Αλγοριθμικές Τεχνικές Ανίχνευσης και Κατάταξης Κακόβουλου Λογισμικού βασισμένες σε Γραφήματα Κλήσεων Συναρτήσεων Συστήματος

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Μηχανικών Η/Υ & Πληροφορικής

Εξεταστική Επιτροπή

από τον

Ιωσήφ Πολενάκη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΗΝ ΘΕΩΡΙΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΕΠΤΕΜΒΡΙΟΣ 2014

Algorithmic Techniques for Malicious Software Detection and Classification based on System-Call Dependency Graphs

MSc Thesis

Department of Computer Science and Engineering

University of Ioannina

GREECE

Iosif R. Polenakis

September 2014

Αφιερωση

Η εργασία αυτή αφιερώνεται:

στην πολυαγαπημένη οικογένειά μου,

στον καθηγητή μου και επιβλέποντα της εργασίας Σταύρο Δ. Νικολόπουλο,

και σε όλους τους *συναγωνιστές στην αρένα της διανόησης* που παρά τα εμπόδια και τις όποιες δυσκολίες συνεχίζουν πιστά να παραμερίζουν τις επίγειες απολαύσεις για χάρη της συνεχούς ανιδιοτελούς προσφοράς στην επιστήμη και την κοινωνία.

Ευχαριστιες

Πρωτίστως θα ήθελα να ευχαριστήσω την πολυαγαπημένη μου οικογένεια, η οποία από τα πρώτα μου βήματα αποτέλεσε εστία ανθρωπισμού, αρετής, πνεύματος και παιδείας. Ευχαριστώ μέσα απ την καρδία μου τους γονείς μου Ρουσάγγελο και Κλεοπάτρα καθώς και την αδελφή μου Ειρήνη, για το φως το οποίο μου μεταλαμπάδευσαν και την αδιάπαυστη συμπαράστασή τους καθ' όλη της διάρκεια του αγώνα μου. Στο σημείο αυτό, θέλω να ευχαριστήσω ιδιαίτερα δυο άτομα στα οποία *αδιαμφισβήτητα οφείλω πολλά*, και τα οποία αποτέλεσαν τους πρωτεργάτες - την πρώτη γραμμή στον αγώνα - για την οικοδόμηση του *ότι κι αν είμαι* - του *ότι κι αν γίνω*, τον παππού μου Απόστολο και την γιαγιά μου Ειρήνη, που άναψαν και κράτησαν μέχρι το τέλος άσβεστη τη φλόγα για βελτίωση και ακατάπαυστη προσπάθεια εξύψωσης των ηθών και του πνεύματος, καθώς επίσης και τους εκλειπόντες Ιωσήφ και Αναστασία, τη σοφία των οποίων γεύτηκα μονάχα για λίγο.

Επιπρόσθετα, θέλω μέσα απο την καρδιά μου να ευχαριστήσω τον πνευματικό μου πατέρα, έναν αληθινό δάσκαλο με όλη την βαθύτερη αξία και ουσιαστικότερη σημασία που κρύβει αυτή η λέξη, ο οποίος δίδαξε στον μαθητή *να πιστεύει στο όραμα και στην ιδέα, και να μάχεται με αυταπάρνηση αποζητώντας το βέλτιστο ενάντια σε όσα εμπόδια κι αν εμφανιστούν, χωρίς ωστόσο να λησμονεί το από που ξεκίνησε.* Αναφέρομαι φυσικά στον επιβλέποντα της εργασίας αυτής, τον καθηγητή κύριο Σταύρο Δ. Νικολόπουλο, ο οποίος μου έκανε την τιμή να με εμπιστευτεί, στέκοντας αρωγός στην περάτωση της εργασίας αυτής. Ακόμη, θα ήθελα να ευχαριστήσω τον αναπληρωτή καθηγητή κύριο Λεωνίδα Παληό και τον επίκουρο καθηγητή κύριο Λουκά Γεωργιάδη για την συμμετοχή του στην τριμελή επιτροπή καθώς επίσης και για τις πολύτιμες και εύστοχες παρατηρήσεις τους που συνέβαλαν στην βελτίωση της εργασίας αυτής.

Τέλος, αισθάνομαι την ανάγκη να ευχαριστήσω όλους τους φίλους και συμφοιτητές για την πολύτιμη συμπαράστασή τους σε όλες τις στιγμές που περάσαμε μαζί, με τους καρπούς του *Nikote* και του *Runge* ανά χείρας, ατενίζοντας με θάρρος και ελπίδα το αβέβαιο μέλλον, στο μπαλκόνι του τρίτου ορόφου, καθώς επίσης και την *A.* της οποίας η εκούσια απουσία με έκανε πιο δυνατό. Ευχαριστώ το Θεό, που έφερε στο δρόμο μου τους παραπάνω ανθρώπους, να προσφέρουν το φώς και να αποτελούν έμπνευση για όσους έχουν την τύχη να σταθούν δίπλα τους.

Table of Contents

List of Figures

List of Tables

Abstract

Author: Joseph R. Polenakis, BSc, Dept. of Computer Science and Engineering, University of Ioannina September 2014,

Thesis Title: Algorithmic Techniques for Malicious Software Detection and Classification based on System Call Graphs

Supervisor: Stavros D. Nikolopoulos, Professor, Dept. of Computer Science and Engineering, University of Ioannina

One of the most dangerous and detrimental threats in computer security is the malicious software, the so called malware. Malware is a type of software indicated to serve a malicious purpose in some fashion, consisting a major threat for systems' security by compromising the integrity, confidentiality and availability so for the systems as whole as for the data stored into them. Thus, in order to protect our systems from such a threat, prevention and detection against malware consists a simplex. The most stable, effective and also efficient method to protect our systems against malware threats is the installation of end-point detection systems, the so called antivirus.

In order to achieve real-time protection AVs use a quite naive approach to identify malware leveraging pattern matching and utilizing a set of byte-level string signatures, expressing an adequate real-time protection. However, because this method is based on static data, the credibility of its results can be compromised during the appearance of a mutated or even more in case of a totally brand-new malware. Since we are not able to predict any brand-new malware our main target is the armoring against any mutated malware.

In this thesis we present an algorithmic technique in the area of dynamic malware analysis, in order to detect if a given specimen is a malware and afterwards to classify it into one of a set of known malware families. Specifically, we propose an elaborated algorithmic technique for malware detection and classification utilizing the System-Call Dependency Graphs (SCDG) obtained by capturing traces through tainted analysis and a set of similarity metrics methods in order to detect and classify a given specimen. More precisely, in order to achieve higher generalizability and thus higher flexibility we have made a transformation using the initial SCDG, by creating a hyper-abstraction of it, where its vertices are consisted by groups of system-calls with similar functionality. After this transformation, we proceed to the detection phase, where we have developed a formula

that combines so the examination of qualitative, as that quantitative and existential characteristics, that are spread among the members of a known malware family. Next, in the classification phase we leverage so the aforementioned characteristics utilized by various similarity metrics as the correlations between the Maximum Strongly Connected Component (MSCC) of the test sample's SCDG and each Strongly Connected Component (SCC) in each malware family member' s SCDG.

Finally, we cite the results produced from experiments when applying our model on a dataset of 2631 malware samples from 48 malware families and 33 commodity benign programs when performing 5-fold cross validation achieving a 99.64 % detection rate with 10% false-positives where our classification accuracy reaches the 82.84 %, and then evaluate our model comparing the results against those produced by other approaches.

Εκτεταμενη Περιληψη στα Ελληνικα

Μια από τις απειλές με τον μεγαλύτερο βαθμό επικινδυνότητας στον τομέας της ασφάλειας υπολογιστικών συστημάτων είναι το κακόβουλο λογισμικό (malicious software), το αποκαλούμενο malware. Το κακόβουλο λογισμικό είναι ένα είδος λογισμικού το οποίο εξυπηρετεί έναν κακόβουλο σκοπό, αποτελώντας μείζονα απειλή για την ακεραιότητα, τη διαθεσιμότητα και την εμπιστευτικότητα των συστημάτων όσο και των δεδομένων που βρίσκονται μέσα σε αυτά. Ως εκ τούτου, για να προστατευθούν τα συστήματα, και κατ' επέκταση τα δεδομένα που βρίσκονται σε αυτά, η πρόληψη και η αντιμετώπιση (ανίχνευση) συγκροτούν τη δεσπόζουσα τακτική. Η πιο αξιόπιστη και αποδοτική μέθοδος για να επιτευχθεί κάτι τέτοιο, είναι η εγκατάσταση συστημάτων ανίχνευσης σε όσο δυνατόν περισσ-ότερα σημεία του εκάστοτε συστήματος, τα αποκαλούμενα αντι-ιικά (Anti-Virus).

Τα συστήματα αυτά τα οποία είναι υπεύθυνα για την ανίχνευση κακόβουλου λογισμικού χρησιμοποιούν άμεσες μεθόδους ανίχνευσης, όπως για παράδειγμα το ταίριασμα κάποιων μοτίβων (pattern matching) βασισμένων σε υπογραφές γραμματοσειρών, επιτυγχάνοντας έτσι ικανοποιητικά ποσοστά ανίχνευσης τέτοιων απειλών σε πραγματικό χρόνο. Ωστόσο, η ικανότητά τους αυτή, λόγω της στατικότητας των δεδομένων τα οποία αξιοποιεί, δύναται να ελαχιστοποιηθεί όταν η εν λόγω απειλή αποτελείται είτε από ένα μεταλλαγμένο είτε από ένα εντελώς νέο κακόβουλο λογισμικό. Κατά συνέπεια, δεδομένου ότι δεν είναι δυνατόν να προβλέψουμε τη δημιουργία οποιουδήποτε νέου κακόβου- λου λογισμικού, ο βασικός μας στόχος είναι να αναπτύξουμε μηχανισμούς οι οποίοι να είναι ικανοί να παρέχουν προστασία ενάντια σε βελτιωμένες μορφές της απειλής αυτής, όπως για παράδειγμα το μεταλλαγμένο κακόβουλο λογισμικό.

Στην εργασία αυτή προτείνουμε, υλοποιούμε και παρουσιάζουμε, μια αλγοριθμική μέθοδο στον τομέα της δυναμικής ανάλυσης κακόβουλου λογισμικού η οποία έχει την ικανότητα, δοθέντος ενός αγνώστου λογισμικού να ανιχνεύει αν είναι κακόβουλο ή όχι, και εν συνεχεία να το ταξινομεί σε αποκλειστικά μια από ένα σύνολο γνωστών οικογενειών κακόβουλων λογισμικών. Συγκεκριμένα, προτείνουμε μια αλγοριθμική τεχνική για την αναγνώριση και ταξινόμηση κακόβουλων λογισμικών βασισμένη σε γραφήματα κλήσεων συναρτήσεων συστήματος (System-Call Dependency Graphs) τα οποία δημιουργήθηκαν αξιοποιώντας δεδομένα τα οποία καταγράφηκαν κατά την εκτέλεση των κακόβουλων λογισμικών μέσω μιας διαδικασίας που ονομάζεται εκτεταμένη ανάλυση (taint analysis).

Πιο συγκεκριμένα, προκειμένου να επιτύχουμε μεγαλύτερη ικανότητα γενίκευσης ενάντια σε ισχυρές μεταλλάξεις δημιουργούμε ένα υπέρ-γράφημα το οποίο δρα ως υπέρ-γενίκευση του γραφήματος κλήσεων συναρτήσεων συστήματος όπου έχουμε αντικαταστήσει κάθε

κόμβο του (συνάρτηση συστήματος) με την ομάδα στην οποία ανήκει αυτή η συνάρτηση συστήματος και η οποία συμπεριλαμβάνει και άλλες συναρτήσεις συστήματος με όμοια λειτουργικότητα. Εν συνεχεία, για την αναγνώριση του κακόβουλου λογισμικού προτείνουμε μια μέθοδο η οποία στηρίζεται σε μια συσχέτιση που συνδυάζει την αξιοποίηση των ποιοτικών, ποσοτικών και υπαρξιακών (αναφορικά με τις ακμές) χαρακτηριστικών που υπάρχουν στα γραφήματα κλήσεων συναρτήσεων συστήματος των μελών μιας οικογένειας κακόβουλων λογισμικών μέσω διαφορετικών μετρικών ομοιότητας. Τέλος, για την κατάταξη ενός κακόβουλου λογισμικού σε μία οικογένεια κακόβουλων λογισμικών, αξιοποιούμε ξανά τα προαναφερθέντα χαρακτηριστικά μέσω μετρικών ομοιότητας και επιπρόσθετα εκμεταλλευόμαστε την συσχέτιση σε επίπεδο Ισχυρά Συνεκτικών Συνιστωσών που παρατηρείται ανάμεσα στο γράγημα του αγνωστου δείγματος και το γράφημα ενός μέλους μιας οικογένειας κακόβουλων λογισμικών.

Επιπρόσθετα, παραθέτουμε τα αποτελέσματα που εξήχθησαν μέσω αποτίμησης διασταυρωμένης σε πέντε τμήματα (5-fold cross validation) εφαρμόζοντας το μοντέλο μας σε 2631 κακόβουλα λογισμικά από 48 οικογένειες κακόβουλων λογισμικών και 33 μη-κακόβουλα λογισμικά, επιτυγχάνοντας 99.64 % ποσοστό αναγνώρισης με 10 % εσφαλμένες ανιχνεύσεις (false-positives) ενώ το ποσοστό ορθής κατάταξης ενός κακόβουλου λογισμικού σε μια οικογένεια κακόβουλων λογισμικών έφτασε το 82.84 %. Τέλος παραθέτουμε μια σύγκριση των αποτελεσμάτων του μοντέλου μας με άλλα μοντέλα βασισμένα σε γραφήματα και μη, σχολιάζοντας και συγκρίνοντάς τα μεταξύ τους.

Chapter 1

Introduction

In this chapter we make an introduction to the topic citing the basic definitions and explaining briefly the basic methods applied for protection against malicious software. To start with, we define the term of malicious software, and then we proceed with the definition of some of the most common malware categories. Next we present the main procedures that are applied in order to develop later some defense techniques with strong theoretical background. We define the procedures of analysis, detection and classification while we describe their main corpus. Later, we define our main motivation for this research, where we describe the detection evasion practices and specifically the mutation procedures that are applied by malware authors in order to evade detection. Finally, we make a brief introduction to our proposed model describing in a very abstract level the method that we have designed and developed in order to detect any mutated malware.

1.1 What is Malicious Software

Malicious Software or malware is any kind of software that its functionality is to cause harm to a user, computer, or network [56] . Thus, any software with malicious purposes can be considered malware. Malware in most of the cases, exhibits a very typical structure so in its programming aspect as in its overall organization. Across the literature, continually we meet the most common description of malware's structure where malware is presented to dispose a payload that we could envisage it as a kernel and additionally the reproduction and cloning instructions and in some cases the propagation instructions that could be envisioned as a cover respectively.

Generally speaking, malware can be considered as the entity in which new features can be easily added to enhance its dark side effects in the form of various attacks [40]. So, according to this consideration malware has to be treated as an alive harmful organism with the ability of evolution during time just like the biological bacteria. The addition, subtraction, modification or any other kind of mutation in a malware, is able to generate a totaly brand new malware that either serves new purposes, or lacks of bugs or even to be just a new variation that can not be detected from a malware detection system.

Malware can be utilized for a variety of unethical purposes. Starting from early nineties, where malware was just a tool for self projection of malware author's programming skills, to nowadays, where we are facing a plenty of examples ranging from economical benefit from personal information stealing, to cyber warfare, malware remains an extremely dangerous tool in any wrong hands.

