that occurred in a target system in term of API function calls. They ran their experiments on more than 10,000 malware samples divided into 14 families labeled by AV software. The behavioral profiles obtained from dynamic analysis serve as a basis to feature extraction and use the vector space model and the bag of words techniques. After that, the Support Vector Machines method is applied to the feature sets for classification purposes. The results presented show that their approach can characterize unknown new malware samples by assigning them to previously-defined clusters.

Bayer et al. [15] present a scalable clustering approach to classify malware samples based on the behavior they present while attacking a system. Dynamic analysis is used to generate the behavioral profiles — sequences of enriched and generalized information abstracted from system call data. The similarity metric used in their work is the Jaccard index, which is then used as an input to the LSH clustering method. To create a reference clustering set and verify the results accomplished by their work, they selected 2,658 samples for which there were a major degree of agreement among 6 AV programs, regarding the samples' attributed labels. They show that their approach could analyze 75,692 samples in 2 hours and 18 minutes.

Very recently, Jang et al. [78] introduced BitShred, an algorithm for fast malware clustering. In this paper, the authors present a new way to efficiently simplify and cluster features from inputs such as (static) code bytes and (dynamic) system call traces. The work is orthogonal to ours: The authors focus on improving clustering performance based on well-known malware activity models. We on the other hand introduce a novel and robust way to model malware activity.

As discussed, there is a myriad of possibilities to classify malware. However, previous work suffers from limitations, such as scalability constraints, a restricted utility of the results, the sole use of AV labeling as ground truth, or performing comparisons using a very limited malware dataset distributed among few different families. In our approach, we do not use AV labels as our main reference clustering; instead, the AV labels are used as an additional reference for the classification scheme. Moreover, our approach is more robust compared to previous work based on system call sequences and/or byte values in executable code. This is because we focus on the data values that are written by computation. These values reflect much closer the semantics of execution.

7.4 Data Value Traces

In this section, we discuss how we build *data value traces* to capture the activity of a (malware) program. To obtain these traces, we developed a prototype system that runs malware samples in an emulated environment. The prototype was developed using PyDBG [133]. This provides us with tight control of the debugging process. Also, PyDBG provides features to hide its debugging activity, which is useful to foil most malware attempts to detect the analysis environment.

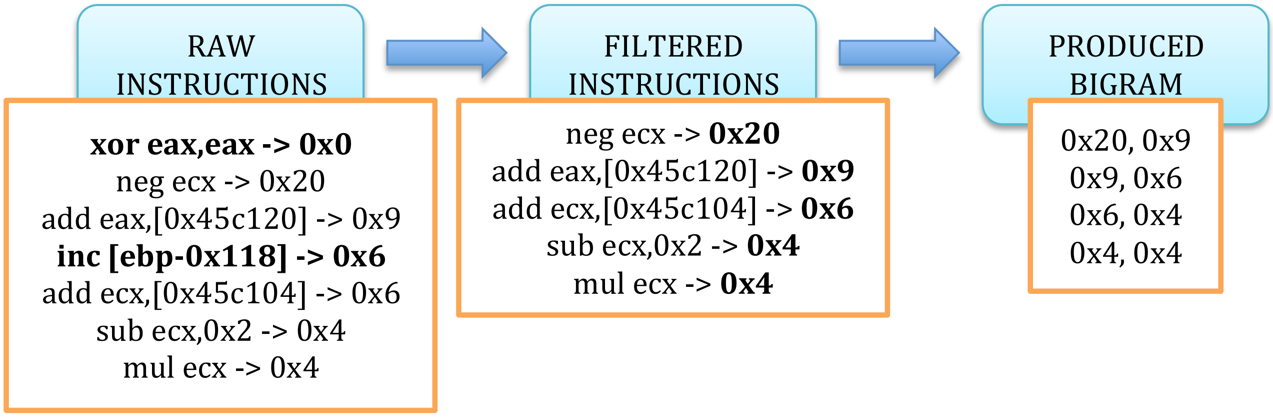
We use our prototype to record an ordered sequence of instructions that modify at least one register or memory value; that is, we are only interested in instructions that *write* to memory. For each of these instructions, we store the numeric value(s) of all memory locations and registers to which this instruction writes (typically, this is one). For instance, if a malware sample executes the instruction `sub esp,0x58`, we will log in our trace the line `sub esp,0x58; 0x12ff58`, which corresponds to the instruction and the new value written to register `%esp`, which is 0x12ff58 in this example (assuming that the initial value of `%esp` was 0x12ffb0). When the malware process terminates or a timeout is reached, we also take a snapshot of the content of the malware's executable (code) segments. This information is needed later to identify code reuse between malware samples, but it is *not* required to identify similarity between samples.

To collect the sequence of executed instructions, our system runs the sample in single-step debugging mode. More precisely, we single step through the code within the malware executable (and all code dynamically generated by the malware). However, calls that are made to standard operating system libraries are not logged, nor are the instructions executed inside these libraries. Fortunately, the system dynamically-loaded libraries (DLLs) are loaded into the memory region ranging from `0x70000000` to `0x78000000` (in Microsoft Windows XP). This speeds up the monitoring process and makes the resulting traces smaller. Also, it focuses the data collection on the actual malicious code.

To increase the efficiency of the collection process and to minimize the size of the traces, we only log a selected subset of instructions related to logic and arithmetic operations, namely: `add`, `adc`, `sub`, `sbb`, `mul`, `imul`, `div`, `idiv`, `neg`, `xadd`, `aaa`, `cmpxchg`, `aad`, `aam`, `aas`, `daa`, `das`, `not`, `xor`, `and`, `or`. We focus on these instructions because we are mostly interested in characterizing computations that the malware performs. Such computations will almost always involve arithmetic and logic instructions. Other instructions, such as data move or stack manipulation routines, are mostly used to prepare the environment for a computation, and hence, are less characteristic than the values that emerge directly as the result of a computation.

Note that the arithmetic instructions `inc` and `dec` are missing from the set of instructions that we are interested in. The reason is that we found these instructions to be typically involved in simple counters (for example, for loops), or in cases where a variable is first set to zero and then increased. Such counters reveal little information about the data that is being computed. Since these instructions appear very frequently, we decided to remove them from the traces. We also decided to remove instructions from the trace when they write the value 0, as this constant is not very characteristic of a particular computation.

We apply one last transformation to convert a sequence of instructions into the final data value trace. This transformation works by moving a sliding window of length two over the instruction sequence. For the two instructions in the window, we extract the two data values that these instructions write (one value for each instruction) and we aggregate then into a pair of values – a bigram. This bigram becomes one element of the data value trace. After the bigram is appended to the data value trace, we advance the sliding window by one instruction. Figure 7.1 shows an example of how a sequence of instructions is transformed into a data value trace (First, the instructions INC, DEC and those that write a '0' are removed. Then, the bigrams are generated from the remaining values).



```
xor eax,eax -> 0x0
neg ecx -> 0x20
add eax,[0x45c120] -> 0x9
inc [ebp-0x118] -> 0x6
add ecx,[0x45c104] -> 0x6
sub ecx,0x2 -> 0x4
mul ecx -> 0x4
```

```
neg ecx -> 0x20
add eax,[0x45c120] -> 0x9
add ecx,[0x45c104] -> 0x6
sub ecx,0x2 -> 0x4
mul ecx -> 0x4
```

```
0x20, 0x9
0x9, 0x6
0x6, 0x4
0x4, 0x4
```

Figure 7.1: Steps to produce a data value trace from a sequence of instructions.

The reason for transforming the sequence of instructions (or written) values into bigrams is the following: If we would compare simple traces of individual values, it is more likely that two values in two traces match by accident. By combining subsequent values into pairs, we add a simple form of *context* to individual data values. We found that this extra context significantly lowers the fraction of coincidental matches and improves the separation between different program executions.

7.5 Comparing Traces

As discussed in the previous section, we capture the activity of a malware program by collecting a trace that consists of a sequence of bigrams of data values that this program has written. For a number of applications (such as malware classification and clustering), we require a technique to determine whether the activities of two malware samples are similar. To perform this comparison, we have developed a two-step algorithm. This algorithm operates on two data value traces as input and outputs a similarity measure S. S is a value between 0 and 1, where 0 means completely different and 1 means identical. The two steps of the algorithm are explained in the next sections.

7.5.1 Step 1: Quick Comparison

The goal of the first step is to decide whether two traces are similar enough to warrant a further, more detailed comparison. This step works by creating a small "identifier" for each trace. This identifier is based on the k least-frequent bigrams that appear in a trace. The underlying assumption behind this choice is that if two samples are variants of each other, they should share some specific features or attributes that are particular to their family. Thus, we can discard the most common bigrams, which can appear within many different families, and focus on the specifics of a certain family's fingerprint. We have experimentally determined that a value of $k = 100$ yields good results.

More formally, we state our approach as follows. Let ID_{M_1} and ID_{M_2} be the k least-frequent bigrams from traces produced by malware samples M_1 and M_2. We compare these two malware identifiers by applying the Jaccard index [165] as follows:

$$J(ID_{M_1}, ID_{M_2}) = \frac{ID_{M_1} \cap ID_{M_2}}{ID_{M_1} \cup ID_{M_2}}, 0 \leq J \leq 1$$

If, and only if, the Jaccard index (ranging from 0 to 1) is greater than the empirically established threshold of 0.3[1], we move to the second step. Otherwise, the result of this computation is used as the similarity value (which indicates low similarity).

7.5.2 Step 2: Full Similarity Computation

In the next step, we compute the overlap of the entire two traces. More specifically, we compute the longest common subsequence (LCS) between them. Suppose that T_1 and T_2 are different data value traces and that L_1 and L_2 are their lengths, respectively. The similarity between the two traces is then calculated as follows:

$$C(M_1, M_2) = \frac{LCS(T_1, T_2)}{min(L_1, L_2)}$$

We chose the longest common subsequence over the longest common substring to tolerate small differences in the computations. Moreover, we note that using a standard LCS algorithm can be computationally expensive. We addressed this by calculating the LCS based on the GNU `diff` tool [164] output. Our experiments, evaluating a standard LCS implemented in C++ and our approximate LCS computation showed that we could accomplish faster results using our approach — in some cases ≈500× faster — with no significant loss of accuracy.

The original `diff` tool has the nice property that it inserts "barriers" while computing the longest common subsequences present in a textual input. Our `diff`-based LCS approach, referred from now on as *eDiff*, enhances this capability by (i) marking the regions that differ between two traces and (ii) by mapping the shared subsequences to the original instructions in the respective execution traces. As a result, we know exactly what malware code produced similar memory writes. This will be useful for identifying code reuse, as explained in Section 7.7.2. To map value traces back to instructions, we simply link the bigram values in the value traces to the raw instructions that produced those values.

[1] To choose this threshold (T), we performed tests with an increment of 0.1 for the range $0.0 < T \leq 0.5$

7.6 Applications

In this section, we discuss two applications that we built on top of our malware trace similarity technique. The first application is clustering; the idea is to group samples that show similar activity based on their value traces. The second application uses the data value traces to find cases of code reuse. That is, we want to find cases in which malware samples that belong to different families share one or more snippets of identical code.

7.6.1 Clustering

The input to the malware clustering application are a set of N data value traces, one trace for each of the N samples to be clustered. The goal is to find groups of malware samples that are similar. Clustering is implemented in two steps: pre-clustering and inter-cluster merging.

Pre-clustering: The goal of the pre-clustering step is to quickly generate an initial clustering and avoid having to perform $N^2/2$ comparisons. To accomplish this, we sequentially process each of the N samples, one after another (in random order), as follows: First of all, two samples are compared using the quick-comparison technique and, if their similarity is within the defined threshold, they are compared again, this time using the full comparison technique (*eDiff*). If they present over 70% of similarity, there is a decision step to elect a cluster leader and they are grouped together (Figure 7.2).

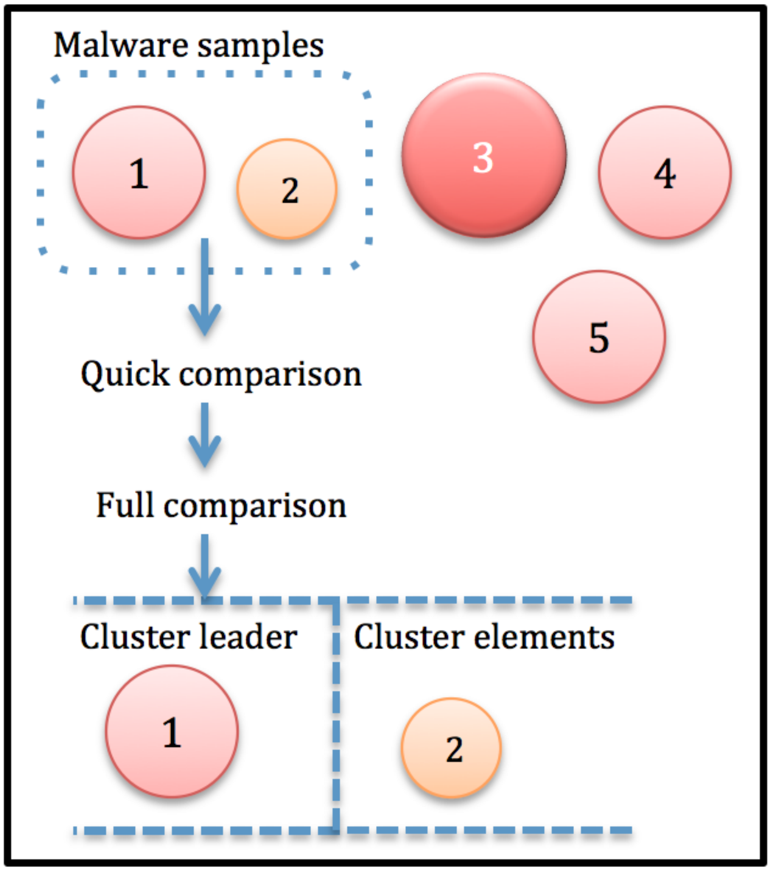


Figure 7.2: Quick comparison technique in action: forming a new cluster.