1.1.1 Basic Malware Types

In malware categorization there are several methods that someone can classify a malware. In example, an observer can classify a malware according to its propagation method, known as propagation vector, where certain malware have a specific method to propagate them selves in contrast with others that do not have this characteristic. On the other hand, a second observer could classify a malware based upon its functionality. Thus, subsequently, we present the most common categorization which is based upon the propagation vector criteria as presented in [28, 59] and later we proceed by presenting a more elaborated categorization based on malware's functionality [56, 59].

- **Virus:** In this category we meet the most common and well-known malware type, the *Viruses*. A computer virus is a small program with harmful intent that its main characteristic is that its operational mode is to replicates itself when inserted into an executable file (.COM, .EXE or .PE). As we will refer to next chapters, a virus has the ability to evolve to new variants by modifying itself, a phenomenon called metamorphism. Just like the biological viruses, a computer virus need an existing host program in order to cause harm to the infected system. In this manner, a program that is infected containing the virus consequently infects any system that executes it. The most common method that a computer virus is implemented, in order to invade into a computer system, is to be attached to some software utility such as a word processing application which when launched triggers the virus to be activated then replicate itself attaching it to other hosts and so on, ending by executing its payload.

- **Worm:** In the second category we have the computer *worms*. A worm, unlike viruses, replicates itself by executing its own code independently of any other program without the need of any host program in order to cause harm. The main differences between worms and viruses is firstly that worms are host independent and secondly that viruses, in the vast majority of the cases, are spread among the files stored in an infected system in contrast with worms that propagate, among the systems of a network infecting as many computers as possible by sending them selves via network connection, with both of these two characteristics making worms more dangerous than viruses.

- **Trojan Horse:** This category, unlike the previous two, includes a very fuzzy group, the *Trojan Horses*. A Trojan Horse is a type of malware that malware author has embedded it in an application. In most of the cases, trojan horses are associated with the access and the sending of unauthorized information from the system that they infect to their malware author or another entity, which is a characteristic that classifies trojan horses as spyware. In general cases, its functionality is the emulation of a legal application in order to gain remote access to a system. However, cases of system damage such as data loss, are not excluded as in many cases, trojan horses are employed in Denial Of Service (DOS) attacks.

1.1.2 Miscellaneous Malware Types

To this point we step beyond the basic types of malware, presenting other categories that are also widespread *in the wild*, possessing a large portion of the types of malware families. Next, we list some of the most known malware categories. However, we ought to notice that a malware many times can belong to more than one categories according to malware author's intents.

- **AdWare:** When a malware has infected a computer, *advertising-supported software* automatically displays or downloads advertisements.

- **Backdoor:** A *backdoor* is a type of self-installed malicious code that allows to attacker access to an infected system. Backdoors exhibit an auxiliary functionality since they are utilized by malware authors in order to gain remote access with little or no authentication and then to execute commands on the infected system.

- **Botnet:** Generally speaking, *botnet* is a type of malware that infects a group of computers and then turns them into *zombies* under the *botmasters* possession. A malware belonging to this category could be a worm or a trojan. Additionally we ought to refer that a botnet (a network of bots) is acting under the instructions of one person (botmaster), thus one of the purposes of botnet is to infect as many computers as possible. The communication with the botmaster can be done via a central hub, propably an IRC Command and Conquer Server, or in a distributed manner for better scalability.

- **Downloader:** *Downloaders* is one more auxilliary category of malware. Downloaders are maliciou code that downloads other malwares. A typical case of downloader usage, is when attackers gain access for first time in a system, then they install a downloader in order to help them to download and install other malwares. Next to downloaders there is one more category of malware used to launch other malicious programs, the *Launchers.*

- **Information-Stealing Malware:** This category of malware is one of the most widespread categories used in financial transaction attacks like web-banking attacks. An *information stealing malware* is a type of malware that collects user information (i.e. sniffers, keyloggers) in an unauthorized manner from victim's computer and sends them back to the attacker. in this category are included the *spywares*, that gather user's personal information like frequently visited pages, email addresses or credit card numbers, and in most of the cases can be installed when free or trial software is downloaded.

- **Rootkit:** A *rootkit* is a type of malware designed to hide other malwares. Typically a rootkit can be utilized combined with a backdoor in order to allow remote access to a system while making it difficult to be detected.

- **Scareware:** Generally speaking, a *scareware* is a type of malware design to frighten the user of an infected computer, to pay for something. In most of the cases it provides to the user a legal an realistic interface, like an organization another application or even an antivirus. Actually, it tells to the user that his system is infected with some kind of malware and then imposes to pay in order to get rid of it while when paid it does nothing more that to remove the scareware.

- **Spam-Sending Malware:** This category of malware, infects users machine and after getting it under its possession, uses this machine in order to send spam. *Spam-sending malware* can be utilized for spam-sending services that could be sold to generate income for the malware authors.

1.2 Defence Against Malicious Software

In this section we present a brief introduction to the fundamental principles for defence against malware. To start with, we offer to refer that shield against malware is consisted solely from prevention. In order to achieve the proper prevention mechanisms we need to be based upon the triad of analysis, detection and classification.

To this point, we offer to explain that in the field of computer security, the processes of malware analysis, malware detection and malware classification are in some fashion interdependent, as depicted in Figure 1.1. What we mean, is that if someone needs to develop a detection method, firstly a knowledge base of software tagged as malicious or benign according to a *classifcation* process should be available. Additionally, in this base it is needed to perform an *analysis* on the malicious ones in order to extract a proper and sufficient set of characteristics that uniquely characterize if a software is benign or malicious and that will consist the feature set of the classification process. Thus, easily someone can understand that the process of malware detection has a classification flavor as the main goal is to classify if a given sample belongs to one of the two classes of benign and malicious. On the other hand, when someone needs to simply classify a given malware to one from a known set of malware families, as in the detection process, it is needed to firstly perform an *analysis* on the members of a family in order to extract a proper and sufficient set of characteristics that uniquely characterizes a family. However, in order to avoid false matches, a previous *detection* is needed since if the specimen is a benign software and the classifier falsely classify it into one malware family this will lead to further false positives in the detection process.

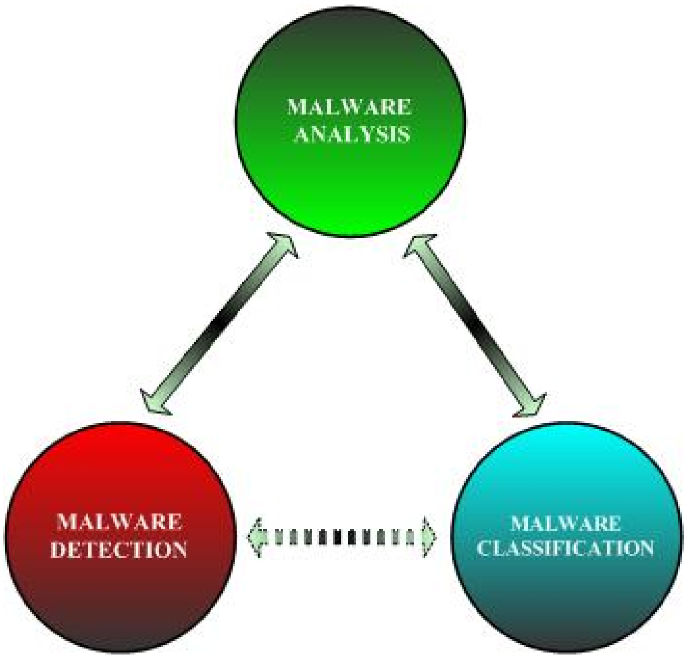


Figure 1.1: Interdependence of Analysis, Detection and Classification

1.2.1 Malware Analysis

Malware Analysis [8] is the process of determining the purpose and the functionality, or in general the behaviour of a given malicious code. Such a process is a necessary prerequisite in order to develop efficient and effective detection and classification methods. By its nature, malware analysis is a manual process demanding a lot of time and much more expertise. Malware analysis can be performed through two fundamental approaches depending to our goals, the given sample or the circumstances. Thus, malware analysis is spitted into two categories *Static* and *Dynamic* [56].

- **Static Analysis:** In static analysis the specimen is examined without its execution. Actually the suspicious sample does not need to be execute and the analysis can be performed on its source code.

- **Dynamic Analysis:** In dynamic analysis an execution of the malware has to be performed in order to collect the required data. However this approach needs more expertise while is extremely dangerous for the host environment. As a result, in most of times dynamic analysis is performed in a virtual environment

However, a further categorization can be done if we distinct these two approaches to Basic or Advanced Static Analysis and Basic or Advanced Dynamic Analysis according to the sophistication level of the techniques utilized in each one. So, according to [56], next we briefly describe these four categories of malware analysis, while a more extensive presentation is cited in the corresponding chapter.

- **Basic Static Analysis:** If we choose to use static analysis utilizing elementary techniques, actually we examine the executable file (malware) without viewing the actual instructions. The result of basic static analysis provides as with the knowledge about to if the specimen is malicious or not and the specimen's functionality. The main advantage of basic static analysis is that is straightforward and thus can be performed quickly. However, its main shortcoming is its ineffectiveness against sophisticated or brand new malwares

- **Advanced Static Analysis:** On the other hand, if the circumstances and our experience permits to use more elaborated techniques such reverse-engineering of the mawlare's internals by loading the executable file into a disassembler and consequently look at the program's instructions in order to determine its functionality.

- **Basic Dynamic Analysis:** In the opposite case where the circumstances permit the execution of the specimen (malware) we can leverage techniques that involve the execution of the malware. The execution of the malware under inspection can reveal precious information of its behavior when executed in a given operating system, while its interaction with it can imprint its behavior.

- **Advanced Dynamic Analysis:** Finally, the choice of advances dynamic analysis techniques invoke the use of a debugger for the examination of the malware's internal state. Through advanced dynamic analysis, the analyst easily can extract detailed information from the malware's executable when could me more difficult to be gathered when another type of analysis was performed.

1.2.2 Malware Detection

As referred in [40] the process of detection is all about to infer if a program is malicious or benign. Consequently, a malware detector is the implementation of a series of specific malware detection techniques [28]. Most common detection technique is the so called signature-based detection. As the name of this technique indicates, this detection method utilized signatures that are byte sequences that uniquely characterizes a specific malware [1]. Thus, formally speaking, a malware detector can be defined as a function that takes input an undefined program p and by scanning it for the existence or not of the signature s, determines if it is malicious or benign respectively.

In malware detection there exist two main approaches, *signature-based detection* and *behavior* or *anomaly-based detection.* The most common one is the signature-based detection that is also the more effective and sufficiently efficient to be applied on the commodity end-hosts, and thus is employed from all the anti-virus systems. The signatures, that consist the knowledge base of these systems, are created by extracting features from the analysis of disassembled code from malware's binary [40]. To this point, we offer to notice that these signatures can address the entire family of malwares that share commonalities.

On the other hand, behavior-based detection has as its main goal to analyze the behavior of known and unknown malwares [40]. Specifying normal behavior actions, then is easy to determine the anomalies exhibited by the behavior of a malware. Additionally, by specifying a rule set of normal behaviors, we conduct a specific type of anomaly-based behavioral detection the so called *specification-based detection* [28], where any program that behaviorally violates the rule set is claimed to be malicious.

1.2.3 Malware Classification

As we referred in the previous sub-section, in the process of malware detection the most efficient approach is the signature-based malware detection. However the main shortcoming of such an approach is the rate in which malware signatures are produced. What we mean, is that due to the high rate that malware is produced and, in most of the cases, the high rate that malware is evolving, lead to the need of acceleration to the malware signature construction. To address properly this situation, we have to notice that a probable solution could be the *compactness* of malware, because as referred in [33], while writing an individual signature for each distinct malware is a cumbersome and time consuming process. Thus, a quite convenient solution could be to cluster sets of malware according to the commonalities that they exhibit and create generic signatures for each group [33]. Through the literature [11, 4, 46, 49, 33, 6, 23], there does not exist a clear definition of malware analysis, where the vast majority of them let to be meant that malware classification is the process of classifying an unknown specimen, after its detection as malware, to one of the predefined malware families. However, there exists another approach that defines as malware classification as the process of categorizing a specimen to one of the malware types (i.e. worm, virus or trojan) as the one referred in [40].

1.3 Malware Mutations and Detection Avoidance

In order to avoid the traditional signature-based detection employed by the vast majority of Antivirus Software product, malware authors have implemented a series of obfuscation techniques. As referred in the literature, obfuscation techniques are deployed in order to contribute to malware's evolution. According to [13], the number of unique malware discovered per day is reaches the 8000 per day. However, to be precise, we should check the percentage of these specimen that are totally brand-new malware's and that of the specimens that actually are variants (mutations) of already existed malwares. In this section we present the categories of malware obfuscation techniques while we cite a more extended description for a specific category of malware obfuscation, the so called *metamorphism* that makes malware analysts' life even harder as consists the most powerful technique for detection avoidance.

1.3.1 Code Obfuscation Techniques and Malware Evolution

As we mentioned above, a series of code obfuscation techniques [40, 13, 57, 29, 48, 44, 62, 61, 69, 47] have been developed from malware authors in order to avoid the detection from AVs. Next we present some of the most known obfuscation techniques, as the knowledge of such techniques is able to help us to develop a deep theoretical background that may lead to the development of more sophisticated and flexible detection techniques.

- **Encryption:** The most straightforward technique to hide a malware's functionality is the *encryption* of its code. Such malware work by containing an extra module (encryptor) that is in position, to encrypt malware's body, while there is respectively another module responsible for the reverse process (decryptor) [57]. Structurally, an encrypted malware is composed by the decryptor and the encrypted main body. Thus, the decryptor decrypts the main malwares' body any time an infected object is run. A simple encryption may use 1-1 mapping, a zero-operand instruction or reversible instructions as AND or XOR [47]. To this point we have to notice that the main functionality of encryption lies in the fact that for each infection the encryptor makes the encrypted part unique by encrypting the main body with a different key and consecutively by hiding its signature [69]. In Figure 1.2 we cite an indicative example of the functionality of encryption in signature based detection avoidance. We use the same example into the following techniques. However, even though the detection of an encrypted malware (i.e. encrypted virus) seems difficult, the problem has been solved with a quite simple approach when a detector just tries to detect decryptors' code body as it remains constant from generation to generation.