The comparison proceeds. Each new sample is compared to all cluster leaders (explained below), using the similarity computation described earlier (quick + full comparison techniques). When the trace for the new sample exhibits more than 70% similarity with one or more cluster leaders, this sample is merged with the existing cluster for which the similarity is highest.

When merging a trace with an existing cluster, we need to elect a new cluster leader (a cluster leader is basically the trace that is selected to represent the entire cluster). For this, we must make a selection between the existing cluster leader and the new trace. We select the *longer* trace as the new leader (Figure 7.3). We do this to increase the probability that a sample, whose behavior is similar to the activity of malware in a cluster, is properly matches with that cluster. In other words, by selecting the longest trace as the cluster leader, a new trace has more chances to find a long, common subsequence. By removing from the comparison computation all except one trace for each cluster, we greatly reduce the number of comparisons that need to be performed.

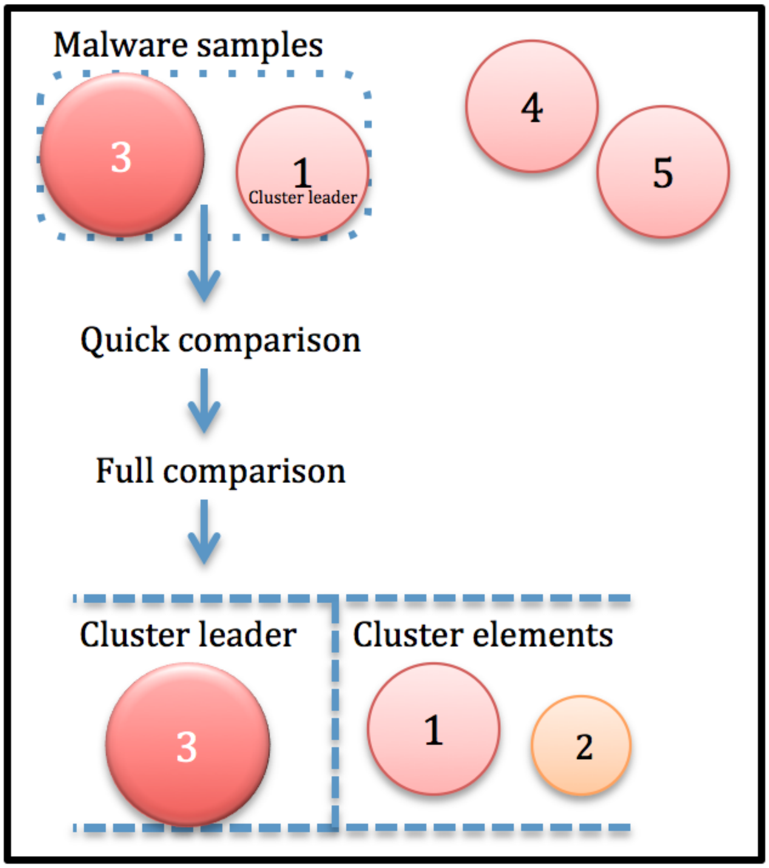


Figure 7.3: Election of a new leader based on trace length.

Otherwise, either the sample (and its trace) forms a cluster with a another sample that is still not member of another cluster, or it is inserted into a new cluster of its own, thus becoming the cluster leader. Figure 7.3 shows the following case: since none of the leaders are well suited, then the sample forms a cluster with another element that is not member of any previous cluster.

Inter-cluster merging: The pre-clustering step results in a set of initial clusters whose traces share at least 70% similarity. However, due to the nature of the quick comparison (first step

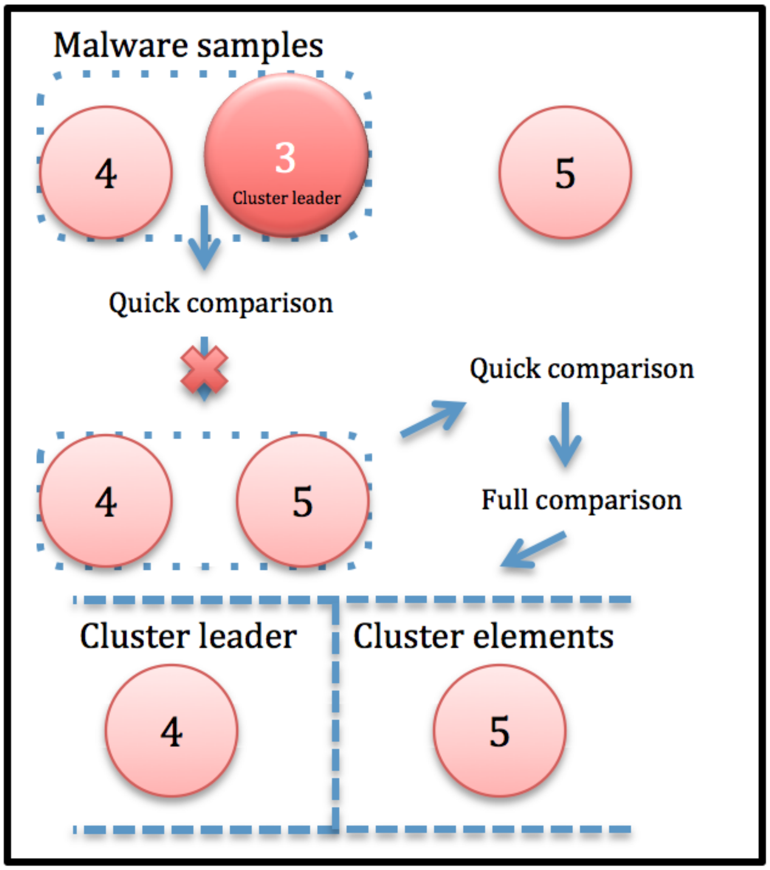


Figure 7.4: Sample clustered with a previously unclustered sample.

of the similarly comparison), there can be clusters that should be merged but are not. That is, it is possible that two traces are actually quite similar, yet their least-frequent bigrams are too different to pass the threshold. In this case, there are different clusters containing malware from the same family, and it is desirable to merge these clusters. The merging step is applied to the output of the pre-clustering step so as to generate a reduced amount of clusters. To this end, we perform a pairwise comparison between all cluster leaders, using our *eDiff* algorithm. If their similarity is greater than the same 70% threshold defined previously, the clusters are merged. This is illustrated in Figure 7.5.

As a byproduct of the merging step, the distances between all clusters are calculated. This yields a distance matrix, which can be analyzed to obtain insights into the phylogeny of the malware samples. For example, we could build a malware family tree (dendogram) by applying hierarchical clustering.

7.6.2 Code Reuse Identification

As mentioned previously, when comparing two traces, our algorithm not only computes a general similarity (overlap) score but also determines which parts of the traces are identical. When we find a stretch of values that are identical between two traces generated by executing different samples, we might naturally ask the question whether these values were produced by

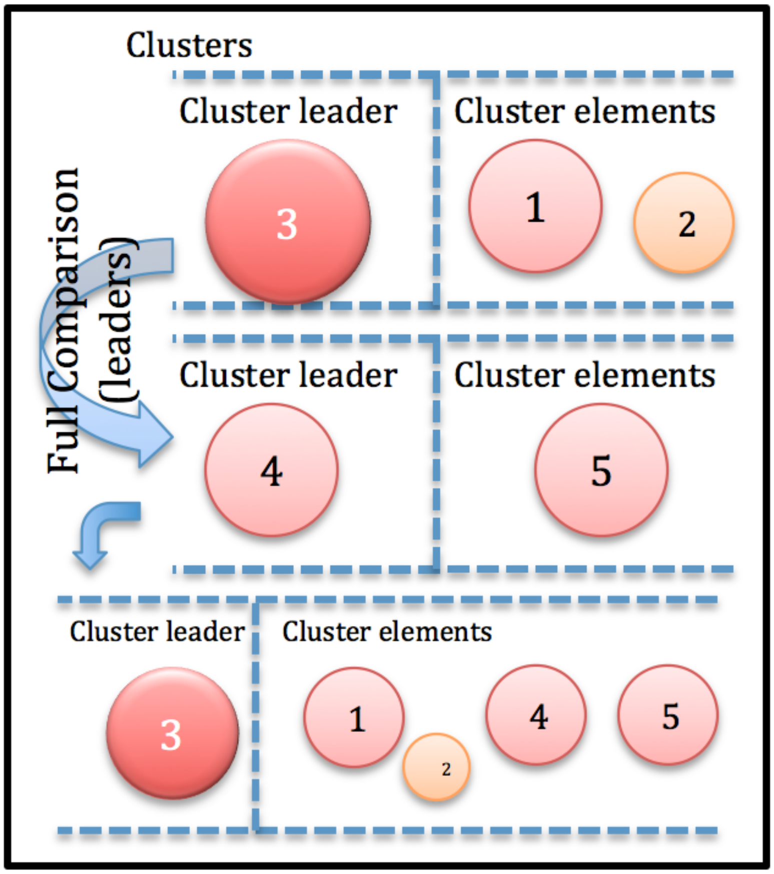
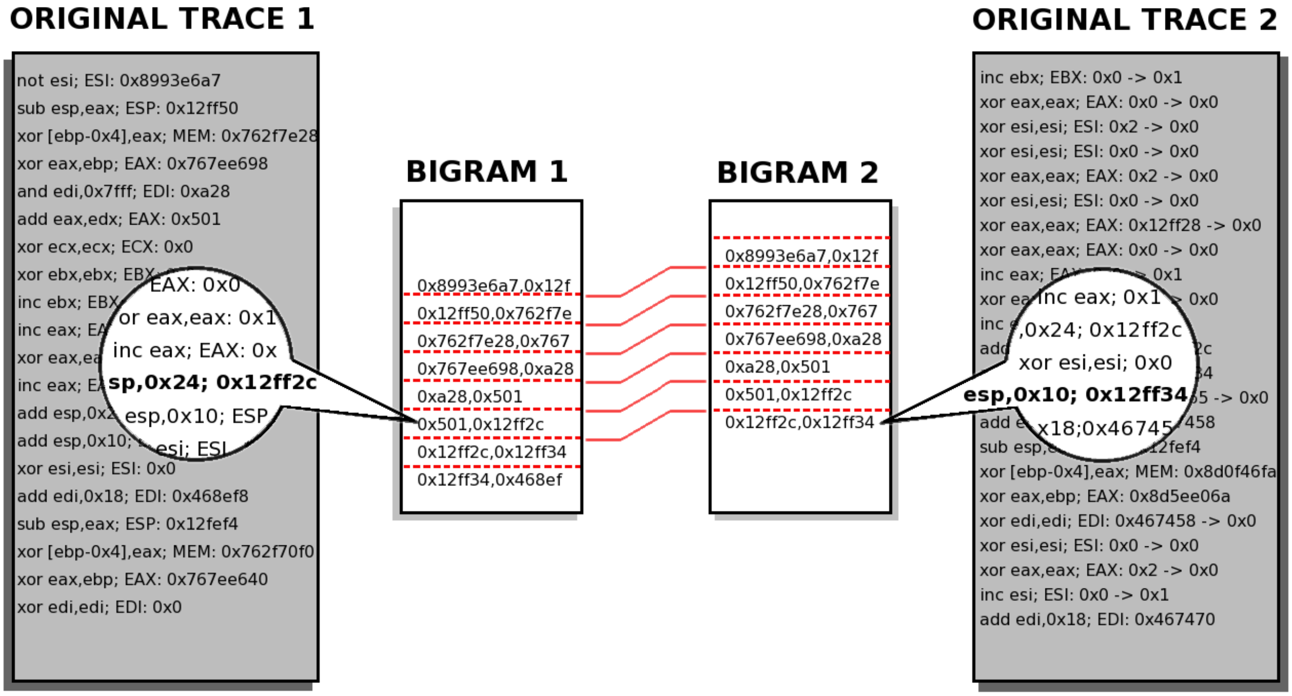


Figure 7.5: Inter-cluster merging being applied to an initial set of clusters.

similar code. This would allow us to identify code that is shared between samples that are otherwise different.

To identify code reuse, when *eDiff* compares two data value traces, it stores for each element (bigram) in the traces whether this element is unique to the trace or shared between both traces. For instance, let us assume that we have two traces. One contains the three bigrams: `(0x1,0x2)`, `(0x2,0x4)`, and `(0x4,0x5)`; the other contains the four bigrams: `(0x1,0x2)`, `(0x2,0x7)`, `(0x7,0x4)`, and `(0x4,0x5)`. In this case, *eDiff* would find that the first and last element in each trace are shared, while the middle one(s) are unique (to each trace). To find code reuse, we check both traces for the presence of at least four consecutive elements that are shared. The threshold of four was empirically determined and allows us to find shared code roughly at the function level. A higher threshold would be possible when we want to find longer parts of shared code. A lower threshold often yields accidental matches that do not reflect true code reuse.

Next, we require a mechanism to "map back" values in a data value trace to the instructions that produced them. This can be done easily because we retain the original instruction sequences that were recorded during dynamic analysis. However, these instructions are only part of the entire malware code, because we drop all non-arithmetic, non-logic instructions and all instructions that write a zero. Instead, we would like to find these instructions, as well as the surrounding ones, in the code of the malware. This mapping-back process can be better illustrated by the example of Figure 7.6, which shows that the code reuse identification process maps matching values to the original instructions so as to allow the search of actual code sharing (related to the instructions that produced the same written value and the surrounding ones). To do it, we leverage the dump of the program's executable sections that we obtained as described in Section 7.4.



```
not esi; ESI: 0x8993e6a7
sub esp,eax; ESP: 0x12ff50
xor [ebp-0x4],eax; MEM: 0x762f7e28
xor eax,ebp; EAX: 0x767ee698
and edi,0x7fff; EDI: 0xa28
add eax,edx; EAX: 0x501
xor ecx,ecx; ECX: 0x0
xor ebx,ebx; EBX
inc ebx; EBX
inc eax; EA
xor eax,ea
inc eax; EA
add esp,0x2
add esp,0x10;
xor esi,esi; ESI: 0x0
add edi,0x18; EDI: 0x468ef8
sub esp,eax; ESP: 0x12fef4
xor [ebp-0x4],eax; MEM: 0x762f70f0
xor eax,ebp; EAX: 0x767ee640
xor edi,edi; EDI: 0x0
```

```
0x8993e6a7,0x12f
0x12ff50,0x762f7e
0x762f7e28,0x767
0x767ee698,0xa28
0xa28,0x501
0x501,0x12ff2c
0x12ff2c,0x12ff34
0x12ff34,0x468ef
```

```
0x8993e6a7,0x12f
0x12ff50,0x762f7e
0x762f7e28,0x767
0x767ee698,0xa28
0xa28,0x501
0x501,0x12ff2c
0x12ff2c,0x12ff34
```

```
inc ebx; EBX: 0x0 -> 0x1
xor eax,eax; EAX: 0x0 -> 0x0
xor esi,esi; ESI: 0x2 -> 0x0
xor esi,esi; ESI: 0x0 -> 0x0
xor eax,eax; EAX: 0x2 -> 0x0
xor esi,esi; ESI: 0x0 -> 0x0
xor eax,eax; EAX: 0x12ff28 -> 0x0
xor eax,eax; EAX: 0x0 -> 0x0
inc eax; EA          > 0x1
xor ea               > 0x0
inc e
add                  2c
                     34
                     5 -> 0x0
add e                458
sub esp,e            12fef4
xor [ebp-0x4],eax; MEM: 0x8d0f46fa
xor eax,ebp; EAX: 0x8d5ee06a
xor edi,edi; EDI: 0x467458 -> 0x0
xor esi,esi; ESI: 0x0 -> 0x0
xor eax,eax; EAX: 0x2 -> 0x0
inc esi; ESI: 0x0 -> 0x1
add edi,0x18; EDI: 0x467470
```

Figure 7.6: Code reuse identification process.

To find the code in the malware program that contains the "shared instructions" (i.e. those four, consecutive instructions that wrote values shared between traces), we proceed as follows: The shared instructions are transformed into a regular expression pattern. This pattern contains the instructions in order, but applies the following modifications. First, we remove repeated, duplicate instructions (or blocks of instructions). This is because a certain instruction might have been executed inside a loop. Hence, it would appear multiple times in the dynamic trace, although it is only present once in the static code region. Second, we insert a wildcard after each instruction. This wildcard allows for a certain number of non-arithmetic, non-logic instructions that might have been present in the code but have been dropped from the dynamic trace.

The resulting pattern is then matched against the dumped code segment. When a match is found, we consider the resulting code block as a candidate for reuse. All matches that are found for each trace are compared, and when we find a sequence of identical code of a minimum length, we identify the code snippet as reused between malware samples.

7.7 Evaluation

In this section, we first describe the experiments we performed to evaluate the quality of our clustering results. Then, we discuss the code reuse we found among samples. To perform the experiments, we analyzed Windows PE32 executable programs with our system. For each executable, we collected a data value trace as described previously.

In total, we submitted 18,285 malware samples for analysis. Each sample was executed for ten minutes. We obtained a non-empty trace for 16,248 of these samples; the remaining ones produced an empty log. There can be many different reasons for an empty log - a malware sample might be broken, its runtime dependencies (libraries) might not all be present, or it might use aggressive anti-debugging techniques. However, overall, we obtained results for a significant majority of the malware samples. The samples were provided to us by the maintainers of a dynamic malware analysis system. They represent a diverse and recent set of different malware families that are currently active in the wild.[2]

7.7.1 Malware Clustering

The evaluation of the quality of a clustering algorithm is a complicated task [77], as clustering results are often not objectively right or wrong but depend on a number of factors, such as the metrics used to calculate the distances among samples and clusters, the final amount of clusters generated, the chosen heuristics, etc. A straightforward way to evaluate the results of a clustering run is to use a reference clustering. More precisely, we want to measure the degree of agreement between the clusters obtained by our system and the clusters of a reference set.

We used three different ways to obtain a reference clustering: one based on the static analysis of malware, one based on the dynamic analysis of malware, and the last one based on anti-virus (AV) labels. To generate these reference clustering sets, we proceeded as follows.

- For the static reference clustering, we used the output of a system provided to us by the authors of Jacob et al. [76]. This system looks for similarities in the bytes (instructions) of malware binaries that "shine through" packing.

- For the dynamic (behavioral) reference, the results were provided by clustering the results (reports) produced by the malware dynamic analysis tool that was the source of the samples. We used an approach similar to the one described in [15].

- For the AV label clustering quality measure, we scanned all the samples with three different antivirus engines ([9], [10], [58]). Then, we relabeled the results using only the general identifier for each label, skipping the versions (e.g., Trojan.Zbot-4955 became "zbot," Worm.Allaple-316 became "allaple," and so on). We assigned those compact labels to the samples already distributed in clusters that our approach produced and, finally, we calculated the level of agreement based on this attribution. Moreover, we calculated statistics about how many samples each one of the anti-virus programs was not able to detect, the cases in which all three AV engines agreed in their labeling, and whether they completely disagreed or not.

After we generated the reference clustering sets, we borrowed the precision and recall metrics from [15] to measure the quality of our clustering results. In that work, the authors state that

[2] For a complete list of MD5 sums of the samples, please contact the authors.

the **precision** aims to measure how well a clustering algorithm can distinguish among different samples, i.e. if each of our generated clusters contains only samples from the same family. Thus, if we have a reference clustering result $T = T_1, T_2, ..., T_t$, where t is the number of clusters, and our clustering approach results in $C = C_1, C_2, ..., C_c$, where c is the number of the generated clusters, for a dataset $A = a_1, a_2, ..., a_n$ with n samples, the precision P_j of a cluster can be calculated as follows, for each $Cj \in C$:

$$P_j = max(|C_j \cap T_1|, |C_j \cap T_2|, ..., |C_j \cap T_t|)$$

Then, the overall precision P is calculated as:

$$P = \frac{(P_1+P_2+...+P_c)}{n}$$

The **recall** metric serves as a basis to measure how well a clustering algorithm can recognizes similar samples, i.e. if samples of the same family are grouped together. Here, the goal is to have all samples from one family assigned to the same cluster by our algorithm. Hence, assuming a reference clustering $T = T_1, T_2, ..., T_t$ — with t clusters — and a clustering $C = C_1, C_2, ..., C_c$ — with c clusters — under evaluation, for each $T_j \in T$, we proceed in the following way to calculate the recall R_j value:

$$R_j = max(|C_1 \cap T_j|, |C_2 \cap T_j|, ..., |C_3 \cap T_j|)$$

The overall recall R is calculated as:

$$R = \frac{(R_1+R_2+...+R_t)}{n}$$

The product of the values obtained from the overall precision and recall can be used to measure the overall clustering **quality** ($Q = P \times R$).

For the their reference clustering based on AV labels, we measured the quality of our clustering scheme by defining the level of agreement related to the labels assigned to each sample in a cluster. Perdisci et al. [115] proposed to use two indexes (cohesion and separation) to validate their HTTP-based malware behavioral clustering. Those indexes use AV label graphs (undirected, weighted graphs) to verify that AV engines assign labels consistently within a cluster, and to check whether the families are well grouped. However, their approach “attenuates the effect of AV label inconsistency” due to the way the Cohesion Index is defined (there is a “gap” and a “distance” value that causes a boost in cohesion). To avoid this boost, we define a simpler level of agreement A for a cluster j that is calculated as:

$$A_j = \frac{\sum max(label_N)}{T_j(total\ \ samples\ \ in\ \ j)}$$

where $label_N$ is the set of the assigned labels and their related frequencies for each AV engine for each cluster ($N = avg, avira, fprot$).

This level-of-agreement value A corresponds to the average of the most assigned label per AV engine in a certain cluster. For instance, one of our clusters contains three samples (identified by the binary’s MD5) and the comma separated values assigned by the three AV engines, as shown below.

```
MALWARE MD5                      AVG,AVIRA,F-PROT
072cb45db4b7e34142183bc70bf8b489 agent_r,agent,busky
050a0b8b78cad111352b372417e467fe agent_r,agent,busky
063d85386df0edb28b3f0182b83a4fe3 agent_r,runner,busky
```

In this example, we have $label_{avg} = 3$, $label_{avira} = 2$ (the "agent" label), and $label_{fprot} = 3$. As a result, we a have a sum of 8 from the labels over 9 samples, resulting in a level of agreement $A = 0.89$ for this cluster. This is a simple, unbiased way to calculate the quality of our clusters and produced very good results during the full dataset evaluation, which is presented below.

Preliminary tests. Before we applied our clustering technique to the entire dataset, we ran preliminary experiments using a smaller subset, which consisted of 1,000 random samples. Those initial tests were important to experiment with and determine different threshold parameters. In particular, we varied the similarity threshold for the second step of the algorithm from 0 to 100% (incrementing by 10% after each iteration).

The precision and recall values for the different thresholds can be seen in Figure 7.7 and in Figure 7.8 where we compare our results to the static and behavioral reference clustering, respectively.

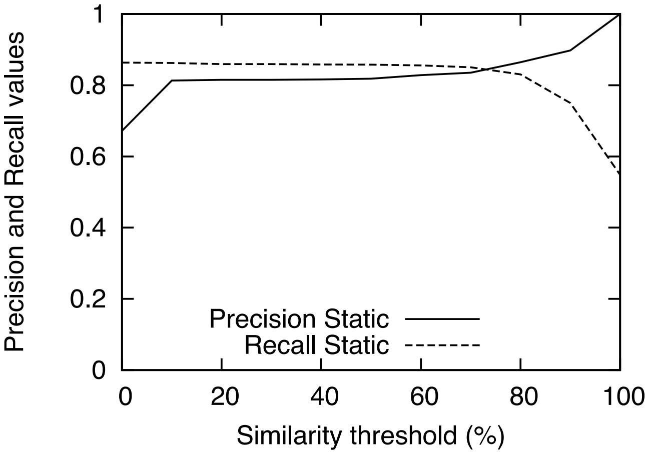


Figure 7.7: Precision and recall of a reduced dataset of 1,000 samples (static clustering).

The static reference clustering for these 1,000 samples consists of 546 clusters and the behavioral reference clustering consists of 335 clusters. The quality values calculated for each iteration (different threshold value), as well as the amount of clusters produced by our approach, can be seen in Table 7.1 (Q_S and Q_B are respectively the quality of our clustering related to the static and behavioral reference clustering). The highest quality, i.e. the average between the obtained static and behavioral quality values was observed for a similarity threshold of 70%. Moreover, the AV labels' level-of-agreement value for this threshold is also very high (0.894).

Full dataset. In the next step, we ran our system on the entire data set. Based on the results of the preliminary evaluation, we defined a similarity threshold of 70% for the *eDiff* process. We continued to use the initially-established Jaccard index threshold of 0.3 for the quick comparison.