- **Oligomorphism:** As easily someone would think, based upon the above solution of decryptors detection the question remains in what could happen if a malware author use an auxiliary encryptor for the whole malware, encrypting so the already encrypted virus body as the encryption and decryption modules, passing to a second layer of encryption (multi-layer encryption). This is the case of *semi-polymorphic* or *oligomorphic* viruses, a specific category of obfuscated malware that dispose encryption/decryption module for multi-layer encryption in order to avoid decryption body detection. The effort was done by malware authors in order to make the decryptor

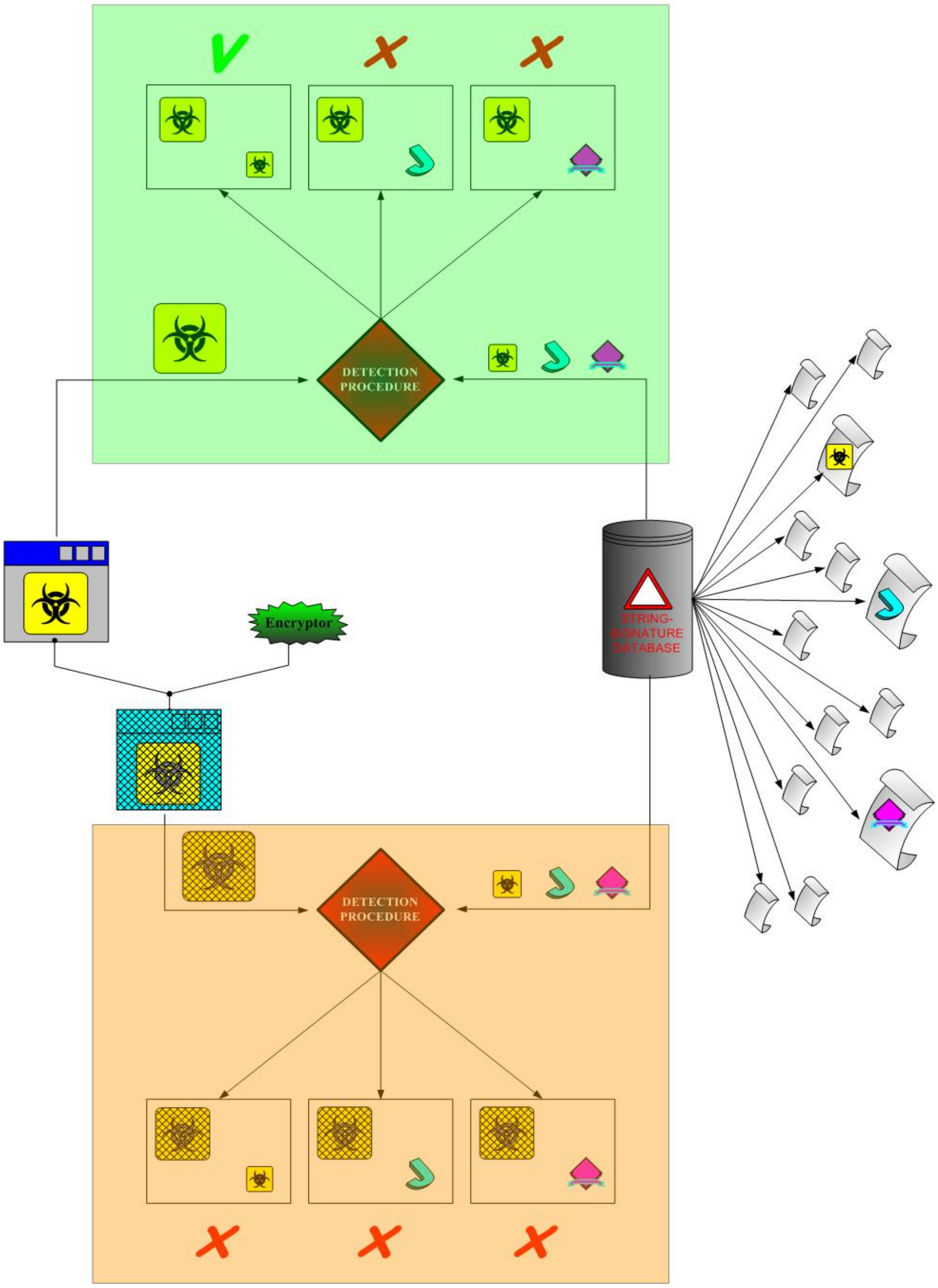


Figure 1.2: Signature-Based Detection Avoidance using Encryption

module to exhibit a different appearance in each new infection [47]. Additionally we ought to notice that did not missed the alternative approach of the containment of different decryptors that where randomly chosen. However, an polymorphic malware in the case of decryptor generation, can produce only a few hundreds of decryptors that are easily detected [69] in contrast with the polymorphic one that can produce countless decryptors, while as referred in [57], a draft solution that seems sufficiently effective is the dynamic decryption of the encrypted code instead.

- **Polymorphism:** A polymorphic malware can create an endless number of new decryptors that use different encryption methods to encrypt the body of the malware [57]. From this aspect someone could say that *polymorphism* is an advanced descendant of *encryption* an *oligomorphism.* As referred in [47], the main principal is to modify the appearance of the code constantly across the copies. However, we ought to underline that polymorphic obfuscation techniques are even harder to implement and manage. Some of the code obfuscation techniques [47, 69] used in order to mutate the decryptor are *dead code insertion*, *junk code insertion*, *code transposition*, *instruction substitution*, *instruction replacement* and *variable substitution* or *register substitution* and are executed any time needed by another module called mutation or obfuscation engine. However, there exist techniques such as *code emulation* [47] or *manual analysis* [57], that are in position to detect polymorphic malware by simple string matching.

1.3.2 Metamorphic Malware: A Major Threat

In contrast with encrypted, ologomorphic and polymorphic malware the metamorphic one has no encrypted part. Thus there does not exist any need for encryption module ,however there exist a corresponding mutation module called *metamorphic engine*, that is responsible for malware's mutation. Short speaking, a typical metamorphic engine [47], includes a disassembler, a code analyzer and transformer and an assembler. Consecutively, the mutation does not applied on the decryptor but on the whole body instead. As we will discuss next , every new copy has modified structure, code sequence size and syntactic properties [47], while its behavior remains the same.

Metamorphic malware changes its structure while keeps its functionality each time it replicate itself [44]. As referred in [13] polymorphic and metamorphic malware is the hardest type of malware to detect, since are able to mutate in an infinite number of functionally equivalent copies of themselves, and thus there is not constant signature for virus scanning [44]. The most advanced type of mutation malware is polymorphic malware. This kind of *body-polymorphic* malware changes its body from one instance to another, using different obfuscation techniques [40] such as *disassembly*, *permutation*, *expansion*, *shrinking*, or other kinds of transformation that we will describe later, to reprogram themselves in order to create descendants that have transformed code similar to the ancestor's code.

According to the definitions given in [44, 69] *metamorphism* is the process of transforming a piece of code into copies that are *structurally* different, *however* they *exhibit the same functionality*, that is a very important clue upon which is based our approach. Next, we proceed by describing of some code obfuscation techniques used by malware authors for the development of polymorphic malware. Organized by the object applied on these techniques are grouped in code-based, instruction-based, register(value)-based and control flow-based.

(A) Code Level Modifiacations

- **Code Insertion -** As easily someone can understand the easiest way to morph a program is to insert *dead* code. Dead code is a part of code that is never executed. The insertion of ineffective instructions can be included within a *dead code block* while it can be stealthy as it is non trivial to determine if it is executed or not [44]. Additionally, we ought to notice that such ineffective instructions change only the appearance of the program and thus its binary sequence while leave invariable its behavior and functionality [69, 47, 48]. The code insertion modifications are also called *control-flow preserving transformations* [44], since the insertion of instructions does not change the data-flow or the control -flow of the program.

- **Code Trasnposition -** Code Transposition is a malware mutation technique where the sequence of the instructions in an original code are reordered either in single instruction or code block level [47], without any impact to programs behavior by preserving the execution flow using conditional jumps or branches[48]. According to [69] it can be achieved either by shuffling the instructions and then recover them utilizing unconditional jumps or by choosing and reordering totally independent instructions. In the same manner works another variation of this technique, the subroutine reordering [69], that changes the order of subroutines in a code in random way generating $n!$ different variants, where n is the number of subroutines.

- **Equivalent Code Substitution -** As referred in [44] another technique able to change the structure of the program while keeping its behavior and functionality is the substitution of a series of instructions with a series of equivalent ones.

- **Code Integration -**Finally, one of the most elaborated code obfuscation techniques in metamorphism, is the so called *code integration* [69], where the malware knits itself to the code of a victim sane program. Staring with the decompilation of the program creates manageable objects and then seamlessly adds itself between them. Finally it assembles again the program with the integrated code creating a new generation.

(B) Assembly Level Modifiacations

- **Instruction Permutation -** A more complex method for code obfuscation is applied in the instruction level, where is possible to change the sequence of independent instructions with no disturbing the execution while byte strings in different versions of the code will appear unlike [48, 47].

- **Instruction Replacement -** Another technique that works in a similar manner is the instruction replacement, as this obfuscation technique substitutes instructions with their equivalent in the newer copies, thus changing its code appearance since a task can be executed in different equal coding instruction set [48, 47, 69].

- **Register-Value Modifiacations -** Finally, another transformation method that mutation engines use to obfuscate malware's code is the register substitution where is applied the usages of different registers or memory variables [48, 47, 69] as keeps the malware's behavior the same. This approach leads to the evasion from string signature-based detection as through this alteration are changed similar bytes in various generations.

1.4 Realted Work

In this section we present approaches that have been proposed through the literature and refer either to malware detection or malware classification. Next, we present some indicative examples of proposed solutions that are not graph-based ones. We ought to notice that we present to a greater extent only the approaches that utilize a graph theoretic background while the number of the works in the field of malware detection is quite large to fit for discussion in this sub-section.

- **Related Work on Malware Detection**
  - In [44] the proposed approach trains a Hidden Markov Model (HMM) using a sequences of opcodes extracted from the suspicious sample's executable.
  - In [55] API call sequences are used in order to classify if a process is benign or malicious.
  - In [50] n-grams are extracted from files and then by utilizing the k-NN the proposed solution distinguish the malicious from the benign programs.
- **Related Work on Malware Classification**
  - In [29] the proposed technique uses the n-grams extracted from the bytes or instructions of the executable as features that then are used to classify malware samples into malware families.
  - In [36] the proposed technique also uses n-grams as features for naive Bayes, decision trees, SVMs and boosting in order to detect and classify malware found in the wild.

Next, we proceed by presenting the related work done in graph-based malware detection techniques and this done in graph-based malware classification techniques. Some of them are using dynamic and others static analysis, while is notable that the vast majority of them are integrated systems incapable of real time appliance in the end-hosts.

1.4.1 Graph-Based Malware Detection

In [1] Alazab *et al.* propose a fully automated system that effectively disassembles and extracts API call features from executable, and then classifies an executable as malware or benign by using $n - gram$ statistical analysis of its binary content.

Kolbitsch *et al.* [35] present the only approach that can be applied in the end-host in order to run in real-time as a substitute of commodity AVs. Namely, it analyzes a malware in a controlled environment building a model that describe the information flows between system calls and thus characterizes malware's *behavior*. Finally after extracting the program slices (*training*) that are in due for such flows they execute them to match their models (*test*) against the run-time behavior of an unknown malware.

Bonfante and Kaczmarek in [9] propose yet another graph-based malware detection technique that utilizes the use of Control Flow Graphs (CFGs) as signatures for malware. There, the nodes of the graph are X86 instructions, as they make a reduction to the graph by omitting nodes with low information.

In [18] Christodorescu *et al.* designed a malware detection algorithm that addresses the deficiency of mutated malware by incorporating instruction semantics to detect malicious traits. Specifically, they describe malicious behavior by using instruction sequences with variables and symbolic constants, the so called *templates*. In this way the algorithm after the disassembling of the binary program it constructs a CFG and searches for matches between each template node and a matching node in the program.

Again Christodorescu *et al.* in [17] propose an algorithm that automatically constructs *specifications* of malicious behavior needed by AV's in order to detect malware. The proposed algorithm constructs such specifications by comparing the execution behavior of a known malware against the corresponding behaviors produced by benign programs.

In [52] Sekar *et al.* present a Finite State Automaton approach is presented, where a compact FSA is builded forwa program without requiring access to its source code while requires low space for storage. It is notable to refer that the proposed FSA-based technique is able to capture short and long term relations between system-calls performing more accurate detection.

In [39] Luh and Tavolato desing an algorithm that automatically grades an unknown executable as potentially malware or benign leveraging behavior-based analysis by executing the sample and creating reports used to score the sample.

Finally, in [3] Babic *et al.* propose an approach to learn and generalize from the observed malware behaviors based on tree automate interference where the proposed algorithm infers k-testable tree automata from system call data flow dependency graphs in order to be utilized in malware detection.

1.4.2 Graph-Based Malware Classification

Park *et al.* in [46] propose a classification method based on maximal common subgraph (MCS) detection for the similarity measurement between two behaviour graphs createdby capturing system-calls during the execution of a program.

In [27] Hu *et al.* design and implement the *Symantec's Malware Indexing Tree* (SMIT), a malware DBMS, that can efficiently determine if a new malware is similar to a previously seen one, based on malware's function-call graphs, using k-nn clustering algorithm.

Kinable and Kostakis in [33] explore the potentials of call graph based malware detection and classification by defining an algorithm that computes the graph similarity through the edit distance taking into account so the vertex and edge cost as the relabeling cost while uses k-medoids clustering in order to cluster samples to families.

In [6] Bayer *et al.* propose an approach for scalable clustering in order to identify and group malware that exhibit similar behavior. By performing dynamic analysis execution traces of malware programs are obtained and then generalized as behaviour profiles. Finally these profiles are served as input to an algorithm that allows to handle sample sets.

Fredrikson *et al.* [23] present a technique that automatically extracts *optimally discriminative specifications* that uniquely identify a classes of programs utilizing graph mining and concept analysis. and thus can be used from behavior-based malware detectors.

Babic *et al.* in [3] propose an approach to learn and generalize from the observed malware behaviors based on tree automate interference where the proposed algorithm infers k-testable tree automata from system call data flow dependency graphs that, except from their appliance on malware detection that we referred in the previous sub-section, they also can be utilized in malware classification.

Finally, in [49] Rieck *et. al* present a method for malware classification that proceeds in three stages, firstly by collecting malwares' behaviors, then using learning techniques to train a classifier with labeled specimens obtained from AV's, and finally by ranking the discirminative features of behavior models in order to explain classification decisions.

1.5 Contribution

In this section we present the basic incentive behind the start of this work that are about the difficulties generated in malware detection because of the mutated malware and also we make a brief description of our proposed model.

1.5.1 Motivation

The main objective of this work is to develop a system that by the implementation of a sophisticated algorithm will be in able to detect any variation of a mutated malware. Specifically, our main viral is the metamorphic malware. As we described in 1.3.2 metamorphic malware can easily avoid the traditional string signature-based detection methods and thus more elaborate techniques ought to be developed. Such difficult problems are triggering us to develop algorithms that leverage abstractions of malware's structure in order to utilize them in detection and classification. The solution that we present is designed and implemented having as set squares so the generalization in terms of variation-independent malware detection and classification, as the perspective of the adoption of an end-point system in terms of real-time protection.

1.5.2 Proposed Solution

The solution that we propose is based on System-Call Dependency Graphs (SCDGs) produced via taint analysis capturing the execution trace of a malware. Having an instance of such a graph we proceed by the creation of an abstraction of it, by utilizing system call classification obtained by the configuration file of NtTrace, a native API tracing tool for MS Windows. This graph abstraction merges each node (system-call) of the initial graph to a node with the name of the system call class and then links the corresponding super-nodes. Finally, having these graph abstractions, we have designed a series of metrics that leverage known similarity metrics in a combinatorial manner in order to produce results about the nature of the examined specimen as to detect if it is malware or not and if so to classify it to the corresponding malware family. In contrast with other approaches, we have developed a system that both detects and classifies a suspicious specimen. Finally, to this point we ought to refer that this implementation although is experimentally tested is not yet ready to be applied to end-point computers.

1.6 Structure of the Thesis

The remainder of this thesis is organized as follows. In the second chapter we present one of the most interesting fields in computer security, malware analysis. In this chapter we describe in a greater extent techniques applied in dynamic and static malware analysis while we suggest some tools that can be utilized for such purposes. In the third chapter we present the various types of methods applied in malware detection describing their pros and cons while we present some state of the art graph-based malware detection techniques proposed through the literature. In the fourth chapter we analyze some of the similarity metrics used in malware classification and additionally we cite alternative malware classification techniques that utilize graph representation of malware. To this point we ought to notice that in the topic of graph-based malware classification there exists less work done than in graph-based malware detection, which is a hint for farther research. In the fifth chapter we present and analyze our proposed model for malware detection and classification based on System Call Dependency Graphs, where we describe the graph construction procedure and the development of our proposed techniques for malware detection and classification. In chapter six we analyze our data set, describe our experimental setup (design) and project the experimental evaluation of our proposed model's implementation against real malware samples. Finally, we compare our model against the results produced by other graph-based approaches with no distinction to if they are designed solely for malware detection or classification.

Chapter 2

Malware Analysis

In this chapter we will present the two main streams in malware analysis, the static and the dynamic malware analysis. Firstly we will discuss the basic methodologies applied in the static analysis approach while we will cite a few tools that malware analysts utilize in order to perform static analysis, and then we will present the basic techniques applied in dynamic malware analysis and respectively we will cite the corresponding tools utilized in dynamic malware analysis. This chapter has a somehow smaller extent since, although malware analysis is a quite interesting technique, there does not exist much work in literature because of its hands-on-craft nature as it is a more human based method. The vast majority of the publications present only implementations that automate traditional made by analysts techniques, while the background of such techniques is out of the scope of this work.

2.1 Static Malware Analysis

As we mentioned in the introduction, with the term *static analysis* we refer to the process of analyzing an unknown program without executing it [22, 56, 32, 15, 8, 43]. Static malware analysis, since it does not demand the execution of the specimen under inspection is thus more safer for the testing environment, however demands a higher level of programming skills and also a deeper knowlegde of object's structure since the available software can be in different types varying from plain source code to binary files. Thus static analysis splits, according to the analyst's level and the techniques he utilizes, to basic and advanced static analysis.

Basic static analysis is straightforward and thus can be performed quickly includin elementary techniques of a brief examination in the executable file without viewing the actual instructions, providing us knowledge about the specimen's type (malicious / benign). As we referred in the introduction, static malware analysis has a few drawbacks such as its inability to detect a totally brand-new malware when is performed in its basic approach, while even in its advanced one, is quite difficult to be performed when malware's source code is unavailable as more sophisticated techniques are required. Specifically, as mentioned in [22], static analysis of binaries may cause some problems to the procedure such as the disassembling that may cause ambiguous results when performed on self-modifying malware. However, despite these drawbacks,static analysis has the advantage that it can cover the complete program's code [8] and in most of the cases is faster that the dynamic one.

2.1.1 Static Analysis Techniques

In this sub-section we will enumerate some of the most used static analysis techniques that when applied can reveal valuable information about the testing specimen's structure.

- **File Fingerprinting:** A typical malware's fingerprint can be consisted from its file's hash value. Hashing is a common method used for identifying malware uniquely. As refered in [56] the hash value can be computed in a part of the malware and then can be quite useful especially when used as label or shared with other analysts for same purposes.

- **Anti-virus Scanning:** Before someone starts the analysis, is advised to firstly scan the testing specimen with at least one or more anti-virus software in order to detect it. Its is probable that some anti-virus software may have already detect this specimen [56] if it is malware and thus no further investigation is needed. Additionally, despite the fact of gaining time from an already done work, the anti-virus vendors provide with a detailed reports [32] about the specimen where the analyst can find information about malware's capabilities, its signatures and in many cases instructions for its removal. However, as we mentioned in the introduction malware authors may have changed the code of malware and consecutively its signature and hence anti-virus software will not be able of detecting it.

- **String Searching:** A very naive approach in elementary static analysis is the string search. There is surprisingly a lot of information in a malware's source code in strings of readable text. As referred in [32] there exist strings that inform the user with update status, an error occurrence, a connection to a URL or to copy a file to a specified folder. As easily someone can understand, a quick web-search of these strings can reveal valuable information.

- **Analyzing Obfuscated Malware:** As we described in the introduction malware authors often use obfuscation techniques in order to evade detection. Except from obfuscation techniques another technique that malware authors utilize for the same purpose is packing. Packed malware is somehow a malware that has been compressed and thus it can not be analyzed. As referred in [56], the legitimate software often includes many strings. This declaration is able to lead us to the conclusion that if a software includes few line then it probably may be a malware. Consecutively, the elementary techniques mentioned above are not enough to perform the analysis. The most helpful knowledge in such circumstances is that when a packed malware is executing then a small wrapper program is running to decompress the packed malware. Such auxiliary program are called *packers* and can be detected using the PEiD program as described in [56].

- **PE File Format:** One of the most valuable information about a program's functionality can be revealed through PE (stands for *Portable Executable*) file format used by executable files on Windows systems [32]. The PE file format is a data structure that contains necessary information for the Windows OS loader to manage wrapped executable code, object code and DLLs [56]. The core segment of PE appears in its begin where there exist the header that includes information about the code, the application type, the library functions, space requirements, compilation date and time, imported and exported functions, version information and *strings* embedded in resources [56, 32].

- **Linked Libraries - Functions :** Additional valuable information can be collected through the library linking. The imports are functions stored in a program and used from another one. Thus, code libraries can be connected to an executable by *linking* [56].

  Next we present three basic linking methods an describe the information retrieval when they are observed in static analysis.

  - **Static Linked Libraries:** In *static linking* the code of the library is copied inside the executable growing its size. The main problem in the analysis of static linked libraries, as described in [56], is that the analyst can not distinguish the linked from the main executable code.
  - **Run-time Linked Libraries:** On the other hand, a commonly used library linking method is the *run-time linking*, the libraries are linked only when needed by the executable. To this point it worth to mention that run-time linking is mainly used by packed or obfuscated malware.
  - **Dynamically Linked Libraries:** Finally, maybe the most interesting type of library linking is the *dynamic linking*, where the host OS searches for the necessary libraries when the program is loaded. The interesting is that the

information relevant to the libraries to be loaded and the functions that will be used is stored in the PE file header we mentioned above.

- **Imported - Exported Functions:** Imported and Exported function can aslo reveal valuable information about an executable's functionality. *Imported* Windows functions can give valuable information to the analyst even by their names revealing somehow what the executable does. On the other hand, the *exported* functions interact wit other programs' code. DLLs in example, provide functionality used by executables. In contrast, if an analyst discovers exported functions inside an executable, since is not designed to provide functionality to other executables [56], is very helpful to claim it as malware.

- **Disassembling:** Right after the conduction of such elementary static analysis techniques, follow more advances static analysis techniques like the disassembling of the examined file and analyzing the assembly code instructions that make up the program [32]. Since the description of disassembling techniques are far out of the scope of this thesis we will mention only that there exist ready-to-use tools like IDA Pro, that we will suggest in next 2.1.2, that are indicated for use in such techniques.

2.1.2 Static Analysis Tools

According to the techniques we previously enumerated, for the hash value computation the most used algorithms are the SHA1 and the MD5. On the other hand for the obfuscated malware in the case of packed one, PEiD is recommended in [56] since it can detect packed files by detecting the type of *packer* or *compiler* employed to build the application. For the investigation of PE files the PEView can browse the analyst through a lot of valuable information Next, the dynamically linked libraries can be explored with the Dependency Walker, distributed with MS Visual Studio, that lists only the dynamically linked functions in an executable. Finally, when advanced static analysis techniques are deployed, the Interactive DisAssembler Professional is recommended and wide used by most of virus analysts. IDA Pro is able to disassemble an entire program and perform function discovery, stack analysis local variable identification and much more are detailed described in [56].

2.2 Dynamic Malware Analysis

In this section we will present another effective technique for analyzing malware, the dynamic malware analysis. With this term we refer to the usage of dynamic techniques for analyzing malware during run-time [8]. The main advantage against static analysis is that in dynamic analysis is immune to obfuscation techniques as the analyzed instructions are the ones that code actually executes. So, firstly we will present the basic dynamic analysis techniques as they are described in the available literature while finally, as in the previous section, we will enumerate some tools that are utilized in dynamic analysis. As referred in [22] the analysis of actions performed by a program while it is being executed is called *dynamic analysis*. As dynamic malware analysis is performed while actually executing the malware it has to be done in a fully isolated and thus safe environment worth to sacrifice, meaning in example a virtual machine. Dynamic analysis is also called *behavioral* analysis since the analyst actually observes the behavior of the malware or in other owrds the interaction it has with its environment, in our case the operating system. As mentioned in [32] a fairy good picture of malware's behavior can be developed by simply monitoring its interaction with the file system, the registry, other processes and the network. To this point we ought to underline that even though dynamic analysis techniques that we present next are extremely powerful and plenty of valuable information, they should be performed only after the performance of static analysis and much more the monitoring should be performed very carefully since may put at risk the analyst's system or its entire network. Finally dynamic analysis has one more limitation, that is not actually a drawback, is the fact that not all possible execution paths my execute when a malware runs [42].

2.2.1 Dynamic Analysis Techniques

Through the dynamic malware analysis technique we focus on capturing the behavior of the testing malware. The term behavior as referred in [63] includes the files that the sample tries to create or modify, the changes it attempts to perform in Windows registry, the loaded DLLs, the accessed virtual memory areas, the creation of processes, the network connections it opened and other information.

- **Function Call Monitoring:** As we know, a function consists of code that is responsible for a specific task. However, even it seems to be a trivial notion, the abstraction of such implementation details can reveal a semantically richer implementation [22]. In order to analyze a program's behavior it is needed to intercept in some fashion between function calls, a process called *function hooking*[63]. Consecutively a dummy function that is responsible for that procedure is called *hooking function* [22]. Such functions are responsible for recording the hooked function's invocation to a log file or analyzing its input parameters, which is information that later we will leverage in order develop our model (see chapter 6). Next, we cite some system related functions that can be monitored in order to observe malware's behavior. When function calls are monitored it results to the *function call trace* [22]. Such traces consist by a a sequence of functions with their arguments invoked by the malware under analysis during its execution.

  - API: These functions form a coherent set of tasks. Usually the operating systems provide many sets of *application program interfaces*used by other applications to perform common tasks [22, 56].

  - System Calls: While the common applications are executed in *user-mode* the operating system is executed in *kernel-mode*. Thus, only the kernel-mode executed code has direct access to system's state. However a user-mode application can request from system to perform a limited set of tasks using the *system calls*, a specific API provided by the system. The interest of such API comes from the fact that malware actually is an application and since it executes in user space it needs to invoke a corresponding system call in order to interact with its environment [22, 56].
  - Windows Native API: Finally, Windows Native API resides between the system calls and the Windows API. As referred in [22]. the legitimate applications use the Windows API to interact with the operating system, whereas malware commonly skips this layer and interact with the Native API to thwart analysis techniques like function hooking.

- **Function Parameter Analysis:** Function parameter analysis in dynamic malware analysis focuses on the actual values passed when a function is invoked [22], as by tracking parameters and return values leads to the correlation of function calls.

- **Information Flow Tracking:** Information flow tracking focuses on how the *interesting* data are processed by the program. This type of data are marked with a label in some fashion (so called *tainted*), and each time they are processed the propagate their label.
  - Taint Source and Taint Sinks: As referred in [22], the introduction of this data's label is made by a *taint source*, while a *taint sink* is a system component that reacts when stimulated with tainted input.
  - Directed Data Dependencies: In order to be propagated the tainted data's labels, a direct assignment of arithmetic operation must be dependent on a tainted value
  - Address Dependencies: Accordingly, when needed to taint addresses a label propagation can be achieved when a read or a write operation has target an address derived from tainted operands.

- **Instruction Trace:** The sequence of machine instructions that the sample executed during its analysis consists its instruction trace[22]. Instruction trace contains includes valuable information that is not contained in form of higher level abstractions of malware's behavior.

- **Auto-start Extensibility Points:** The auto-start extensibility points [22], define system mechanisms that allow programs to be invoked when the system boots. So, it is of major interest to investigate them since it is probable for malware to try to add itself to an available auto-start extensibility point.

- **Taint Analysis** Since we have developed a basic background about function call monitoring we proceed by presenting a specific type o dynamic analysis, the so called Dynamic Taint Analysis. Dynamic Taint Analysis is the monitoring of the data flows in programs or whole systems during the execution of the sample [3]. Dynamic Taint Analysis is a very powerful technique to extract data-flow dependencies among executed system calls. Additionally, it can be applied in a set of taints as a single path symbolic execution. As referred in [3, 51], and we explained above, a label (taint) is introduced by a taint source (system calls) and through program execution it propagates according to some propagation rules to the taint sink (system call arguments). In Figure 2.1 we present an analyze to a greater extent the procedure of Dynamic Taint Analysis of an unknown executable since is the technique that as we referred we will utilize in our approach.

2.2.2 Dynamic Analysis Tools

In this section we present some tools widely used in dynamic malware analysis as they described in [56]. In order to monitor registry, file system, network, processes and thread activity Process Monitor is an advanced monitoring tool suitable for windows. On the other hand when performing dynamic analysis centralized in process monitoring, Process Explorer is referred as displays child-parent relations between the running processes. In a deeper level, through Process explorer, the analyst can launch the Dependency Walker, a powerful tool that let provide the analyst with valuable information about handles and DLLs. Additionally, RegShot is one more powerful tool that can compare two registry snapshot in order to check the changes happened in registry during malware's execution. Finally when needed to observe the network activity performed by malwares execution Netcat can be used in order to capture inbound and outbound connections for port scanning, forwarding and much more.

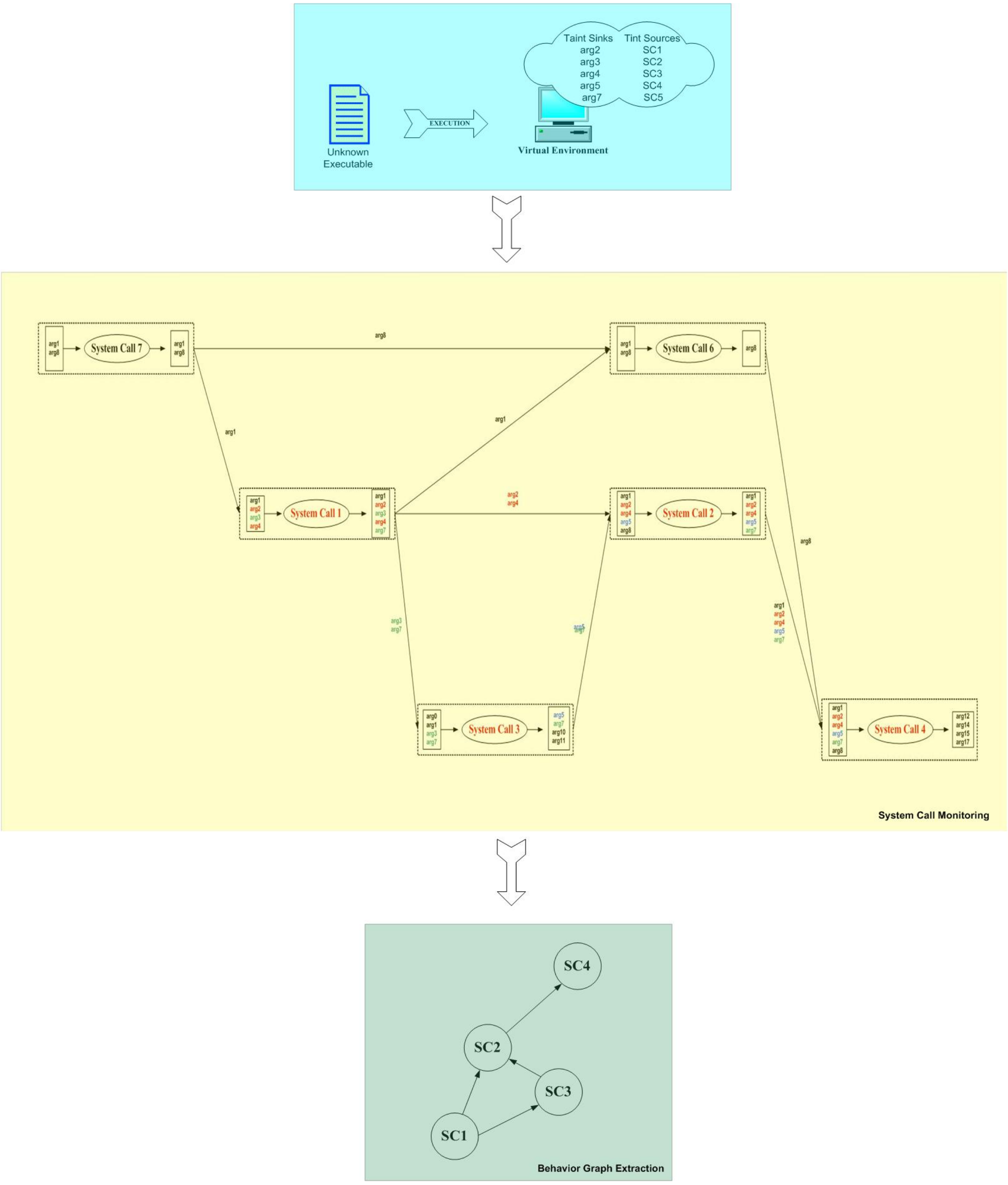


Figure 2.1: Dynamic Taint Analysis Procedure

Chapter 3

Malware Detection

In this chapter we will present the process of malware detection to a greater extent, by citing firstly the definitions of detection techniques, and then by describing a series of state of the art approaches presented in the literature while we focus on the graph-based ones as they are relative to our model composing its background. In order to make the things crystal-clear from the start and not to confuse the reader, we define the *specific* term *signature* as something that an object *has* while we define the *general* term *behavior* as something that an object *does.* Thus, the signature depicts the sometimes variant and others invariant characteristics, that a program has in order to implement some actions, while its behavior represents the actions performed by a program in order to achieve its goals, including its interaction with its environment.