The amount of clusters produced by the two reference clustering sets for the 16,248 samples with traces were 7,900 clusters for the static approach and 3,410 for the behavioral one. Our

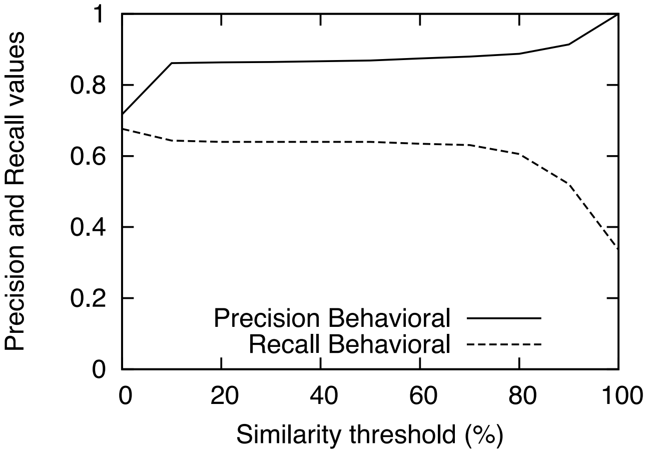


Figure 7.8: Precision and recall of a reduced dataset of 1,000 samples (dynamic clustering).

Table 7.1: Quality (Q) and amount (#) of our produced clustering (1,000 samples).

T	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
#	551	583	586	586	588	591	602	612	639	734	1000
Q_S	0.579	0.701	0.700	0.700	0.700	0.701	0.708	0.710	0.717	0.673	0.548
Q_B	0.485	0.554	0.552	0.552	0.554	0.555	0.554	0.554	0.537	0.476	0.336

approach produced 7,793 clusters that were compared to the reference clustering sets, generating the precision values of 0.758 and 0.846 and the recall values of 0.81 and 0.572 for the static and behavioral reference, respectively. Calculating the AV labels' level-of-agreement for our clustering yielded 0.871. These results are summarized in Table 7.2, along with the associated quality results.

Table 7.2: Precision, Recall and Quality for the three reference clustering sets ($T = 70\%$).

Ref. Clustering	**Precision**	**Recall**	**Quality**
Static	0.758	0.810	0.614
Behavioral	0.846	0.572	0.485
AV labeling	-	-	0.871

The values from Table 7.2 yield average results of 0.802, 0.691, and 0.656 for precision, recall, and quality, respectively. It took 37 hours and 42 minutes to produce our clustering: 13 hours were spent in the pre-clustering step and 24 hours and 42 minutes in the merging step. The merging step is slower (it requires $\frac{N_C \times (N_C+1)}{2}$ comparisons, where N_C is the amount of clusters) as it involves comparing all the leaders of the clusters with each other, using *eDiff*. However, this is a crucial step, as it calculates the distances between all clusters and merges those whose samples have been incorrectly discarded by the quick comparison step.

Interestingly, for the entire malware set, the three AV engines completely agreed for only 8% of 18,285 samples. Moreover, all three AV engines agreed that 13.93% of the samples were "clean," being unable to detect them as a malware (although all AV engines had the latest signatures). Note that all our samples were obtained as part of exploits (e.g., from

honeypots specifically tailored to act as malware collectors) and from collections containing verified malware, and hence, they are very likely malicious. More in detail, AVG, Avira, and F-Prot were individually unable to detect 19.01%, 15.50% and 20.28% of all malware samples.
Discussion. The use of different reference clustering sets is important to validate a new approach. In particular, it allows for a richer understanding of the clustering schemes used as ground truth, as well as of our new clustering.

When we used the static clustering as ground truth, the precision obtained was 0.758, and the recall was 0.810. The use of the behavioral clustering as reference yielded a precision value of 0.846 and a recall value of 0.572. Finally, the AV labels' level-of-agreement resulted in a value of 0.871, corroborating our clustering scheme by showing that similarly labeled samples are mostly grouped together by our approach.

Overall, we find that the more specialized the reference clustering is, the better the recall for our approach. On the other hand, the more generalized the reference cluster is, the better our precision. When looking at the static clustering, we find that it tends to split the samples according to the structural information of the binary file, as well as to the packer they used. This leads to cases in which the samples of one specific malware family are separated into more than one cluster (because they are packed or encrypted with different tools). As a result, we obtain very specific clusters (which translates to good recall values).

Behavioral clustering is more generic, as it groups samples based on the actions they perform in a dynamic analysis system. This leads to the generation of a smaller amount of clusters, as the general behavior of a malware family after unpacking tends to be similar. Thus, the relaxed boundaries of behavioral clustering result in better precision for our approach (it is less likely that we miss that two samples are different). However, the behavioral clustering results in a worse recall, as it is more likely that our approach is more precise and (correctly) splits a cluster of different families that exhibited similar behavior. Indeed, the AV labels' level-of-agreement value is consistent with the precision values found for the other reference clustering sets, showing that our clusters are well formed and contain mostly samples from the same family.

To summarize, we find that our approach achieves a good clustering of malware samples. The results fit in between those produced by the static and the behavioral clustering approaches, and we were able to expose and account for certain errors present when performing clustering using static measures or dynamic runtime activity.

7.7.2 Code Reuse

To look for code reuse in samples that likely belong to different malware families, we only check pairs of malware clusters that are sufficiently different. More precisely, based on the clustering results obtained in the previous step, we look for pairs of clusters that have a similarity score between 10% and 30%. We identified 974 pairs of clusters that fulfill this requirement.

For each of these pairs (i.e. for the corresponding cluster leaders), we ran our code reuse identification technique (as explained in Section 7.6.2). That is, we first checked for four consecutive, shared values among each pair of traces. When such a stretch of values was found, we produced the corresponding code pattern and matched it against the dumped malware code. For each pattern match, we subsequently checked whether the identified code regions were identical.

We discovered 15 pairs (involving ten different clusters) that share code between them. More precisely, we found seven different blocks of code that seem to be reused among samples. These code blocks can be seen in the Table 7.3. We sent the ten representatives (one for each cluster) to VirusTotal [67]. Looking at the results, we noticed that the most common label assigned to all of them refers to different Trojan malware that all seem to attack online games (and, apparently, shared code to do so).

Table 7.3: Blocks of code shared among samples.

Instructions
add ebx,ebx ; jnz 0x4070f9 ; pop es ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc
add ebx,ebx ; jnz 0x40710b ; pop es ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc
add ebx,ebx ; jnc 0x407100 ; out dx,eax ; jnz 0x40711c ; or [ebx-0x3117ce2],ecx ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc
add ebx,ebx ; jnz 0x40713b ; pop es ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc
add ebx,ebx ; jnz 0x407148 ; pop es ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc
add ebx,ebx ; jnz 0x407158 ; pop es ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc
add ebx,ebx ; jnc 0x40714d ; out dx,eax ; jnz 0x407169 ; or [ebx-0x3117ce2],ecx ; mov ebx,[esi] ; push ds ; sub esi,0xfffffffc

In an additional experiment, we applied the *eDiff* algorithm directly to the instructions extracted from the static, dumped code regions (for all 974 pairs of clusters). This resulted in 889 pairs with less than 30% of similarity, and only three pairs with a similarity score above 70%. The different similarity values for the memory-mapped code regions for all 974 pairs can be seen in Table 7.4. As expected, the code similarity is low, which is well captured by the fact that the data value traces are also dissimilar. That is, different malware samples produce, in almost all cases, very different data value traces (recall that the 974 pairs reflect clusters that have produced different value traces).

Table 7.4: Similarity scores (S) for static malware code.

S (%)	**0-10**	**10-20**	**20-30**	**30-40**	**40-50**	**50-60**	**60-70**	**70-80**	**80-90**	**90-100**
Pairs	787	79	23	52	15	6	9	1	2	0

The three aforementioned exceptions, where the static code regions (instructions) are more than 70% similar, involve six unique clusters. This cases are interesting because they represent (somewhat) similar code that has resulted in different value traces. In Table 7.5, we show those pairs, their respective AV labels, and the portion of code they share (as produced by *eDiff*).

Table 7.5: Shared code rate (%) of malware samples' pairs and respective AV labels.

Pair	**AV labels (summarized): AVG | Avira | F-Prot**	**Share %**
MW 1	Downloader.Generic7 | Dldr.Delphi.Gen | Downldr2	76.22
MW 2	Downloader.Generic7 | Dldr.Delphi.Gen | Downldr2	
MW 3	Downloader.Agent | DR/Zlob.Gen | NSIS.gen	84.66
MW 4	Generic12 | DR/Zlob.Gen | Zlob.X.gen	
MW 5	Downloader.Zlob | DR/Zlob.Gen | DldrX	86.70
MW 6	FakeAlert | DR/Zlob.Gen | SuspPack.AJ.gen	

As we can observe in Table 7.5, the AV labels assigned to the first pair's samples are generic, i.e. they do not refer to any family and they were probably assigned by heuristics that verified a "downloader-like" behavior. Although this pair shares 76.22% of its samples' static code and their AV labels are consistent, the runtime behaviors were very different. This could be due to the fact that certain remote resources were not available, and as a result, the downloaders might have exhibited different behavior. Another possibility is that the downloader simply fetches, installs, and runs different malware components (that result in different activity). For the other two pairs, we found code similarities of 84.66% and 86.70%, respectively. We can observe significant disagreement among the labels, some distinct enough such as SuspPack and FakeAlert, others linking the pairs to the Zlob Trojan family [166]. Again, most of the labels agreed that the samples act as downloaders. The common theme among all three cases is that the samples likely share some common downloading code. This results in the samples to be grouped together based on their static code. However, for the various reasons discussed above, the programs exhibited different activity at the trace level.

7.8 Concluding Remarks

Malware classification is an important process to improve systems security, as it allows for the identification of malicious programs based on attributes that can be used to generate signatures for detection or incident response.

In this chapter, we empirically demonstrated that the values stored in memory and registers after write operations can be used to detect and cluster malware in families. We also presented a different approach to perform the similarity score calculation that is simple and effective when applied to the malware problem. We compared the results from more than 16 thousand malware samples executed and processed in our prototype system to three reference clustering sets—static, behavioral (dynamic), and AV labeling—and ours reached an average precision value of 0.802 for the first two sets and a level of agreement value of 0.871 for the last one.

Finally, we showed that our classification process can also be used to verify for code reuse, which helps to investigate the sharing of functions in different families of malware. This result is very promising, because it is a first step towards a mechanism to search for common functions or procedures with specific functionality that are reused in malicious code that is seen in the wild. This, in turn, can then be used by law enforcement during attribution analysis.

Chapter 8

Conclusion

This thesis discusses several aspects related to the behavior of malicious programs, from the definition of potentially dangerous activities that malware may perform during an infection, to the proposition of a behavior-based taxonomy, to the detection, clustering and visualization of malware. The aim of the research work presented in this thesis is mainly to provide a better understanding of how the diversity of current malware samples actually behave, as well as to aid in the development of practical and effective incident response procedures. In this thesis, we introduced a general, flexible and extensible taxonomy, yet of simple use and easy to understand, which provides an overall view of malware infection. We showed that this taxonomy may be used in a plenty of applications, which are distributed among this thesis' chapters. Below, we present a review of the topics discussed in this thesis.

In Chapter 2, we provide a background about malicious software and malware taxonomies. We briefly presented a history of the evolution of malware, since they were only concepts to their changing into weapons of an already established cyber war. We focused on the multiple behaviors that a malware sample can exhibit in order to accomplish its infection, unlike the malware's well-behaved nature present in the current classification scheme. This led us to discuss the naming scheme adopted by antivirus (AV) vendors, which does not handle modern malware very well, confusing the users and security analysts. Then, we discussed some security-related taxonomies and taxonomic properties, as well as we described several taxonomies regarding specific types of malware.

In Chapter 3, we introduce our proposed taxonomy. Initially, we discussed the behavioral aspects of malware and some recent work related to malware behavior. We also defined, in the scope of this thesis, what is the general behavior of a program (divided among the active, passive and neutral subsets) and what is the suspicious behavior, which can be involved in malware attacks. Furthermore, we gathered 24 suspicious behaviors that are potentially dangerous to a computer system and classified them in network or operating system activity groups. Based on those behaviors, we leveraged a behavior-centric taxonomy that considers four classes, the suspicious behaviors associated to each of them, and the violated security principle.

In Chapter 4, we evaluate the proposed behavior-centric taxonomy using over 12 thousand recent malware samples collected in the wild, either from phishing e-mail messages and donations from collaborators, or from malware collectors and public databases. We introduced our infrastructure for dynamic malware analysis, which was used to run all the malware samples from our dataset in a controlled environment in order to extract their behavioral profiles. We applied our taxonomy to the samples dataset and evaluated the produced results. In addition, we obtained some AV labels and classified the samples accordingly. Finally, we compared these two approaches with respect to information they are able to provide to their users, showing that our taxonomy is very useful to yield a better understanding about AV detected (and undetected) malware.

In Chapter 5, we present BanDIT, a system to detect Internet banking malicious programs (bankers) that is built upon a specialization of our dynamic malware analysis system and our behavior-centric taxonomy. We showed that our proposed taxonomy can be easily extended to add more behaviors, as well as that it can be used in different scopes (other than classification). In this case, we borrowed behaviors from the taxonomy to identify bankers, which are a major threat for Internet banking users, and empirically showed that we are able to detect almost all bankers of a malware dataset using three techniques: visual identification, network pattern matching and monitoring of filesystem changes.