3.1 Concept and Implementation

In this section we will present the main concept of mawlre detection providing the basic definition while we will describe what is and how works in general a malware detector. We cite a typical malware detection system's design compiling information from various approaches throug the literature.

3.1.1 Malware Detection

Malware detection as a general term is the process of determining, if a given program is malicious or benign [18, 40, 28, 1], according to an a priori knowledge. For this purpose there have been implemented techniques that leverage a series of distinct characteristics in order to be able to distinguish malicious from benign programs. The implementation of malware detection can be treated as a procedure highly intertwined with the process of classification. Actually one can think that the detection of malware has the sense of classifying an unknown specimen into exclusively one of the solely two classes malicious or benign. However, formally speaking, a malware detector can be defined as a function that takes input an undefined program p and by scanning it for the existence or not of the signature s, determines if it is malicious or benign respectively [1].

Malware detection is implemented through the utilization of a series of specific malware detection techniques [28]. In current system,malware detection is implemented with two approaches, *signature-based detection* and *behaviour* or *anomaly-based detection*. In fugure 3.1 we cite a brief representation of malware's detection method.

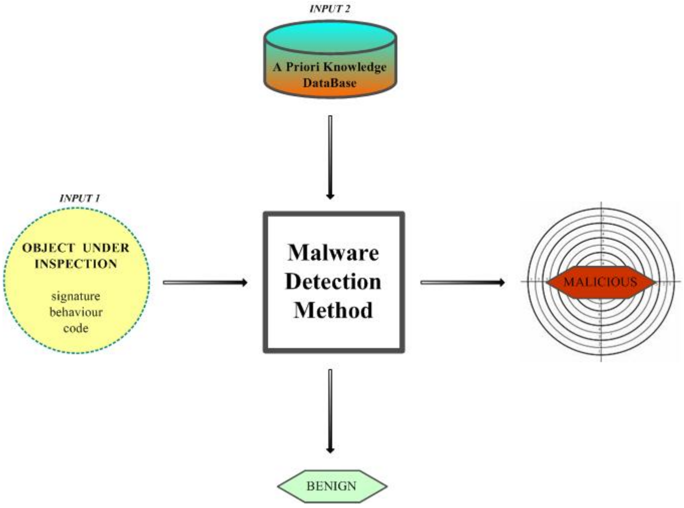


Figure 3.1: Malware Detection

Despite the fact that these techniques are quite efficient, offering relatively high detection rates whereas provide real-time protection on the end-host, they have a significant drawback. That is, the above mentioned techniques, and especially the signature-based one, is inadequate to detect malware that has been morphed through an obfuscation technique. As we referred in the introduction, the main prickle in malware detection that is also one of the most complicated and most researched topics in malware analysis, is the so called polymorphic malware. Thus, malware analysts in order to be able to detect mutated malware worked on more sophisticated techniques that utilize more abstract characteristics of malware such as its behavior. Formally speaking, the problem of obfuscated malware's detection as described in [16] can be expressed as follows: lets as assume that we have an initial variant (*vanilla*) of a virus V containing a set or sequence of instructions σ. When V is obfuscated, let us say $O(V)$, the problem is transformed as to detect the existence of a sequence σ' that is semantically equivalent to σ.

3.1.2 Malware Detector Design

As referred in [28], the implementation of techniques for malware detection is called *malware detector*. In order to understand how malware detectors work we present a generalized view of a typical malware detector's design, synthesizing various approaches from the literature [9, 16, 15, 11, 40, 64, 1, 66, 24]. As we mentioned in the introduction, the procedures of malware analysis , detection and classification are strongly dependent. Malware detection needs an a priori knowledge in order to *compare characteristics* extracted from a previously *analyzed* malware and a currently *analyzed* unknown specimen. Thus, having already composing the background of malware analysis from chapter 2, before we proceed with the presentation of detectors design we cite in Figure 3.2 the process of analysis that can be treated as a training phase in order to create the aforementioned a priori knowledge, stored into a data base.

Next, in Figure 3.3 we cite a very brief representation of how a malware detector works, omitting specific information about how the similarity measurement is computed. The feature that we will choose to select may refer to a signature extracted from executable's binary, or to a sub-graph of the flow-graph produced from the executable through static analysis [11], while the similarity measurement may include either a string-signature based technique or a behavioral one. Additionally as we will refer later it may include a graph matching process if a graph-based detection technique is employed. In the next sections we will discuss how these features are created and extracted in order to extract characteristics able to distinguish malware from benign programs.

Describing the example presented in Figures 3.2 and 3.3 we start with our data base creation. In order to be able to detect if a suspicious sample is malware or benign we must have some *clues* that indicate that a sample is malicious. This way, we start by performing dynamic or static analysis on a set of known malware samples, one at a time, extracting specific features (Figure 3.2). When we have in our hands either a signature or a behavior or any other kind of a representation of the aforementioned characteristics (i.e. a graph) we store it into a data base in order to be able to use it later. When a new suspicious executable is needed to be categorized as malware or benign, we start by extracting the same characteristics (Figure 3.3), either proceeding with their comparison with what we have in the database, in the case of behavior-based detection, or searching for the existence of a specific one from the database into the sample, in case of string signature-based detection respectively.

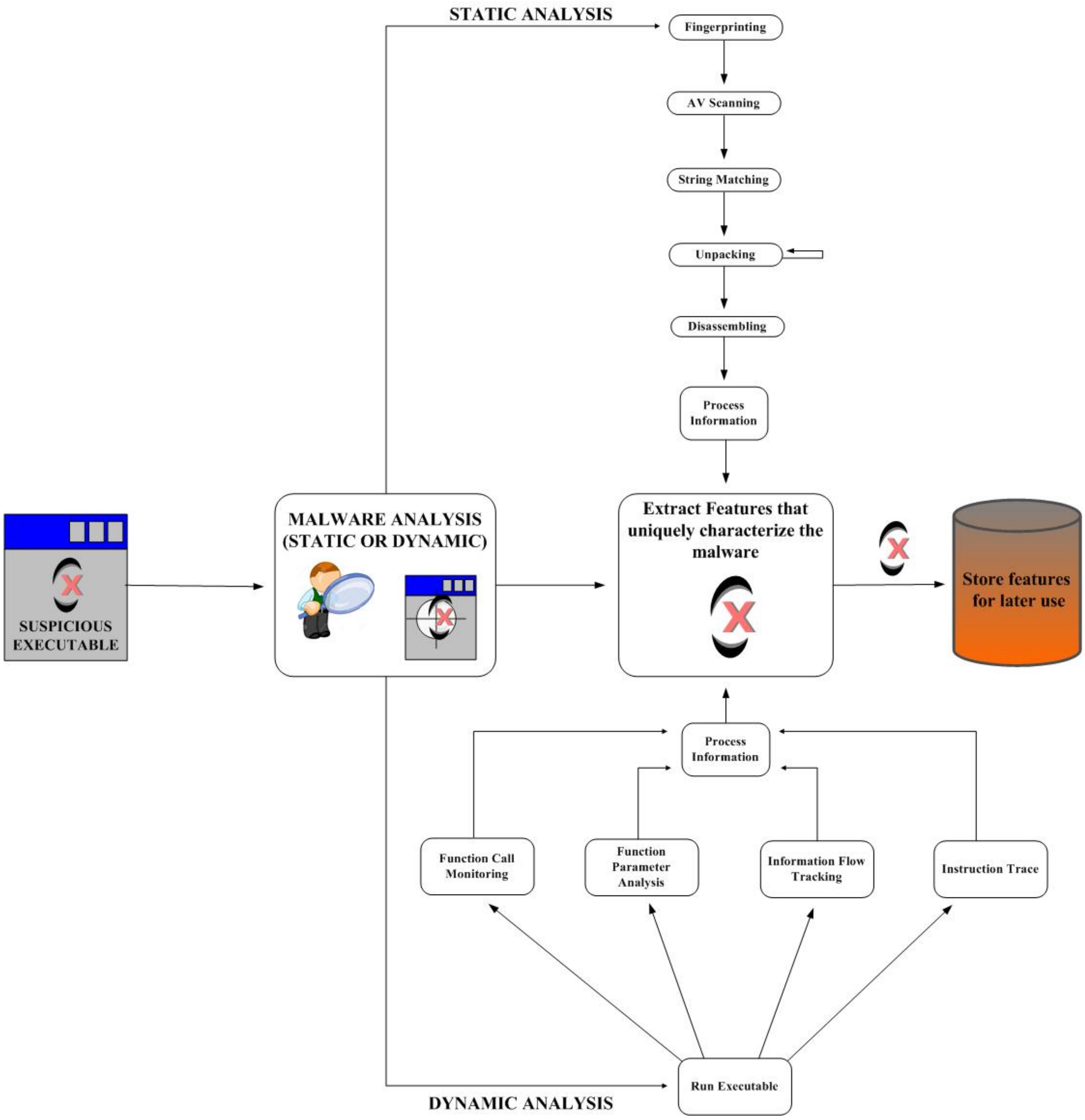


Figure 3.2: Malware Analysis

3.2 Categorizing Detection Methods

As we referred in the introduction malware detection methods can be categorized into signature-based detection and anomaly or behavior-based detection, according to the object the are applied on. In this section we will discuss to a greater extent the categories of malware detection methods enumerating their pros and cons respectively.

3.2.1 Signature Based Detection Methods

Signature-based detection is the dominant virus-detection method. Implementing this technique, a malware detector searches in program's under inspection raw content for the presence of a virus-specific sequence of instructions, the so called *virus signature* [15]. If malware detector find such signature then the program under inspection is probably

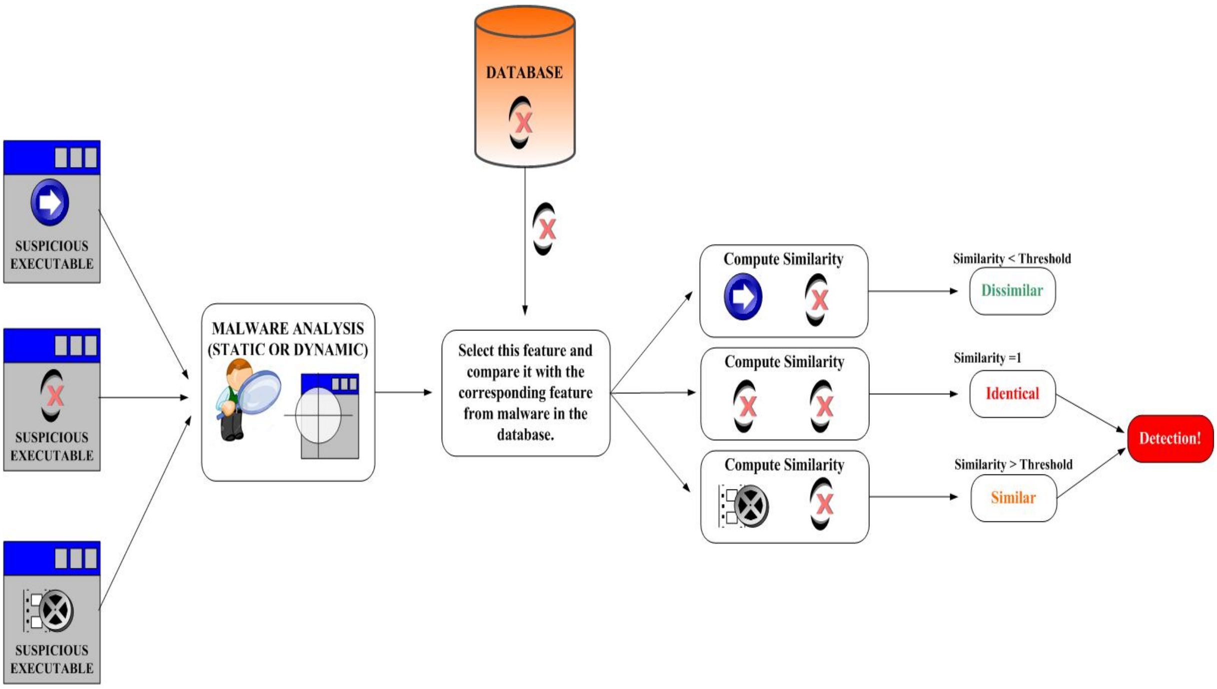
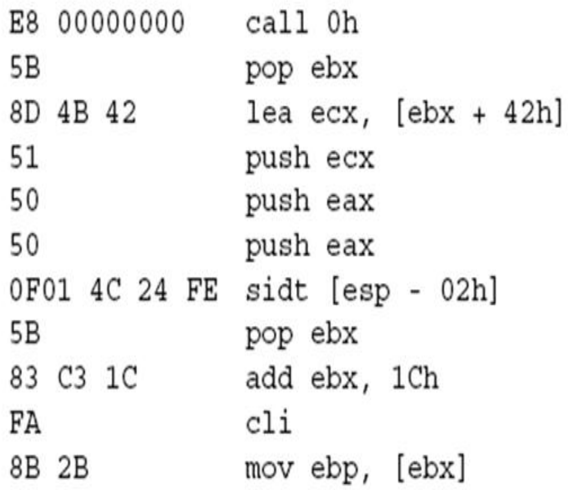


Figure 3.3: Malware Detection

infected. Actually, a string signature represent a *pattern* in a suspicious program's raw content and thus is used in order to uniquely identify it. Fast string matching algorithms are used in signature-based detection, utilizing regular expressions and string alignment techniques in an effort to detect malware variants. The extraction of malware's signature can be achieved by disassembling the malware's file and selecting some pieces of unique code [1].

In Figure 3.4 we cite the IA-32 instructions (right) from a program and its hexadecimal representation (left), as presented in [15]. Now, let us assume that we have a given signature $S : \{E800\ 0000\ 005B\ 8D4B\ 4251\ 5050\ 0F01\ 4C24\ FE5B\ 83C3\ 1CFA\ 8B2B\}$, stored into our signature database, corresponding to Chernobyl/CIH virus. As easily one can understand, if the malware detector during the scanning of a suspicious program, finds the instructions sequence S, then will determine that the given program is infected by Chernobyl/CIH virus. Now that we have developed a better knowledge about how signatures work we can return to Figure 3.3 and explain the case of string signature-based detection. As referred in [1], the signature of a malware is consisted by a byte sequence that uniquely identifies this malware. Once a set of such signatures has been collected for a series of malware and then been stored in a database, then the *malware detector* utilizes this set by looking for code signatures or byte sequences inside the programs of the system it is installed on. Thus, the *malware detector* scans specific locations in the system and if in a program is found a signature that matches with one in detector's database then this program is claimed as malware and its access to the system is blocked by the detector. Even though this practice seems extremely efficient for the end-host considering its accuracy and speed, however its main drawback is its inability to detect



```
E8 00000000    call 0h
5B             pop ebx
8D 4B 42       lea ecx, [ebx + 42h]
51             push ecx
50             push eax
50             push eax
0F01 4C 24 FE  sidt [esp - 02h]
5B             pop ebx
83 C3 1C       add ebx, 1Ch
FA             cli
8B 2B          mov ebp, [ebx]
```

Figure 3.4: Virus Chernobyl/CIH body and corresponding IA-32 instructions [15]

brand-new or mutated malware, or in other words its inflexibility to generalization. Thus, the only solution for such approaches to work properly is to keep updated their signatures databases in order to be possible to detect at least as many malware variants have been already detected by the Anti-virus system vendor.

Despite all the theory we cited above, we ought to notice that the term signature is more generic as it seems. Through the literature, the term signature may also refers to more abstract objects such as a set of actions and many times it may gets confusing. We will just mention the example in [18], where malware signatures are represented by templates that actually are set of actions that compose a profile. Similarly, we will refer the terms host-based signature and network signatures as they discussed in [56]. A *host-based signature* is used to detect malicious code on a victim computer by identify files created or modified by a malware or by detecting changes made to the registry. These signatures are also called *indicators* and focus on what the malware does to a system in a more behavioral manner in contrast with the traditional string signature that focus on the characteristics of the malware. Thus, as a result indicators are more resilient to morphed malware. Finally, there are also exist the *network signatures*, that detect malicious code by monitoring the network traffic.

Additionally we can proceed to a further categorization of signature-based detection where this hyper-category of detection methods is divided into static and dynamic [28], just like the analysis. Thus, in *Static Signature-based Detection* the program under inspection is examined for *sequences of code* and so the signature are representing sequences of code. On the other hand in *Dynamic Signature-based Detection* the maliciousness of the program under inspection results from data gathered during its execution time, such as patterns of behavior (not to be confused with behavior-based detection).

Finally in order to make the things crystal-clear, we notice that the main difference between Signature-based detection and Behavior-based detection is that the Signature-based one is in some fashion a *static* detection method, as it relies on something that we got *a priori* and it is fixed, demanding consequently update for each new variant. On the other hand, the Behavior-based Detection is a more *dynamic* one as it relies on some global rules that if offended then the maliciousness of the subject can be claimed without the need of updating this *a priori* knowledge as it can be applied to all. Summing all the above we can conclude that a signature is something characteristic for an object that its existence indicates the objects identity while a behavior, as we will see next, is a set of rules, that when violated then the identity of the object is indicated. Thus, if a malware detector uses signatures in order to detect if a program is malicious or not, then it is actually searching for the existence of something (i.e. byte/instruction sequence, set of actions etc.) existed also to other malwares, while if it uses the behavior then it is actually searching for a violation of a rule (i.e. resource misuse) that benign programs do not as we will discuss next.

3.2.2 Behavior Based Detection Methods

As referred in [28] anomaly-based detection depends on the normal behavior of an executed object. Actually it occurs in two phases which is the training or learning phase and the detection and monitoring phase. The goal is for the *detector* actually to learn the behavior exhibited by a program under inspection. More precisely, anomaly detection systems build models of *expected behavior* of applications by analyzing events that are generated during their *normal* execution [38]. So, when such a model is developed then spare events can be analyzed partially in order to observe any deviations. Consequently, such deviations are adequate to indicate the presence of maliciousness. Next we will present the two dominant types of behavior-based malware detection the Anomaly-based Detection and the Specification-Based Detection explaining their functionality and discussing their pros and cons.

(A) Anomaly Based Detection Methods

Malware detectors that utilize anomaly-based detection techniques, base their method for detection on models of *normal behavior* of users and applications, called *profiles* [28]. Thus any violations to this kind of rules indicates an attack. Additionally utilizing such methods a malware detector is not restricted to what is known till now where can be detect any abnormal behavior whether is aprt of the model or not. However, using this technique may results in higher false positive rates, as newer benign application may exhibit a behavior different from the older ones.

As we referred in the previous section, in behavioral detection and more precisely in the anomaly-based one, there do not exist any *a priori* assumptions about applications. In contrast, the behavior profiles are built analyzing system call invocations during a normal execution by collecting distinct fixed-length system call sequences [38]. So, as easily on can understand if during the execution of the program under inspection the produced system call sequences compared to the pre-recorded exhibit a variation then this is an event that indicates a possible malware existence.

Similarly to signature-based detection we divide the anomaly-based one into static and dynamic. In Static Anomaly-based Detection the detection of malware relies on characteristics of suspicious file's structure, providing thus the ability to not execute the host program [28]. On the other hand, in Dynamic Anomaly-based Detection, the detection of malware relies on the gathered information of malware's execution. So, any profile inconsistencies are caught in the detection phase during monitoring and compared with the learned profile conclude to the detection of malware. In Figure 3.5 we cote a simple example of the behavior-based detection method we discussed above.

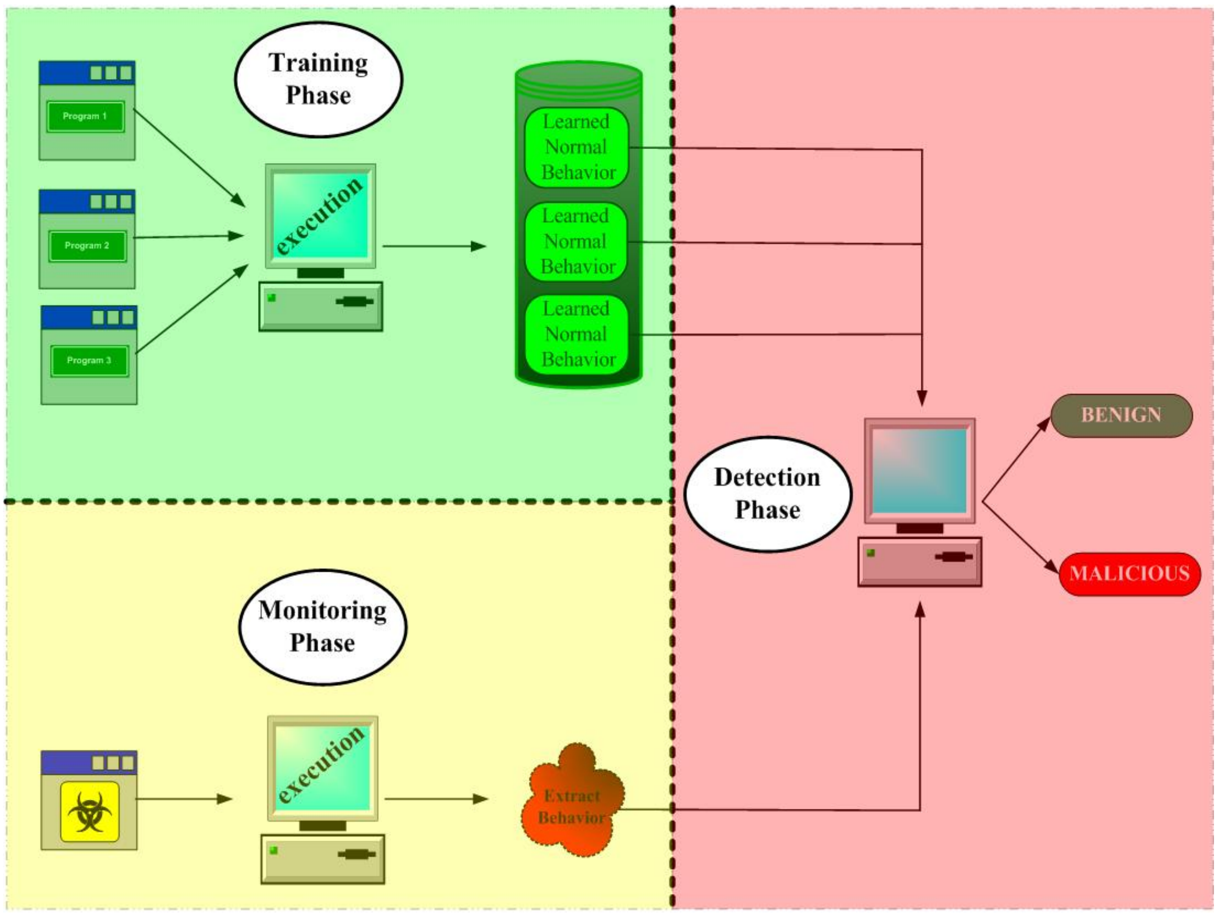


Figure 3.5: Visualization of Behavior-based Detection

(B) Specification Based Detection Methods

Malware detectors that utilize misuse-based methods are based upon descriptions of attacks (*signatures*) while they try to match data logged during the execution of a program as clues of a modeled attack. As easily one can understand there exist the same drawback as in traditional signature-based detection where only the satisfaction of *a priori* specified models indicates an attack. As we referred above, the main drawback of Anomaly-based

Detection techniques is the high false positiv rates exhibited through detection. Thus, in order to mitigate this limitation there has been proposed a type of Anomaly-based Detection the Specification-based Detection. Specification-based Detection approximates the requirements (*specifications*) for a system or an application running on the end-point, instead of its implementation [28]. In this type of behavioral detection the whole process relies on, either manually written or through static analysis, application-specific models [38]. So, the main goal is the development of a *rule set* specifying the valid behavior that should be exhibited by any running application [28]. Thus, if during the monitoring of an application a non-conforming system-call is invoked then this is a clue for the existence of a malware leading to detection. However, in Specification-based Detection there is exhibited another drawback that is the very limited capability of generalization from the pre-defined specifications. As someone could expect, if the approach uses the run-time behavior then the type of Specification-based Detection is defined as Dynamic, whereas Static Specification-based Detection relies on structural characteristics of the program respectively.

3.3 Graph-Based Detection Methods

Having already all the necessary information and an adequate background on signature-based and behavior-based detection techniques, we are able in this section to proceed to the presentation of one of the most powerful structures used for malware detection, the graphs. Using graphs the behaviours can be modeled efficiently and additionally, because of its structure, a graph can be used as signature of a malware. In this section we will discuss the graph-based detection methods, presenting indicative examples from the use of the Control-Flow Graphs (CFGs), the Function Call Graphs (FCGs), and also the System-call Dependency Graphs that consist the main tool that we utilize in our approach as we will discuss in the corresponding chapter.

3.3.1 Malware Detection using Control Flow Graphs

Control flow describes the possible execution paths of a program or a procedure and is represented as a directed graph the so called Control-Flow Graph (CFG) or simply flow-graph . Consequently, when such an abstract representation depicts the internal control flow of a procedure is generated a flow-graph, while when depicts the control flows between procedures is generated a call graph respectively [11].

As referred in [15, 18], most automatic analysis tools utilize an abstract representation of malware's structure such as the Control Flow Graphs (CFGs). According to [9], a *Control Flow Graph* is composed of *linked nodes* of one of the following types *jmp* (non-conditional jump), *jcc* (conditional jump), *call* (function call), *ret* (function return), *inst* and *end*, constructing the graph as presented in Figure 3.6. More precisely as referred in [37], each node of the Control-Flow Graph represent a sequence of instructions that are not interrupted from any jump instructions, the so called *basic blocks*. Accordingly, an edge from block u to block v represents the flow of control from block u to block v. Summing, as defined in [15], a *basic block* B is a maximal sequence of instructions $\langle I_i, ..., I_m \rangle$ containing at least one control flow instruction at its end. Let V be the set of Bs for a program P and $E \subseteq V$ ×V×$\{T, F\}$, be the set of control flow transitions between basic blocks. then the directed graph CFG(P) =$\langle V, E \rangle$ is called *control flow graph*. The graph-based representations are of major importance since they are able to

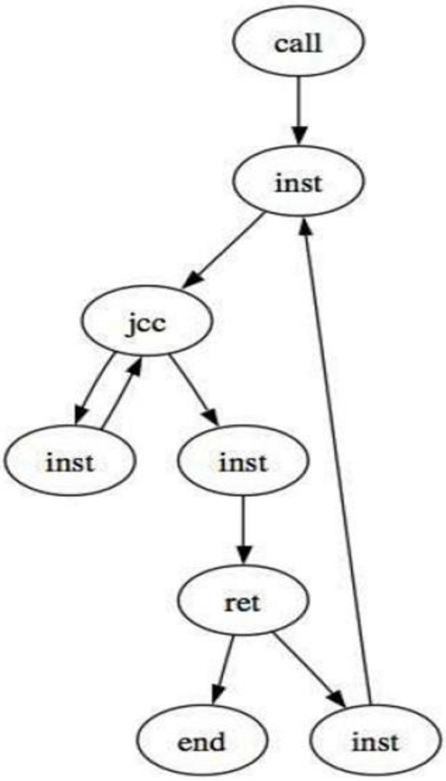


Figure 3.6: Control Flow Graph Representation [9]

capture different execution paths of the program under inspection [24]. Additionally, the nodes of the graph can store the instructions and values while they can be interpreted according to more generic semantics [11].

The use of CFGs as signatures for malware detection is based on sub-graph isomorphism that is theoretically NP-complete. However its complexity can be reduced in the detection context. Actually, except the indirect jumps and the returns all the other nodes of a CFG have a bounded number of typically one or two successors [24]. Additionally, isomorphism remains sensitive in morph techniques as code permutation or injection (see 1.3.2) that impact the graph, however these limitations can be addressed by compiler optimizations as referred in [24].

A typical approach for signature creation is presented in [11]. In order to generate a signature from a CFG, depth first order can be utilized, consisting thus a signature by listing the graph edges for the ordered nodes using ordering as node labels and finally representing the signature as a string (see Figure 3.7). Additionally approximate matches of flow-graph based characteristics can be used in order to detect a broader range of malware variants. Finally, in order to proceed to malware detection the proposed approach make use of the malware database that stores the string signature produced as described above together with a normalized weight computed for each procedure .

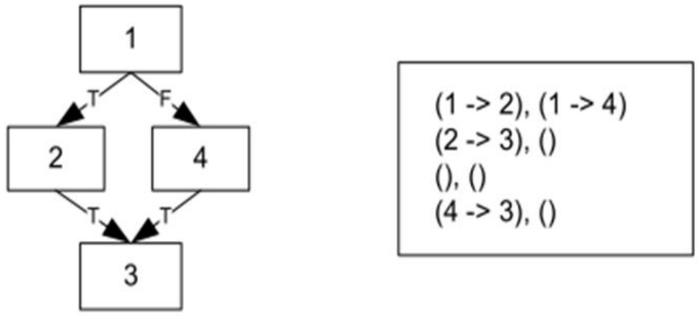


Figure 3.7: String signature derived by CFG [11]

3.3.2 Malware Detection using Function Call Graphs

A Function Call Graph (FCG) is a directed graph that its vertices depict the functions that an executable binary includes and its edges represent the interconnection between the functions according to their calls (see Figure 3.7). As referred in [33], the call graph can be gathered from a binary executable through *static analysis*. Actually, disassembly tools like the ones we referred in 2.1.2 (i.e. IDA Pro) are utilized after the removal of the obfuscation layers (i.e. unpacking).