In Chapter 6, we leverage two interactive visualization tools that take advantage of behavioral profiles to aid in computer security incident response procedures. The goal is to apply visualization techniques to the behavioral trace to produce an easy to use system whose intent is to allow an analyst to interact with the malware execution steps and identify patternsor similarities among malware families and unknown samples.

In Chapter 7, we introduce a novel way to classify malware that, otherwise based on the execution behavior, considered the values written in memory or registers to generate the behavioral profiles. We analyzed over 16 thousand malware samples and proposed a heuristic method to approximate the longest common subsequence (LCS) function to calculate the similarity among them. We proposed a clustering technique that involved two steps, a quick comparison and a full and more expensive comparison to group our malware samples. The results indicated that we were able to cluster them with precision rates higher than 0.8 and that our results were consistent with other clustering approaches (static, dynamic and AV labeling reference sets).

Future Work

The study of malware behavior is a wide research field that requires constant monitoring as new technologies rise. Malware behavior can also be obtained in a great variety of forms (e.g., reverse engineering, static analysis, debugging, disassembling, controlled running etc.). Moreover, there is still not a widely adopted standard to handle malware, creating several opportunities for future works.

We outline below some possible future works involving the themes discussed in this thesis. It is worthwhile to note that each chapter has its own discussion about the limitations and the additional work that can be done regarding its respective subject.

Behavioral Extraction. There is a plethora of research works aiming to monitor intrusive activities and to extract malware features from the execution of samples in controlled environments. However, novel hardware features and new versions of operating systems often change the *status quo*, either from the malware developer perspective, or from the security researcher. Furthermore, malware variants are able to subvert current monitoring systems, consequently creating the need for updates in these systems so as to handle a broader range of malware types.

Classification. As underlined in this thesis, our behavior-centric taxonomy can be extended to address new potentially dangerous behaviors. The discovering of new behaviors involves the continuous analysis of malware samples that appear in the wild, improving the classification process as a whole. In addition, ongoing future works include evolving the taxonomy until it can become a standard, as well as to test the effectiveness of using well-established machine learning algorithms to perform malware classification in a more precise and automated (unsupervised) way.

Detection. Finally, newly discovered behaviors, features and monitoring methods can help the detection process. Detection is important since it may improve the security of systems (and its users), turning feasible the development of new tools and solutions to serve as countermeasures for malware infections. Moreover, additional detection techniques can allow the improvement of AV engines.

Publications

The following list includes the published results that served as the basis for this text. Most of them are ranked by Brazilian Qualis Ranking 2012-2014, as shown next.

1. **An Empirical Analysis of Malicious Internet Banking Software Behavior.** André Ricardo Abed Grégio, Vitor Monte Afonso, Victor Furuse Martins, Dario Simões Fernandes Filho, Paulo Lício de Geus, Mario Jino. ACM Symposium on Applied Computing (SAC). Coimbra, Portugal, March, 2013. **Qualis A1**.

2. **Tracking Memory Writes for Malware Classification and Code Reuse Identification.** André Ricardo Abed Grégio, Paulo Lício de Geus, Christopher Kruegel, Giovanni Vigna. 9^{th} Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Lecture Notes in Computer Science, Springer Verlag. Greece, July 2012. **Qualis B1**.

3. **Pinpointing Malicious Activities through Network and System-Level Malware Execution Behavior.** André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, Mario Jino, Rafael Duarte Coelho dos Santos. 12^{th} International Conference on Computational Science and Its Applications (ICCSA), Lecture Notes in Computer Science, Springer Verlag. Brazil, June 2012. **Qualis B1**.

4. **Interactive, Visual-Aided Tools to Analyze Malware Behavior.** André Ricardo Abed Grégio, Alexandre Or Cansian Baruque, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, Mario Jino, Rafael Duarte Coelho dos Santos. 12^{th} International Conference on Computational Science and Its Applications (ICCSA), Lecture Notes in Computer Science, Springer Verlag. Brazil, June 2012. **Qualis B1**.

5. **A Hybrid Framework to Analyze Web and OS Malware.** Vitor Monte Afonso, Dario Simões Fernandes Filho, André Ricardo Abed Grégio, Paulo Lício de Geus, Mario Jino. IEEE International Conference on Communications (ICC), Proceedings of the IEEE ICC'12. Canada, June 2012. **Qualis A2**.

6. (In Portuguese) **Análise Visual de Comportamento de Código Malicioso.** Alexandre Or Cansian Baruqu, André Ricardo Abed Grégio, Paulo Lício de Geus. Workshop de Trabalhos de Iniciação Científica e de Graduação (WTICG), Anais do XI SBSEG. Brazil, 2011.

7. (In Portuguese) **Análise Comportamental de Código Malicioso através da Monitoração de Chamadas de Sistema e Tráfego de Rede.** Dario Simões Fernandes Filho, André Ricardo Abed Grégio, Vitor Monte Afonso, Rafael Duarte Coelho dos Santos, Mario Jino, Paulo Lício de Geus. Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG), Anais do X SBSEG. Brazil, October 2010. **Qualis B4**.

Other published papers that are not directly related to this thesis are:

1. **A Malware Detection System Inspired on the Human Immune System.** Isabela Liane Oliveira, André Ricardo Abed Grégio, Adriano Mauro Cansian. 12^{th} International Conference on Computational Science and Its Applications (ICCSA), Lecture Notes in Computer Science, Springer Verlag. Brazil, June 2012. **Qualis B1**.

2. **Analysis of web-related threats in ten years of logs from a scientific portal.** Rafael Duarte Coelho dos Santos, André Ricardo Abed Grégio, Jordan Raddick, Vamsi Vattki, Alex Szalay. Cyber Sensing 2012, Proceedings of SPIE. USA, May 2012.

3. **Behavioral analysis of malicious code through network traffic and system call monitoring.** André Ricardo Abed Grégio, Dario Simões Fernandes Filho, Vitor Monte Afonso, Rafael Duarte Coelho dos Santos, Mario Jino, Paulo Lício de Geus. Defense, Security and Sensing 2011, Proceedings of SPIE. USA, April 2011.

4. **Visualization techniques for malware behavior analysis.** André Ricardo Abed Grégio, Rafael Duarte Coelho dos Santos. Defense, Security and Sensing 2011, Proceedingsof SPIE. USA, April 2011.

5. **A hybrid system for analysis and detection of web-based client-side malicious code.** Vitor Monte Afonso, André Ricardo Abed Grégio, Dario Simões Fernandes Filho, Paulo Lício de Geus. IADIS International Conference WWW/Internet (ICWI'2011), Proceedings of ICWI, 2011. **Qualis B2**.

6. (In Portuguese) **Sistema de coleta, análise e detecção de código malicioso baseado no sistema imunológico humano.** Isabela Liane Oliveira, André Ricardo Abed Grégio, Adriano Mauro Cansian. Conferência IADIS Ibero-Americana WWW/Internet (CIAWI), Anais da CIAWI, 2011.

7. (In Portuguese) **xFile: Uma Ferramenta Modular para Identificação de Packers em Executáveis do Microsoft Windows.** Victor Furuse Martins, André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus. Workshop de Trabalhos de Iniciação Científica e de Graduação (WTICG), Anais do X SBSEG. Brazil, October 2010.

8. **Malware distributed collection and pre-classification system using honeypot technology.** André Ricardo Abed Grégio, Isabela Liane Oliveira, Rafael Duarte Coelho dos Santos, Adriano Mauro Cansian, Paulo Lício de Geus. Data Mining, Intrusion Detection, Information Security and Assurance, and Data Networks Security. Proceedings of SPIE Defense, Security and Sensing, USA, 2009.

Furthermore, during the review of related research works, we produced two book chapters that were presented as short courses:

1. (In Portuguese) **Técnicas para Análise Dinâmica de Malware.** Dario Simões Fernandes Filho, Vitor Monte Afonso, Victor Furuse Martins, André Ricardo Abed Grégio, Paulo Lício de Geus, Mario Jino, Rafael Duarte Coelho dos Santos. Minicursos do SBSEG 2011, pp.107–147, SBC, Brazil, 2011.

2. (In Portuguese) **Análise e Visualização de Logs de Segurança.** Rafael Duarte Coelho dos Santos, André Ricardo Abed Grégio. Livro de Minicursos do Computer on the Beach, Univali, Brazil, 2010.

Submitted Articles

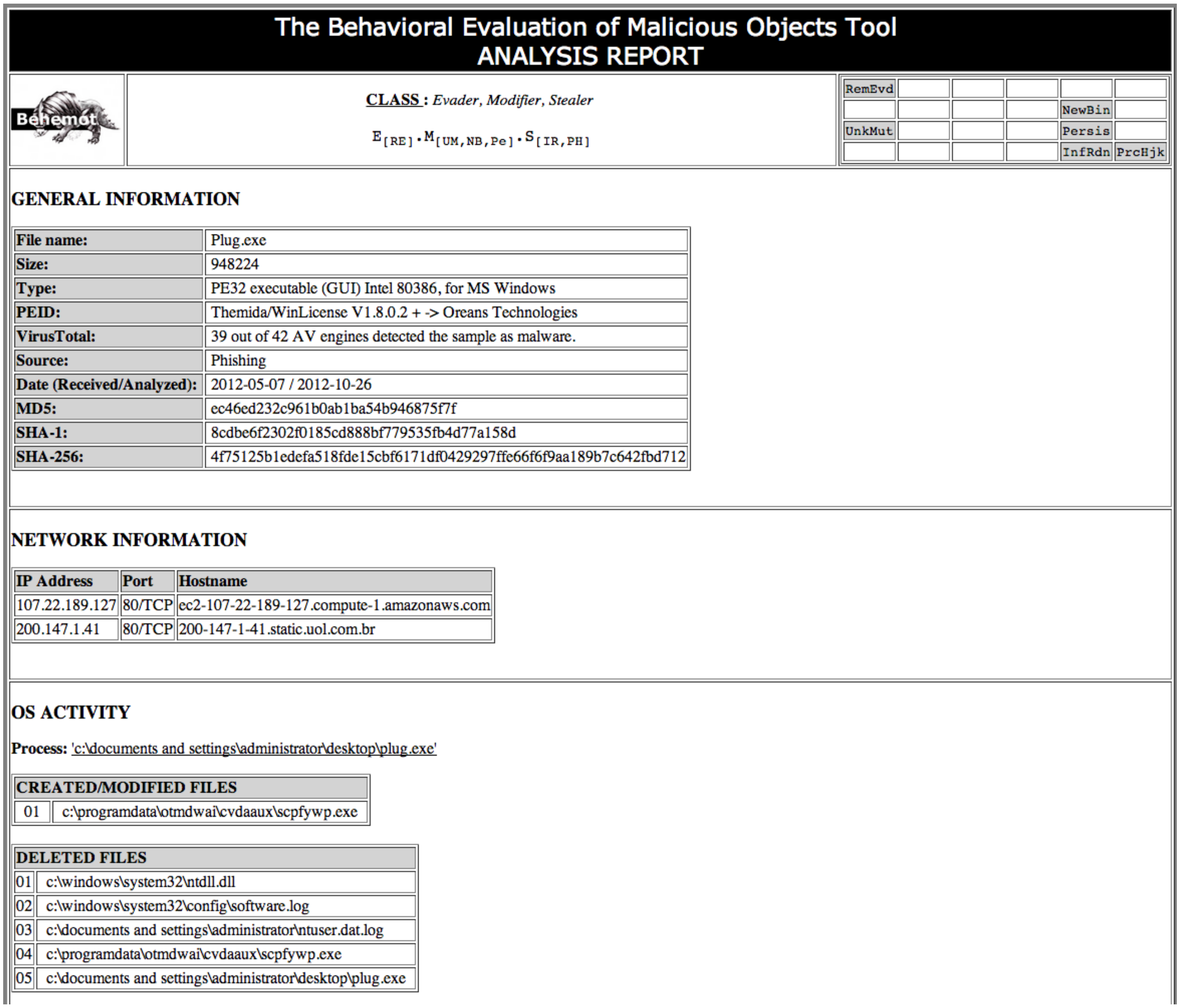
Finally, we submitted two articles to journals specialized in computer security:

1. **A Behavior-Centric Malware Taxonomy.** André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, Mario Jino. Computers & Security, ISSN: 0167-4048, Elsevier. Submitted in October 2012. **Qualis B1**.

2. **Tracking Memory Writes for Malware Classification and Code Reuse Identification (Extended Version).** André Ricardo Abed Grégio, Paulo Lício de Geus, Christopher Kruegel, Giovanni Vigna. Journal of Computer Security, ISSN: 0926-227X, IOS Press. Submitted in June 2012.