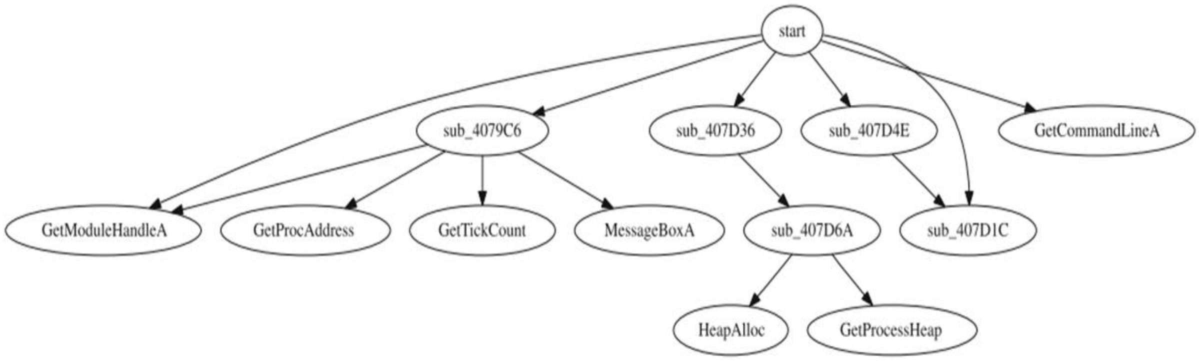


Figure 3.8: Function Call Graph (local and external functions) [33]

To this point we ought to notice that the functions, represented by the vertices may be either local functions, i.e. functions wrote by the malware author, or external functions like System or Library calls invoked during the execution of the binary. In Figure 3.8 we cite a representation of a Control Flow Graph produce by an executable as it is depicted in [33]. The nodes of this graph represent local and external functions. Specifically, the one's whose name starts with the prefix *sub* refer to local functions.

3.3.3 Malware Detection using System-Call Dependency Graphs

The behavior of a program can be modeled based upon system-call dependencies as the capture its interaction with its hosting environment, the operating system. As easily one can understand, a representation that captures a sequence of system calls would be liable since any reorder or addition of one or more system calls could change the sequence, so a more flexible representation that would capture their in between relations , as a graph in example, would satisfy that demand [35] Thus a program's behavior can be modeled by a *behavior graph*. A behavior graph is a directed graph generated from system call traces collected during the execution of the program under inspection, while their arguments indicate their relations [46].

To start with, we should define the term behaviour as its effect on operating system's state. As referred in [23] most malware relies on system calls in order to deliver their payload, and thus system calls are able to representations of malwares intent omitting useless implementation artifacts. In almost all of the works the program's under inspection behavior is represented as a graph, the so called behaviour graph.

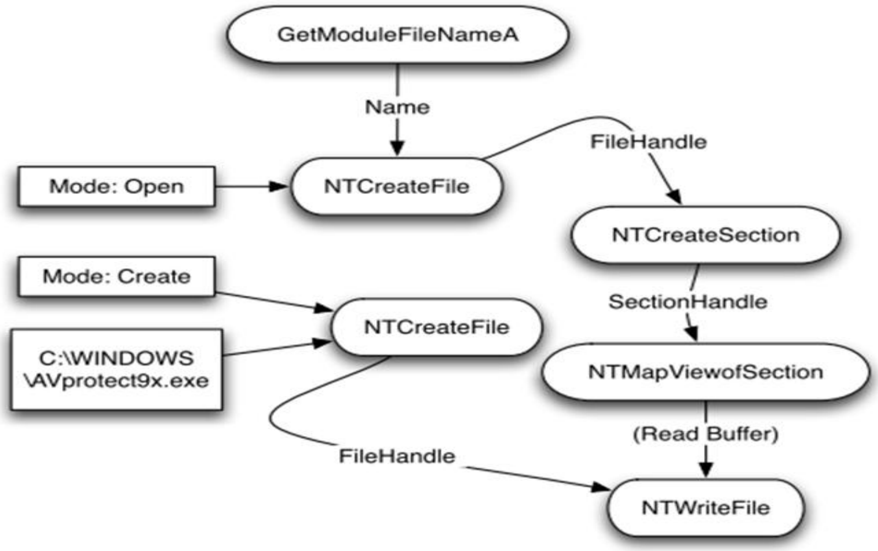


Figure 3.9: Behavior Graph from malware NetSky [35]

As easily someone can suppose, the nodes of this graph are the system calls captured by the programs trace during its execution time utilizing *taint analysis* (see 2.2.2). The most straightforward approach to define an edge in a behavioral graph is the one discussed in [35] where an edge introduced from node x to node y when the system call referenced by y uses as input argument the argument produced as output from system call referenced by x. As a result the existence of an edge in such a graph represents the data dependecy between two system calls. Such dependecies can be monitored as we mentioned above through the tainting of data during taint analysis. In Figure 3.9 we cite the behavior graph depicting the dependeces between system calls captured throug taint analysis dyring the execution of NetSky malware as presented in [35].

Now, let us proceed with some proper definitions about the behavioral graphs as they are presented in [35, 17, 23, 46]. Compiling the definitions of behavioral graph presented in [35, 17, 23] we concluded at a global structure that we present next. Generally speaking, the behavioral graph includes, except from its basic components that are its vertex-set V and its edge-set E, two labeling function that are responsible for the association, the first one of vertices and edges with system-calls and their in between dependencies respectively, and the second one of vertices and edges with some constraints on operations and dependencies. Before we start we ought to refer some preliminaries about the vertices and edges as the fact that such graphs are *Directed Acyclic Graphs* (DAG) as defined a *malicious specification -malspec* [35] where the nodes are labeled using system-calls from an alphabet Σ and edges labeled using logic formulas from a logic L_{Dep}. Thus, we proceed by citing the definitions of malicious specification and the corresponding behavior grarph and as they are presented in [17] and [23] respectively.

Definition 3.1: A *malicious specification* is a *Directed Acyclic Graphs* (DAG), with nodes labeled using system calls from an alphabet Σ and edges labeled using logic formulas from a logic L_{dep}. The malicious specification (*malpsec*) M is written as $M = (V, E, \gamma, \rho)$, where:

- V is the vertex-set and E is the edge-set, $E \subseteq V \times V$,
- γ associates vertices with symbolic system calls, $\gamma : V \rightarrow \Sigma \times 2^{Vars}$ and
- ρ associates constraints with nodes and edges, $\rho : V \cup E \rightarrow L_{Dep}$.

Definition 3.2: A behavior graph is a *data dependence* graph $G = (V, E, \alpha, \beta)$, where:

- the set of vertices V corresponds to operations from Σ,
- the set of edges $E \subseteq V \times V$ corresponds to *dependencies* between operations,
- the labeling function $\alpha : V \rightarrow \Sigma$ associates nodes with the name of their corresponding operations and
- the labeling function $\beta : V \cup E \rightarrow L_{Dep}$ associates vertices and edges with formulas in some logic L_{Dep} capable of expressing constraints on operations and the dependencies between their arguments.

The dependencies we referred above are classified into three categories [23, 17]:

- *def-use dependence:* A def-use dependence expresses that a value output by one system call is used as input argument to another system call.
- *value dependence:* A value dependence is a *logic* formula that expresses the conditions placed on an argument (values) of one or more system calls, describing any non trivial data manipulations performed by the program between system-calls.
- *ordering dependence:* Finally, an ordering dependence between two system calls expresses that the first system call must precede the second system call.

Next, we proceed by presenting a simple detection example that uses behavior graphs as presented in [17]. The term *malspec* referred in [17], actually refers to a *sub-graph* of the malware's dependence graph that *does not appear* in any of the benign dependence graphs. As mentioned in the corresponding work, the simplest solution would be to choose the whole graph, however the resulting malspec would be too large and *too specific* to the malware sample. Thus, the *minimal contrast subgraph* is utilized in order to generalize and make as small as possible the malspec. A minimal contrast sub-gaph (MCS) is a smallest sub-graph of a graph that does not exist in another graph. So, a contrast sub-graph of G_1, G_2 is a sub-graph of G_1 that is not *sub-graph isomorphic* to G_2 and consequently is minimal iff none of its sub-graph is contrast sub-graph. Thus, a malspec can be thought as a minimal contrast sub-graph of a malware's dependence graph and a benign program's dependence graph. The computation of MCS can be done using the Ting-Bailey algorithm as proposed in [58]. Actually the algorithm works as follows, first *maximal common edge sets* are computed between two graphs using backtracking tree, next maximal common edge sets are unioned together and minimal traversals of theri complements are computed in order to yield minimal contrast edge sets and finally the minimal contrast edge sets are unioned with the minimal contrast vertex sets to to produce a complete set of minimal contrast sub-graphs.

Chapter 4

Malware Classification

In this chapter we will discuss some major topics concerning malware evolution. Specifically, we will focus on the evolution of malware according to how malware families share common characteristics through their commonalities in their specimens' source codes resulting from phylogeny. Additionally we will discuss the importance of malware classification into malware families and how this grouping is able to increase the detection rates through the leverage of signature generalization when a signature can be applied globally to the members of a malware family decreasing subsequently the need for new signature production for individual malware.

4.1 Philogeny

One of the most important issues concerning the protection against malware's spread is how the AV production industries will be able to manage the thousands of suspicious files arriving for analysis every and most of the are malware. Obviously, the construction of individual signature for each malware sample does not consists an effective solution. As referred in the literature, and exists as a general sense, each individual malware is not developed from scratch, since if so, then there would not appear so many *new* malware samples every day to be analyzed. Contrary, malware authors almost always, exception consist the targeted attacks (i.e. STUXNET), either share their code or use mutation engines in order to develop their malware or to morph them respectively. This work is grown based upon the wider axes of malware analysis including the components of pure analysis, malware detection in terms of determining if an object is malicious or benign, malware classification in terms of classifying a malware specimen into one malware family. As we referred in the introduction, the procedures of malware analysis, detection and classification are strongly connected, however, malware classification is also connected to another sense, concerning how malware families are interconnected and how malware is evolved sharing and distinguishing characteristics between samples, the so called *phylogeny*.

As referred in [29, 30], various types of malware (i.e. viruses, worms, trojans etc.) share common characteristics, so between them as to other previously seen malware. Leveraging this observation, a malware analyst is in position to build a phylogeny model that capturing this relations to be able to contribute in a proper family naming or to the development of more flexible detection and classification techniques.

Malware authors have developed a network of code sharing, exchanging code for the development of their malware. Every day new malware strains are released that in almost all of the cases are mutations of previously seen malware, either including code through code reusability in terms of recycling, or by fixing bugs existed in previous versions. So, easily someone can understand that this effort of malware authors to cooperate in malware development can be leveraged from malware analysts in order to develop more efficient detection techniques, as we mentioned above. The information, provided from the build of a phylogeny network that captures the share of code in malware development, may be proven quite precious on understanding the relations of malware and how new strains are actually evolutions of older ones. Thus, these relations can be interpreted either to mutations caused due to any need for change to malware's functionality, or to mutations caused as a result of morphing engines used in malware's detection avoidance i.e polymorphism or metamorphism (see sections 1.3.1-2).

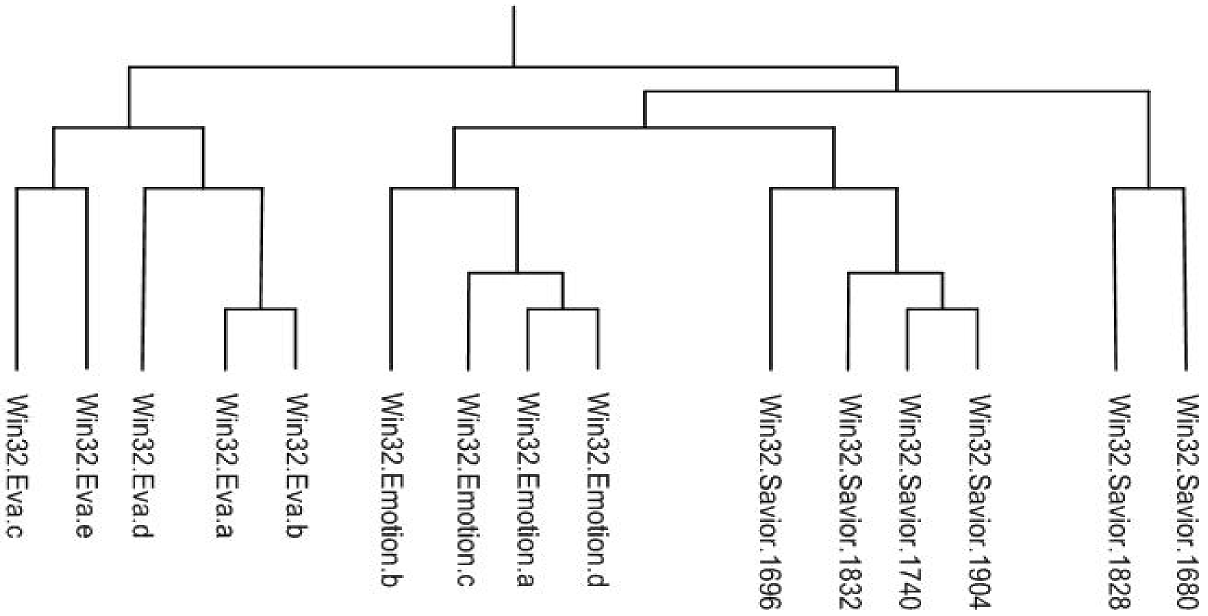


Figure 4.1: Dendrogram Representing Phylogeny Between Individual Specimens [61]

So, the main goal in building a phylogeny model is to examine software artifacts in order to observe where there exist commonalities and differences in order to construct an *evolution history* [29]. A quite convenient representation of malware's evolution could be a tree-like one as a *dendrogram* [61] (see Figure 4.1) where malware samples have been clustered according to a technique that detects commonalities between specimens.

4.2 Software Similarity

Software as a general term can be classified into two categories, malicious or benign, according to the existence or not of maliciousness to its functionality. So, if a program belongs into the class of malicious programs then it has inherited the characteristics of its mother class, the software. Consequently, malware just like the software has the ability to evolve. As we referred in the previous section, a family of malware can be evolved as to fulfill some new added requirements or simply because of some bug-fixes. So, in order to be able to determine if a given *unknown* malware is actually an evolution of a previously seen one, in other words is member of a specific family, we need to define a *method* that will be in position to determine according to a given input and a background knowledge if this specimen is member of a known family. Thus, a rational approach could be to compare the *similarity* of the given object against some pre-classified objects.

As we have all ready describe, the traditional signature-based detection techniques are unable to detect morphed malware, and thus an approach of creating distinct signature for each individual malware could be ineffective and for sure counterproductive. So, as the need of family level signature construction grows we need to develop techniques that are able to classify with high accuracy rate a given unknown malware, since it is not a brand-new one, to a malware family. Thus, in order to address these needs, there have been developed a series of techniques spare in the literature that utilize either data mining techniques or are graph-based ones with orientation to the behavioral graphs (see sections 3.3.3 and 4.4).

Generally speaking, the *software similarity problem* focuses on determining the similarity between two programs [11]. Thus, the result of a method that computes metrics for such purposes result in a value between 0 and 1, where values near 0 indicate low similarity while values near to 1 indicate high similarity based on a threshold value. An approach to software similarity problem using known similarity metrics on profiles produced by characteristics of two objects (i.e. a recurring pattern existed in a known malware and its variations) may lead to the immediate detection of new variants straight from their release, to generic signature construction and in the observation of commonalities and relationships between different malware families [61].