Appendix A

Example of BehEMOT Report



The Behavioral Evaluation of Malicious Objects Tool
ANALYSIS REPORT

CLASS : *Evader, Modifier, Stealer*

$E_{[RE]} \cdot M_{[UM,NB,Pe]} \cdot S_{[IR,PH]}$

RemEvd					
				NewBin	
UnkMut				Persis	
				InfRdn	PrcHjk

GENERAL INFORMATION

File name:	Plug.exe
Size:	948224
Type:	PE32 executable (GUI) Intel 80386, for MS Windows
PEID:	Themida/WinLicense V1.8.0.2 + -> Oreans Technologies
VirusTotal:	39 out of 42 AV engines detected the sample as malware.
Source:	Phishing
Date (Received/Analyzed):	2012-05-07 / 2012-10-26
MD5:	ec46ed232c961b0ab1ba54b946875f7f
SHA-1:	8cdbe6f2302f0185cd888bf779535fb4d77a158d
SHA-256:	4f75125b1edefa518fde15cbf6171df0429297ffe66f6f9aa189b7c642fbd712

NETWORK INFORMATION

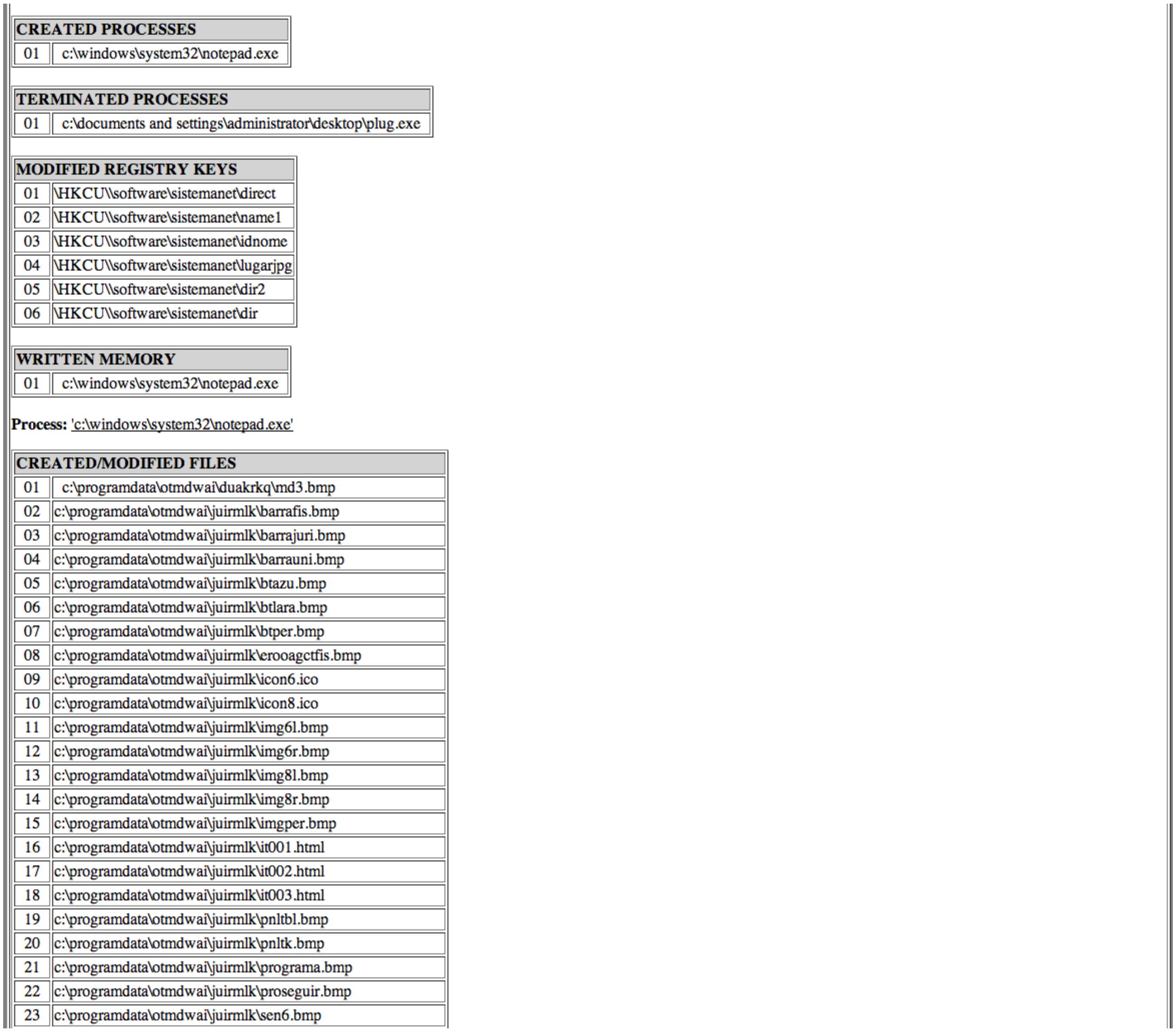
IP Address	Port	Hostname
107.22.189.127	80/TCP	ec2-107-22-189-127.compute-1.amazonaws.com
200.147.1.41	80/TCP	200-147-1-41.static.uol.com.br

OS ACTIVITY

Process: 'c:\documents and settings\administrator\desktop\plug.exe'

CREATED/MODIFIED FILES	
01	c:\programdata\otmdwai\cvdaaux\scpfywp.exe

DELETED FILES	
01	c:\windows\system32\ntdll.dll
02	c:\windows\system32\config\software.log
03	c:\documents and settings\administrator\ntuser.dat.log
04	c:\programdata\otmdwai\cvdaaux\scpfywp.exe
05	c:\documents and settings\administrator\desktop\plug.exe



CREATED PROCESSES	
01	c:\windows\system32\notepad.exe

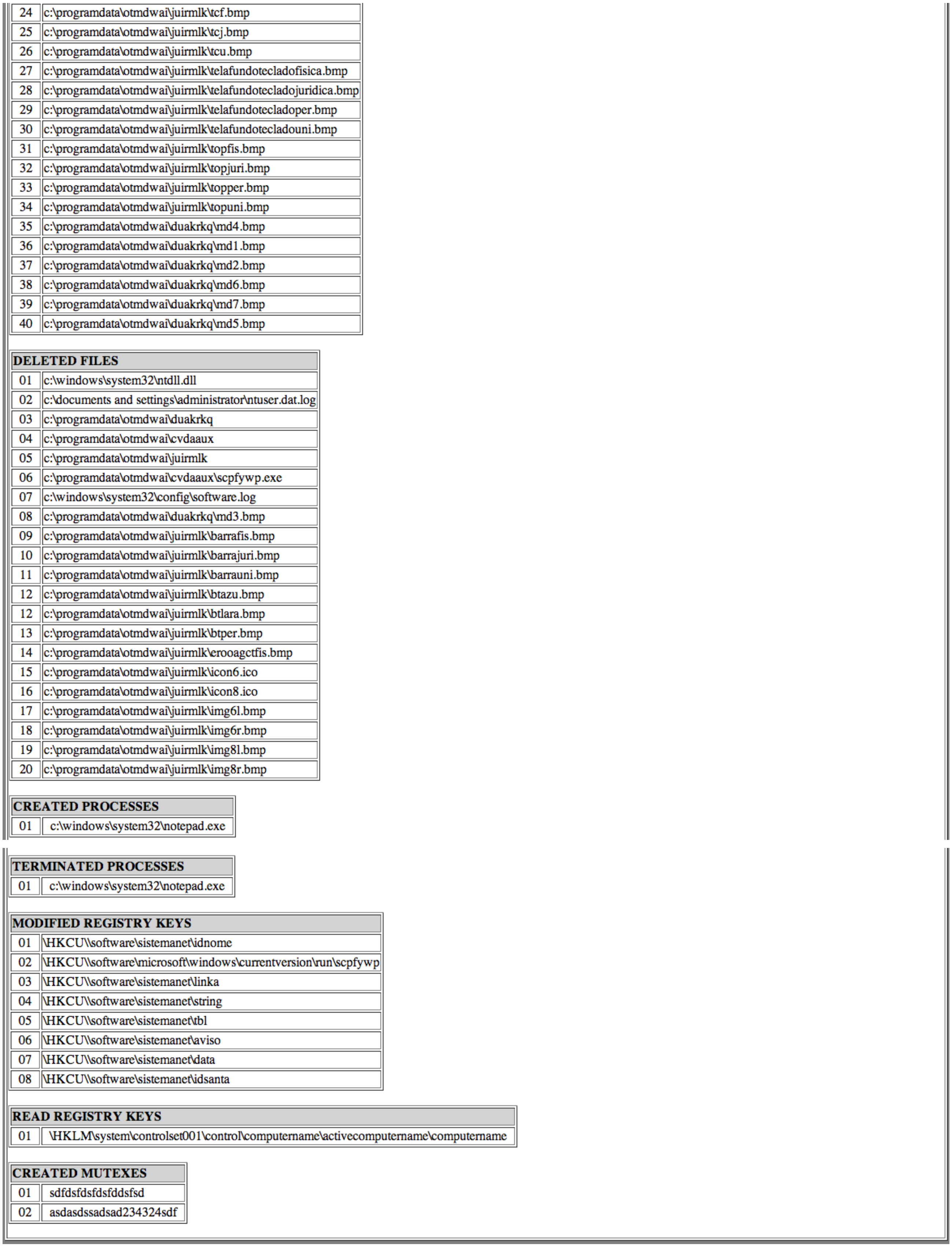
TERMINATED PROCESSES	
01	c:\documents and settings\administrator\desktop\plug.exe

MODIFIED REGISTRY KEYS	
01	\HKCU\\software\sistemanet\direct
02	\HKCU\\software\sistemanet\name1
03	\HKCU\\software\sistemanet\idnome
04	\HKCU\\software\sistemanet\lugarjpg
05	\HKCU\\software\sistemanet\dir2
06	\HKCU\\software\sistemanet\dir

WRITTEN MEMORY	
01	c:\windows\system32\notepad.exe

Process: 'c:\windows\system32\notepad.exe'

CREATED/MODIFIED FILES	
01	c:\programdata\otmdwai\duakrkq\md3.bmp
02	c:\programdata\otmdwai\juirmlk\barrafis.bmp
03	c:\programdata\otmdwai\juirmlk\barrajuri.bmp
04	c:\programdata\otmdwai\juirmlk\barrauni.bmp
05	c:\programdata\otmdwai\juirmlk\btazu.bmp
06	c:\programdata\otmdwai\juirmlk\btlara.bmp
07	c:\programdata\otmdwai\juirmlk\btper.bmp
08	c:\programdata\otmdwai\juirmlk\erooagctfis.bmp
09	c:\programdata\otmdwai\juirmlk\icon6.ico
10	c:\programdata\otmdwai\juirmlk\icon8.ico
11	c:\programdata\otmdwai\juirmlk\img6l.bmp
12	c:\programdata\otmdwai\juirmlk\img6r.bmp
13	c:\programdata\otmdwai\juirmlk\img8l.bmp
14	c:\programdata\otmdwai\juirmlk\img8r.bmp
15	c:\programdata\otmdwai\juirmlk\imgper.bmp
16	c:\programdata\otmdwai\juirmlk\it001.html
17	c:\programdata\otmdwai\juirmlk\it002.html
18	c:\programdata\otmdwai\juirmlk\it003.html
19	c:\programdata\otmdwai\juirmlk\pnltbl.bmp
20	c:\programdata\otmdwai\juirmlk\pnltk.bmp
21	c:\programdata\otmdwai\juirmlk\programa.bmp
22	c:\programdata\otmdwai\juirmlk\proseguir.bmp
23	c:\programdata\otmdwai\juirmlk\sen6.bmp



24	c:\programdata\otmdwai\juirmlk\tcf.bmp
25	c:\programdata\otmdwai\juirmlk\tcj.bmp
26	c:\programdata\otmdwai\juirmlk\tcu.bmp
27	c:\programdata\otmdwai\juirmlk\telafundotecladofisica.bmp
28	c:\programdata\otmdwai\juirmlk\telafundotecladojuridica.bmp
29	c:\programdata\otmdwai\juirmlk\telafundotecladoper.bmp
30	c:\programdata\otmdwai\juirmlk\telafundotecladouni.bmp
31	c:\programdata\otmdwai\juirmlk\topfis.bmp
32	c:\programdata\otmdwai\juirmlk\topjuri.bmp
33	c:\programdata\otmdwai\juirmlk\topper.bmp
34	c:\programdata\otmdwai\juirmlk\topuni.bmp
35	c:\programdata\otmdwai\duakrkq\md4.bmp
36	c:\programdata\otmdwai\duakrkq\md1.bmp
37	c:\programdata\otmdwai\duakrkq\md2.bmp
38	c:\programdata\otmdwai\duakrkq\md6.bmp
39	c:\programdata\otmdwai\duakrkq\md7.bmp
40	c:\programdata\otmdwai\duakrkq\md5.bmp

DELETED FILES	
01	c:\windows\system32\ntdll.dll
02	c:\documents and settings\administrator\ntuser.dat.log
03	c:\programdata\otmdwai\duakrkq
04	c:\programdata\otmdwai\cvdaaux
05	c:\programdata\otmdwai\juirmlk
06	c:\programdata\otmdwai\cvdaaux\scpfywp.exe
07	c:\windows\system32\config\software.log
08	c:\programdata\otmdwai\duakrkq\md3.bmp
09	c:\programdata\otmdwai\juirmlk\barrafis.bmp
10	c:\programdata\otmdwai\juirmlk\barrajuri.bmp
11	c:\programdata\otmdwai\juirmlk\barrauni.bmp
12	c:\programdata\otmdwai\juirmlk\btazu.bmp
12	c:\programdata\otmdwai\juirmlk\btlara.bmp
13	c:\programdata\otmdwai\juirmlk\btper.bmp
14	c:\programdata\otmdwai\juirmlk\erooagctfis.bmp
15	c:\programdata\otmdwai\juirmlk\icon6.ico
16	c:\programdata\otmdwai\juirmlk\icon8.ico
17	c:\programdata\otmdwai\juirmlk\img6l.bmp
18	c:\programdata\otmdwai\juirmlk\img6r.bmp
19	c:\programdata\otmdwai\juirmlk\img8l.bmp
20	c:\programdata\otmdwai\juirmlk\img8r.bmp

CREATED PROCESSES	
01	c:\windows\system32\notepad.exe

TERMINATED PROCESSES	
01	c:\windows\system32\notepad.exe

MODIFIED REGISTRY KEYS	
01	\HKCU\\software\sistemanet\idnome
02	\HKCU\\software\microsoft\windows\currentversion\run\scpfywp
03	\HKCU\\software\sistemanet\linka
04	\HKCU\\software\sistemanet\string
05	\HKCU\\software\sistemanet\tbl
06	\HKCU\\software\sistemanet\aviso
07	\HKCU\\software\sistemanet\data
08	\HKCU\\software\sistemanet\idsanta

READ REGISTRY KEYS	
01	\HKLM\system\controlset001\control\computername\activecomputername\computername

CREATED MUTEXES	
01	sdfdsfdsfdsfddsfsd
02	asdasdssadsad234324sdf

Bibliography

[1] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC ’06, pages 41–52, New York, NY, USA, 2006. ACM.

[2] abuse.ch. Spyeye tracker. `https://spyeyetracker.abuse.ch`, 2011. Accessed on June, 2012.

[3] abuse.ch. Zeus tracker. `https://zeustracker.abuse.ch/`, 2011. Accessed on June, 2012.

[4] Leonard M. Adleman. An Abstract Theory of Computer Viruses. In *Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology*, pages 354–374, London, UK, 1990. Springer-Verlag.

[5] Vitor Monte Afonso, Dario Simões Fernandes Filho, André Ricardo Abed Grégio, Paulo Lício de Geus, and Mario Jino. A hybrid framework to analyze web and os malware. In *Proceedings of the IEEE Internactional Conference of Communications (ICC)*, Ottawa, Canada, June 2012.

[6] Edward Amoroso. *Fundamentals of Computer Security Technology.* Prentice-Hall PTR, 1^{st} edition, April 1994.

[7] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Electronic Systems Division, USAF, L. G. Hanscom Field, Bedford, Massachusetts, October 1972.

[8] Taimur Aslam, Ivan Krsul, and Eugene Spafford. Use of A Taxonomy of Security Faults. In *Proceedings of the 19th National Information Systems Security Conference*, pages 551–560, 1996.

[9] AVG. Avg antivirus. `http://www.avg.com`, 2012.

[10] Avira. Avira antivirus. `http://www.avira.com`, 2012.

[11] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario. The Blaster Worm: Then and Now. *Security Privacy, IEEE*, 3(4):26–31, July/August 2005.

[12] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID '07)*, pages 178–197, Gold Coast, Australia, September 2007.

[13] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, Giovanni Vigna, et al. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[14] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, pages 404–417, 2006.

[15] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS)*, 2009.

[16] BBC. Stuxnet worm hits Iran nuclar plant staff computers. `http://www.bbc.co.uk/news/world-middle-east-11414483`. Acessed on March 9, 2012, September 2010.

[17] Philippe Beaucamps. Advanced Polymorphic Techniques. In *Proceedings of World Academy of Science, Engineering and Technology*, volume 25, 2007.

[18] Michael Becher, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Security and Privacy Symposium*, pages 96–111, May 2011.

[19] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. Duqu: A Stuxnet-like malware found in the wild. Technical report, Laboratory of Cryptography and System Security (CrySyS), Department of Telecommunications, Budapest University of Technology and Economics, Hungaria, October 2011.

[20] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the Analysis of the Zeus Botnet Crimeware Toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38, August 2010.

[21] Matt Bishop. Vulnerability Analysis: an Extended Abstract. In *Recent Advances in Intrusion Detection*, 1999.

[22] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 1^{st} edition, December 2002.

[23] Martin Boldt, Bengt Carlsson, and Andreas Jacobsson. Exploring Spyware Effects. In *Nordic Workshop on Secure IT Systems (NORDSEC)*, Helsinki, Finland, 2004.

[24] Vesselin Bontchev. Possible Macro Virus Attacks and How to Prevent Them. *Computers & Security*, 15:595–626, 1996.

[25] Vesselin Bontchev. Current Status of the CARO Malware Naming Scheme. In *Virus Bulletin (VB2005)*, Dublin, Ireland, 2005.

[26] S. Buehlmann and C. Liebchen. Joebox: a secure sandbox application for windows to analyse the behaviour of malware. `http://www.joebox.org`, 2012.

[27] Armin Buescher, Felix Leder, and Thomas Siebert. Banksafe information stealer detection inside the web browser. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 262–280, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Pedro H. Calais, Douglas E. V. Pires, Dorgival Olavo Guedes, Wagner Meira, Cristine Hoepers, and Klaus Steding-jessen. A campaign-based characterization of spamming strategies. In *Proceedings of the Fifth Conference on Email and Anti-Spam (CEAS)*, 2008.

[29] Kevin Zhijie Chen, Guofei Gu, Jose Nazario, Xinhui Han, and Jianwei Zhuge. WebPatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 2011 ACM Symposium on Information, Computer, and Communication Security (ASIACCS'11)*, March 2011.

[30] Thomas Chen and Jean-Marc Robert. The Evolution of Viruses and Worms. In William W. S. Chen, editor, *Statistical Methods in Computer Security*, pages 265–282. CRC Press, 2004.

[31] Xu Chen, Jon Andersen, Z. Morley, Mao Michael, and Bailey Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[32] Chia Yuan Cho, Juan Caballero, Chris Grier, Vern Paxson, and Dawn Song. Insights from the inside: a view of botnet management from infiltration. In *Proceedings of the 3^{rd} USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[33] ClamAV. Clam antivirus. `http://www.clamav.net`, 2012.

[34] Fred Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1986.

[35] Fred Cohen. Computer Viruses: Theory and Experiments. *Computers & Security*, 6:22–35, February 1987.

[36] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual reverse engineering of binary and data files. In *Proceedings of the 5th international workshop on Visualization for Computer Security*, VizSec '08, pages 1–17, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Peter Coogan. Spyeye bot versus zeus bot. `http://www.symantec.com/connect/blogs/spyeye-bot-versus-zeus-bot`, February 2010. Accessed on May, 2012.

[38] Evan Cooke, Farnam Jahanian, and Danny McPherson. The zombie roundup: understanding, detecting, and disrupting botnets. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, SRUTI'05, pages 39–44, Berkeley, CA, USA, 2005. USENIX Association.

[39] M. Cova, C. Kruegel, and G. Vigna. There is No Free Phish: An Analysis of "Free" and Live Phishing Kits. In *Proceedings of the USENIX Workshop On Offensive Technologies (WOOT)*, San Jose, CA, August 2008.

[40] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[41] Cult of the Dead Cow. Back Orifice Windows Remote Administration Tool. `http://www.cultdeadcow.com/tools/bo.html`. Accessed on March 5, 2012, 1998.

[42] David Dagon, Guofei Gu, Christopher P. Lee, and Wenke Lee. A Taxonomy of Botnet Structures. In *23rd Annual Computer Security Applications Conference (ACSAC)*, pages 325–339, December 2007.

[43] Anthony Desnos, Eric Filiol, and Ivan Lefou. Detecting (and creating !) a hvm rootkit (aka bluepill-like). *Journal in Computer Virology*, 7(1):23–49, February 2011.

[44] DHS. A roadmap for cybersecurity research. http://www.cyber.st.dhs.gov/docs/DHS-Cybersecurity-Roadmap.pdf, 2009.

[45] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[46] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[47] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, 2009.

[48] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2007. USENIX Association.

[49] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys*, 44(2), February 2012.

[50] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, November 1992.

[51] F-Secure Corp. The trojan money spinner, 2007. Available at `http://www.f-secure.com/weblog/archives/VB2007_TheTrojanMoneySpinner.pdf`.

[52] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical report, Symantec Security Response, February 2011.

[53] Hanno Fallmann, Gilbert Wondracek, and Christian Platzer. Covertly probing underground economy marketplaces. In *Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2010.

[54] Eric Filiol. *Computer viruses: from theory to applications*. Springer, 2005.

[55] Eric Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2(1):35–50, 2006.

[56] Eric Filiol, Grégoire Jacob, and Mickaël Le Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3(1):23–37, 2007.

[57] Fitsec. Tool release: A banking trojan detection tool. `http://www.fitsec.com/blog/index.php/2011/08/15/tool-release-a-banking-trojan-detection-tool/`, 2011. Accessed on June, 2012.

[58] FProt. F-prot antivirus. `http:\\www.f-prot.com`, 2012.

[59] Lee Garber. Melissa Virus Creates a New Type of Threat. *Computer*, 32:16–19, June 1999.

[60] Jonathon Giffin, Somesh Jha, and Barton Miller. Automated discovery of mimicry attacks. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[61] Google. Google maps api. `https://developers.google.com/maps/`, 2012. Acessed on June, 2012.

[62] Sarah Gordon. What a (winword.)concept. Virus Bulletin, September 1995.

[63] André Grégio, Dario S. Fernandes, Vitor Afonso, Rafael Santos, Mario Jino, and Paulo L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. In *Proc. of the SPIE*, volume 8059, 2011.

[64] AndrÃ© Ricardo Abed Gregio, Isabela L. Oliveira, Rafael Duarte Coelho dos Santos, Adriano M. Cansian, and Paulo L. de Geus. Malware distributed collection and pre-classification system using honeypot technology. In *Proceedings of SPIE*, volume 7344. Data Mining, Intrusion Detection, Information Security and Assurance, and Data Networks Security 2009., SPIE, 2009.

[65] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43, 2005.

[66] Harold Joseph Highland. A Macro Virus. *Computers & Security*, 8:178–188, 1989.

[67] Hispasec. Virustotal. `http://www.virustotal.com/`, 2012.

[68] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: a case-study of keyloggers and dropzones. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 1–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[69] John D. Howard and Thomas A. Longstaff. A Common Language for Computer Security Incidents. Technical Report SAND98-8667, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, October 1998.

[70] IBM. The cost of a data breach. *IBM Systems Magazine*, pages 14–15, September/October 2011.

[71] Claudiu Ilioiu. What is trojan banking, computer virus designed to attack online transactions. `http://www.financial-magazine.org/what-is-trojan-banking-computer-virus-designed-to-attack-online-transactions-66473.html`, May 2012. Accessed on May, 2012.

[72] iSecLab. Anubis - analyzing unknown binaries. `http://anubis.iseclab.org/`, 2007.

[73] Gregoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Ninth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[74] Grégoire Jacob, Hervé Debar, and Eric Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 81–100, Berlin, Heidelberg, 2009. Springer-Verlag.

[75] Gregoire Jacob, Eric Filiol, and Herve Debar. Functional polymorphic engines: formalisation, Implementation and use cases. *J. Comput. Virol.*, 5(3), 2008.

[76] Gregoire Jacob, Matthias Neugschwandtner, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. Technical Report 2010-26, UCSB, November 2010.

[77] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31:264–323, 1999.

[78] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[79] Martin Karresand. Separating Trojan Horses, Viruses and Worms - A Proposed Taxonomy of Software Weapons. In *IEEE Information Assurance Workshop*, pages 127–134, 2003.

[80] Kaspersky. Number of the week: 780 new malicious programs designed to steal users' online banking data detected every day, 2012. Available at `http://www.kaspersky.com/about/news/virus/2012/Number_of_the_week_780_new_malicious_programs`.

[81] Brendan P. Kehoe. Zen and the Art of the Internet. `https://www.cs.indiana.edu/docproject/zen/zen-1.0_toc.html`, January 1992.

[82] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *J. Comput. Virol.*, 7(4):233–245, November 2011.

[83] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareBox: Efficient Malware Analysis on Bare-Metal. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, December 2011.

[84] Brian Krebs. 'Citadel' Trojan Touts Trouble-Ticket System. `http://krebsonsecurity.com/2012/01/citadel-trojan-touts-trouble-ticket-system/`. Accessed on March 9, 2012, January 2012.

[85] Ivan Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, May 1998.

[86] Christopher Kruegel, Engin Kirda, and Ulrich Bayer. TTAnalyze: A tool for analyzing malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

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

[88] Eduardo Labir. VX Reversing II, Sasser.B. *The Code Breakers Journal*, 1(1), 2004.

[89] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26:211–254, September 1994.

[90] Tobias Lauinger, Engin Kirda, and Pietro Michiardi. Paying for piracy? an analysis of one-click hosters' controversial reward schemes. In *15^{th} Internationa Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012.

[91] Kirill Levchenko, Andreas Pitsillidis, Neha Chachra Br, Tristan Halvorson, Chris Kanich, Christian Kreibich, He Liu, Damon Mccoy, Nicholas Weaver, Vern Paxson, Geoffrey M. Voelker, and Stefan Savage. Click trajectories: End-to-end analysis of the spam value chain. In *In Proceedings of IEEE Symposium on Security & Privacy*, pages 431–446, 2011.

[92] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, Menlo Park, CA, USA, September 2011.

[93] Ulf Lindqvist and Erland Jonsson. How to Systematically Classify Computer Security Intrusions. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 154–163, Washington, DC, USA, 1997. IEEE Computer Society.

[94] Carl Linnaeus. *Systema naturae, sive regna tria naturae systematice proposita per classes, ordines, genera, & species.* 1^{st} edition, 1735.

[95] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow Attacks: Automatically Evading System-Call-Behavior based Malware Detection. *Springer Journal in Computer Virology*, 2012.

[96] Mariette Manktelow. History of Taxonomy. Department of Systematic Biology, Uppsala University. `http://atbi.eu/summerschool/files/summerschool/Manktelow_Syllabus.pdf`. Acesso realizado em 27 de fevereiro de 2012, 2008.

[97] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] Eric Medvet, Engin Kirda, and Christopher Kruegel. Visual-similarity-based phishing detection. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, SecureComm '08, pages 22:1–22:6, New York, NY, USA, 2008. ACM.

[99] Charles D. Michener, John O. Corliss, Richard S. Cowan, Peter H. Raven, Curtis W. Sabrosky, Donald F. Squires, and G. W. Wharton. Systematics in Support of Biological Research. Technical report, Division of Biology and Agriculture, National Research Council, Washington, DC, January 1970.

[100] Microsoft. Win32/allaple, 2007. Available at `http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Win32%2fAllaple`.

[101] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *Security Privacy, IEEE*, 1(4):33–39, July/August 2003.

[102] David Moore, Colleen Shannon, and Jeffery Brown. 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.

[103] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. of the 23rd Annual Computer Security Applications Conference*, ACSAC '07, pages 421 –430, December 2007.

[104] Mozilla. Rhino: Javascript for java. http://www.mozilla.org/rhino, 2012. Accessed on May, 2012.

[105] Lysa Myers. Aim for Bot Coordination. In *Virus Bulletin (VB2006)*, pages 35–37, Montreal, Canada, October 2006.

[106] J. Nazario. Phoneyc: A virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, pages 6–6. USENIX Association, 2009.

[107] Matthew Nelson. Developer creates the first Java virus and names it 'Strange Brew'. `http://www.javaworld.com/javaworld/jw-09-1998/jw-09-iw-virus.html`. Accessed on March 5, 2012, August 1998.

[108] John Von Neumann and Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois, 1^{st} edition, 1966.

[109] Peter G. Neumann and Donn B. Parker. A Summary of Computer Misuse Techniques. In *12th National Computer Security Conference, Baltimore, MD*, pages 396–406, October 1989.

[110] NHM. What is Taxonomy. UK Natural History Museum. `http://www.nhm.ac.uk/nature-online/science-of-natural-history/taxonomy-systematics/what-is-taxonomy/index.html`. Acesso realizado em 27 de fevereiro de 2012, 2012.

[111] Norman. Norman sandbox whitepaper. `http://download.norman.no/whitepapers/whitepaper\_Norman\_SandBox.pdf`, 2003.

[112] J. C. Oliveros. VENNY. An interactive tool for comparing lists with Venn Diagrams. `http://bioinfogp.cnb.csic.es/tools/venny/index.html`. Acessed on September 13, 2012, 2007.

[113] Bob Page. A Report on the Internet Worm. `http://www.ee.ryerson.ca/~elf/hack/iworm.html`, 1988.

[114] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45:1–45:4, New York, NY, USA, 2010. ACM.

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

[116] Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. A Multi-perspective Analysis of the Storm (Peacomm) Worm. Technical report, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, October 2007.

[117] Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. Technical report, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, February 2009.

[118] Niels Provos. Honeyd: a virtual honeypot daemon, 2003. Available at `http://www.citi.umich.edu/u/provos/papers/honeyd-eabstract.pdf`.

[119] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley Professional, first edition, 2007.

[120] D.A. Quist and L.M. Liebrock. Visualizing compiled executables for malware analysis. In *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, pages 27 –32, oct. 2009.

[121] Danny Quist and Val Smith. Detecting the Presence of Virtual Machines Using the Local Data Table, 2008. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.

[122] Nguyen Anh Quynh and Kuniyasu Suzaki. Virt-ice: Next-generation debugger for malware analysis. `https://media.blackhat.com/bh-us-10/whitepapers/Anh/BlackHat-USA-2010-Anh-Virt-ICE-wp.pdf`, 2010.

[123] Costin Raiu. A Virus by Any Other Name: Virus Naming Practices. `http://www.symantec.com/connect/articles/virus-any-other-name-virus-naming-practices`. Accessed on March 1, 2012, June 2002.

[124] Huw Read, Konstantinos Xynos, and Andrew Blyth. Presenting devise: data exchange for visualizing security events. *IEEE Comput. Graph. Appl.*, 29(3):6–11, May 2009.

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

[126] Konrad Rieck, Guido Schwenk, Tobias Limmer, Thorsten Holz, and Pavel Laskov. Botzilla: detecting the "phoning home" of malicious software. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1978–1984, New York, NY, USA, 2010. ACM.

[127] Joanna Rutkowska. Introducing Stealth Malware Taxonomy. `http://invisiblethings.org/papers/malware-taxonomy.pdf`. Acesso realizado em 28 de fevereiro de 2012, 2006.

[128] Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. Measurement and Analysis of Spyware in a University Environment. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–153, 2004.

[129] Bruce Schneier. The zotob storm. *Security Privacy, IEEE*, 3(6):96, November/December 2005.

[130] Seculert. Citadel - An Open Source Malware Project. `http://blog.seculert.com/2012/02/citadel-open-source-malware-project.html`. Accessed on March 9, 2012, February 2012.

[131] SecureList. Net-worm.win32.allaple.a, 2007. Available at `http://www.securelist.com/en/descriptions/old145521`.

[132] C. Seifert and R. Steenson. Capture - honeypot client (capture-hpc), 2006.

[133] Justin Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers.* No Starch Press, San Francisco, CA, USA, 2009.

[134] Madhu Shankarapani, Subbu Ramamoorthy, Ram Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *J. Comput. Virol.*, 7:107–119, 2011.

[135] Seungwon Shin and Guofei Gu. Conficker and Beyond: a Large-Scale Empirical Study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 151–160, New York, NY, USA, 2010. ACM.

[136] Seungwon Shin, Raymond Lin, and Guofei Gu. Cross-analysis of botnet victims: New insights and implications. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11))*, September 2011.

[137] John F. Shoch and Jon A. Hupp. The Worm Programs—Early Experience with a Distributed Computation. *Communications of the ACM*, 25:172–180, March 1982.

[138] R. Shyamasundar, Harshit Shah, and N. Kumar. Malware: From Modelling to Practical Detection. In Tomasz Janowski and Hrushikesha Mohanty, editors, *Distributed Computing and Internet Technology*, volume 5966 of *Lecture Notes in Computer Science*, pages 21–39. Springer Berlin / Heidelberg, 2010.

[139] Garry Sidaway. The Rise and Rise of Bot Networks. *Network Security*, 2005(5):19–20, May 2005.

[140] Ed Skoudis and Lenny Zeltser. *Malware: Fighting Malicious Code.* Prentice Hall, 1st edition, 2003.

[141] Fridrik Skulason, Alan Solomon, and Vesselin Bontchev. CARO Naming Scheme. `http://www.caro.org/naming/scheme.html`. Accessed on March 1, 2012, 1991.

[142] Eugene H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, November 1988.

[143] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[144] B. Stone-Gross, R. Abman, R. Kemmerer, C. Kruegel, D. Steigerwald, and G. Vigna. The Underground Economy of Fake Antivirus Software. In *Proceedings of the Workshop on Economics of Information Security (WEIS)*, 2011.

[145] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[146] B. Stone-Gross, R. Stevens, A. Zarras, R. Kemmerer, C. Kruegel, and G. Vigna. Understanding Fraudulent Activities in Online Ad Exchanges. In *Proceedings of the Internet Measurement Conference (IMC)*, 2011.

[147] Brett Stone-Gross, Marco Cova, Bob Gilbert, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Analysis of a Botnet Takeover. *IEEE Security & Privacy Magazine*, 9(1):64–72, January/February 2011.

[148] G. Stringhini, M. Egele, C. Kruegel, and G. Vigna. Poultry markets: On the underground economy of twitter followers. In *Workshop on Online Social Network (WOSN)*. ACM, 2012.

[149] G. Stringhini, C. Kruegel, and G. Vigna. Detecting spammers on social networks. In *Annual Computer Security Applications Conference*, 2010.

[150] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 1^{st} edition, February 2005.

[151] Sapon Tanachaiwiwat and Ahmed Helmy. VACCINE: War of the Worms in Wired and Wireless Networks. In *Proceedings of the IEEE INFOCOM*, Barcelona, Spain, April 2006.

[152] The Honeynet Project. Nepenthes - the finest collection, 2008. Available at `http://www.honeynet.org/project/nepenthes`.

[153] The Honeynet Project. Dionaea: catches bugs, 2009. Available at `http://dionaea.carnivore.it/`.

[154] Kurt Thomas and David M. Nicol. The Koobface Botnet and the Rise of Social Malware. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 63–70, October 2010.

[155] Threat Expert Ltd. Threatexpert. `http://www.threatexpert.com`, 2012. Accessed on May, 2012.

[156] TrendMicro. The ZLOB Show: Trojan Poses as Fake Video Codec, Loads More Threats. `http://about-threats.trendmicro.com/archivevulnerability.aspx?language=us&name=The+ZLOB+Show:+Trojan+Poses+as+Fake+Video+Codec,+Loads+More+Threats`. Accessed on March 8, 2012, 2012.

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

[158] UPI. New malware attacks target online banking. `http://www.upi.com/Science_News/2012/02/02/New-malware-attacks-target-online-banking/UPI-25351328224292/`, February 2012. Accessed on May, 2012.

[159] US Department of Homeland Security. Current hard problems in infosec research. issue 7 of 11: Combatting malware and botnets. Technical report, DHS, November 2009.

[160] VxHeavens. Virus database, 2010. Available at `http://vx.netlux.org/`.

[161] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A Taxonomy of Computer Worms. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode (WORM)*, pages 11–18, New York, NY, USA, 2003.

[162] James A. Whittaker and Andres De Vivanco. Neutralizing Windows-based Malicious Mobile Code. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 242–246, New York, NY, USA, 2002. ACM.

[163] Georg Wicherski. pehash: a novel approach to fast malware clustering. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[164] Wikipedia. Gnu diff. `http://en.wikipedia.org/wiki/Diff`, 2002.

[165] Wikipedia. The jaccard index. `http://en.wikipedia.org/wiki/Jaccard\_index`, 2005.

[166] Wikipedia. Zlob trojan, 2011. Available at `http://en.wikipedia.org/wiki/Zlob\_trojan`.

[167] Wikipedia. Systematics. `http://en.wikipedia.org/wiki/Systematics`. Acesso realizado em 27 de fevereiro de 2012, 2012.

[168] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5:32–39, March 2007.

[169] Chester Wisniewski. Malware shuts down hospital near Atlanta, Georgia. `http://nakedsecurity.sophos.com/2011/12/13/malware-shuts-down-hospital-near-atlanta-georgia/`. Accessed on April 1, 2012, December 2011.

[170] Chao Yang, Robert Harkreader, and Guofei Gu. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *Proceedings of the 14^{th} International Symposium on Recent Advances in Intrusion Detection (RAID'11))*, September 2011.

[171] Junjie Zhang, Xiapu Luo, Roberto Perdisci, Guofei Gu, Wenke Lee, and Nick Feamster. Boosting the scalability of botnet detection using adaptive traffic sampling. In *Proceedings of the 2011 ACM Symposium on Information, Computer, and Communication Security (ASIACCS'11)*, March 2011.

[172] Junjie Zhang, Christian Seifert, Jack W. Stokes, and Wenke Lee. Arrow: Generating signatures to detect drive-by downloads. In *Proceedings of the 20th International conference on World Wide Web*, WWW '11, pages 187–196, New York, NY, USA, 2011. ACM.

[173] Qinghua Zhang and D.S. Reeves. Metaaware: Identifying metamorphic malware. In *Proc. of the 23rd Annual Computer Security Applications Conference*, ACSAC '07, pages 411 –420, December 2007.

[174] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM conference on Computer and Communications Security*, CCS '02, pages 138–147, New York, NY, USA, 2002. ACM.