4.3 Classification of Malware into Families

As we referred above the construction of distinct signatures for each individual malware is inefficient and counterproductive. Thus, the grouping of each individual malware into families that exhibit similar characteristics (i.e. *similar behavior*) is a rational and effective solution. The main requirement for clustering malware into families is for the members in each family to exhibit the highest similarity with the other members belonging to the same family and the minimum similarity with members belonging to other families. However, there exists families of malware that are of the same type (i.e. bots, bankers, downloader etc.) meaning that in general exhibit the same behavior, resulting to misclassifications.

Everyday thousands of files arriving to AV industries in order to be analyzed. In order to make analysis more efficient and to be able to handle large amounts of data, a proper clustering of malware that exhibit similar behaviors is needed so to not spent time in analyzing a malware that is a variant of a previously seen one, as to create more generic signatures that satisfy the detection of any member belonging to a specific family [33, 6].

Label	**McAfee**	**Trend**
A	Not detected	W32/Backdoor.QWO
B	Not detected	W32/Backdoor.QWO
C	W32/Mytob.gr@MM	W32/IRCBot-based!Maximus
D	W32/Mytob.gr@MM	Not detected
E	PWS-Banker.gen.i	W32/Bancos.IQK
F	IRC/Flood.gen.b	W32/Backdoor.AHJJ
G	W32/Pate.b	W32/Parite.B
H	Not detected	W32/Bancos.IJG
I	IRC/Generic Flooder	IRC/Zapchast.AK@bd
J	Generic BackDoor.f	W32/VB-Backdoor!MAximus

Table 4.1: Spare Malware Samples [4]

One of the most applicable approaches in malware classification is to extract invariant characteristics of each sample in order to construct a profile. Then, comparing the profiles of given any two samples using a similarity metric is straightforward to determine if two samples are similar or dissimilar and consequently to classify them into the same class or not, respectively . So, in Table 4.1 we cite a table from [4] where are presented some initially uncorrelated malware samples labeled by two AV vendors. In Table 4.2 there has been computed the Normalized Compression Distance (NCD) of them everyone with each other.

	A	B	C	D	E	F	G	H	I	J
A	0,06	0,07	0,84	0,84	0,82	0,73	0,8	0,82	0,68	0,77
B	0,07	0,06	0,84	0,85	0,82	0,73	0,8	0,82	0,68	0,77
C	0,84	0,84	0,04	0,22	0,45	0,77	0,64	0,45	0,84	0,86
D	0,85	0,85	0,23	0,05	0,45	0,76	0,62	0,43	0,83	0,86
E	0,83	0,83	0,48	0,47	0,03	0,72	0,38	0,09	0,8	0,85
F	0,71	0,71	0,77	0,76	0,72	0,05	0,77	0,72	0,37	0,54
G	0,8	0,8	0,65	0,62	0,38	0,78	0,04	0,35	0,78	0,86
H	0,83	0,83	0,48	0,46	0,09	0,73	0,36	0,04	0,8	0,85
I	0,67	0,67	0,83	0,82	0,79	0,38	0,77	0,79	0,05	0,53
J	0,75	0,75	0,86	0,85	0,83	0,52	0,85	0,83	0,52	0,08

Table 4.2: NCD Computation of Malware Samples [4]

Finally in Figure 4.2 according to the aforementioned metric the previously uncorrelated malware samples are combined in clusters ($c_1 : c_9$) including malwares exhibiting the minimum NCD between them. As we can observe through this process, finding some characteristics among malware samples can afford quite valuable information about malware's behavior and to a greater extent, about the evolution of malware. Such information then can be leveraged to build detection and classification techniques that will be more accurate while being more elastic since depend on structure's abstractions.

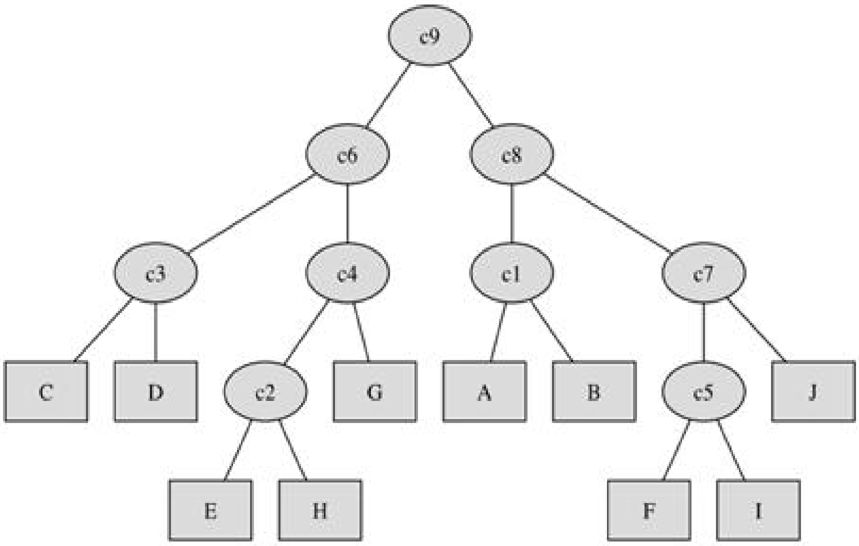


Figure 4.2: Clustering of Malware Samples according to NCD[4]

4.4 Graph-Based Classification Methods

As we referred in previous sections, malwares that belong to the same families tend to exhibit the same or at least a similar behavior. Consequently, the ability of recognizing commonalities among samples that belong to the same family leads to the development of techniques that immediately detect both known and unknown malware based on their abstract manifestations such as their behavior. Since, as we mentioned in chapter 3, graphs are from their nature quite adequate to represent such representations we proceed by presenting the application of graphs in malware's behavior representation and their use in automated classification of unknown malware samples to malware families. Similarly to section 3.3, next we will present two indicative examples from the use of Function Call Graphs (FCGs) and System Call Dependency Graphs for the depiction of malware's behavior in order to classify a given sample.

4.4.1 Malware Classification using Function Call Graphs

In [33] Function Call graphs (FCGs) are utilized in order to compare and classify malware samples, according to their structural similarity, to malware families. To this point we ought to remind that Function Call Graphs Are directed graphs that their vertices represent the functions of an executable, while their edges represent their calls (see section 3.3.2). Specifically, having composed the CFGs from two executables the the computation of similarity may include the search for graph isomorphism or the maximum common sug-graph (MCS) or the minimum edit distance (GED). The classification of an unknown sample can be achieved by computing the distance between the sample and each cluster's center μ_{C_i}, assigning the sample to the cluster with the minimum distance.

So, in order to compute the similarity between the Function Call Graphs of the members in a cluster and hence the distance of an unknown sample's Function Call Graph from the center of a cluster, they utilize the aforementioned graph edit distance (GED), that for any given two graphs G, H it is computed as:

$$\lambda_\phi(G, H) = VertexCost + EdgeCost + RelabelCost \ , \tag{4.1}$$

where VertexCost and EdgeCost is the number of inserted or deleted vertices or edges respectively and RelabelCost is the number of mismatched vertices (functions). Thus, having already computed the graph edit distance, the similarity of the two graphs is computed as:

$$\sigma(G, H) = \frac{\lambda_\phi(G, H)}{|V(G)| + |V(H)| + |E(G)| + |E(H)|} \tag{4.2}$$

Finally, the center of a cluster is selected as the graph that has the more similarity with all other graphs in the cluster. Then the computation of the distance between a sample and the cluster's center is the calculation of the Euclidean distance as:

$$min \sum_{i=1}^{k} \sum_{\chi \epsilon C_i} \sigma(\chi, \mu_{C_i}) \tag{4.3}$$

where k is the number of predefined clusters, and χ is the unclassified sample.

4.4.2 Malware Classification using System-Call Dependency Graphs

Another graph-based method for classifying malware is that of leveraging execution trace in order to construct a behavioural graph, actually by representing system call dependencies. In [46] is presented an approach that utilizes behavioral graph matching in order to classify an unknown malware sample into a malware family.

Specifically, the behavior graph, also called Dynamic System Call Dependence Graph (DSCDG), is extracted during the suspicious program's execution, representing the system call sequences and their in between dependencies. Individual system calls are captures by intercepting every SYSENTER instruction while the sequence is obtained by their traces when matching their arguments comparing both their type and value [46]. The focusing in arguments is mostly centralized in specific ones such as handles. Thus, when a handle produces as output from one system call (S_1) and then is feeded as input to another one (S_2) then an edge is added from node S_1 to node S_2.

Thus, the behavioral graph (DSCDG) is defined as : $G = (N, E, \mu, u)$, where N is the vertex set (System Call : $S_i \ \epsilon \ N$), E is the edge set (dependency: $S_i \rightarrow S_j \ \epsilon \ E$), μ is a *node labeling function* defined as $\mu \ : \ N \longrightarrow L_N$ assigning system calls to nodes and u is an edge labeling function respectively, that is defined as $u : E \longrightarrow L_E$. The main difference between μ and u is that u also describes the dependence of two system calls according to their arguments.

Finally, in order to compute the similarity between two behavioral graphs and hence utilize it to classify an unknown malware sample the *maximal common sub-graph* hs to be computed before they proceed with the computation of the similarity formula. So, assuming that are given two behavioral graphs G_1 and G_2 as $G = (N_1, E_1, \mu_1, u_1)$ and $G = (N_2, E_2, \mu_2, u_2)$, then the $G' = (N', E', \mu', u')$ is called *common sub-graph* of G_1, G_2 iff there exists sub-graph isomorphism from G' to G_1 and from G' to G_2, while is called *maximal common sub-graph* (MCS) when there is no other common sub-graph of G_1 and G_2 that include more nodes that G' [46]. Finally the similarity (*distance*) between two given behavioral graphs can be computed as :

$$D(G_1, G_2) = 1 - \frac{|G_{MCS}|}{max(|G_1|, |G_2|)} \ , \tag{4.4}$$

where $|G|$ is the size of the vertex set.

Chapter 5

Our Model

In this chapter we will present the model that we have designed in order to detect and classify malicious software based on their system-call dependency graphs. As referred in the literature, system-call dependency graphs exhibit a great potential of representing malware's functionality and behavior. Specifically, since in their general form, graphs constitute a software's abstraction, they have the ability to capture its interaction with its environment, in this case the operating system. So, to start with, we firstly discuss the representation of malware as a graph, and to be precise the system-call dependency graph produced during its run-time through taint analysis (see chapter 2.2.1). Secondly, we make an introduction to the techniques (such as: *Jaccard Index, Cosine Similarity, Tanimoto Coefficient etc.*) that we will be based on, concerning those used in computing graph similarity in most of time on their intermediate auxiliary representations. Then in the third section we present and discuss our graph-based proposed technique that we utilize in malware detection and that leverages system-call dependency graphs. Finally in the last sectino we will present and describe our also graph-based malware classification technique that leverages system-call dependancy graphs too, and its application on classifying an unknown malware into one malware family.

5.1 Graph Representation of Malicious Software

The core works that we base our intuition in the use of system-call dependency graphs are [18, 17, 23] and [3]. To this point, we ought to underline that our work is totally complement to the aforementioned ones, while we have developed a totally novel intermediate graph representation that exhibits an auxiliary functionality in our model while it can capture and represent a much more abstract depiction of malware's behavior. As we will describe later in this section, we use the well known classification of system-calls into classes of similar functionality, constructing finally a graph that its vertices actually are super-vertices containing the system-calls captured in the system-call dependency graph and are from the same class. This sophisticated hyper-abstraction of malware's system-call dependency graph provides us with the ability of a wider generalization depicting what actually performs *in general*.

As we referred in previous chapters, the use of traditional string signature-based detection is inadequate in detecting morphed malware. So, in order to develop more elaborate techniques for malware detection and also for classification, the use of more abstract structures need to be utilized. Thus, we leverage the use of graphs, since as referred in the literature there have been widely used for this purpose. Indicative and also quite successful examples constitute the Function Call Graphs (FCGs), the Control Flow Graphs (CFGs) from the aspect of static malware analysis and also the System-Call Dependency Graphs (SCDGs) or behavioral graphs from the aspect of dynamic analysis. To this point we ought to notice that we will utilize the use of System-Call Dependency Graphs since we want to leverage the depiction of the behavior of a malware concerning its environment. Additionally, System-Call Dependency Graphs provide us with information about the real behavior (actions) performed by the testing malware instead of the other kinds of graphs that provide information about probable actions since in static analysis the sample has not been executed.

5.1.1 System-Call Dependency Graph Construction

Generally speaking, the actions performed by a program depicting its behavior, rely on system-calls in order to be executed. So, capturing the system-calls performed during the execution of a malware we can represent its behavior interpreting this information with a graph while we are able to determine malware's intent independently of any implementation artifacts.

In order to result in the construction of a System-Call Dependency Graph primarily some operation need to be performed. First the suspicious sample needs to be executed in a contained environment (i.e. a virtual machine). During its execution time taint analysis is performed in order to capture system-call traces. As we referred in section 3.3.3 three types of dependence are involved in order to connect system calls. Specifically in order to create the edges of a System-Call Dependency Graph, through taint analysis are captured the system calls and the arguments they exchange as input/output where the output arguments of one system call are used as input arguments to another one.

So, the constructed System-Call Dependency graph has as its vertex set all the system-calls that took place during the execution of the suspicious sample while its edge set consists from the pairs of system call that passed argument the one to the other during the execution.

Next, we proceed by citing a simple example that includes the system-call trace obtained through taint analysis during the execution of a sample from malware family Hupigon, and we explain how the SCDG is constructed after the whole process. As we will also refer to chapter 6, the data set we utilize in order to evaluate the implementation of our proposed model is downloaded from Domagoj Babic's personal web-page and is the same data set utilized in [3] for the evaluation of the corresponding model. So next we describe how is constructed a graph according to the description provided in the data-set. Before we continue we ought to explain the contents of each column in Table 5.1. In the first column there is placed the ID of each system-call captured during the analysis, while in column 2 is placed the name of each system-call. Finally, in column 3 are cited the number (in terms of cardinality) of input arguments for each system-call, while in column 4 are cited the number of output arguments for each system-call.

ID	System-call Name	InArgs	OutArgs
0	NtOpenSection	2	1
1	ACCESS-MASK	0	1
2	POBJECT-ATTRIBUTES	0	1
3	NtQueryAttributesFile	1	1
4	NtQueryAttributesFile	1	1
5	NtQueryAttributesFile	1	1
6	NtQueryAttributesFile	1	1
7	NtQueryAttributesFile	1	1
8	NtQueryAttributesFile	1	1
9	NtQueryAttributesFile	1	1
10	NtQueryAttributesFile	1	1
11	NtQueryAttributesFile	1	1
12	NtQueryAttributesFile	1	1
13	NtRaiseHardError	5	0
14	NTSTATUS	0	1
15	ULONG	0	1
16	PULONG-PTR	0	1
17	HARDERROR-RESPONSE-OPTION	0	1

Table 5.1: System Call Traces

Having already captured the system-calls that took place utilizing taint analysis and hence having composed the vertex set, the next step is to create the edge set by connecting each pair of system-calls that exchange arguments. As depicted in the next table a tuple of type $\{sc_1{:}I,\ sc_2{:}III\}$ indicates that the system-call sc_2 takes as her fourth input argument the second output argument of system-call sc_1.

Now, let us give an easy and quite simple example. Let us suppose that we have a system-call with ID =10 and has 3 input arguments and 2 output argument, then when it appears as 10:1 in the *from* side of an edge it indicates that the system call 10 *passes* as output her second (because this number is a zero-based index) output argument to another system-call, while when appears 10:1 in the *to* side of an edge it indicates that the system call 10 *receives* as her second input argument the argument produced from another system-call. In other words if we have two system-calls the previous one and another one with ID=12 and who has 2 input and 5 output argument then the expression (10:0, 12:1) is interpreted as the first output argument of system call 10 is passed as the second input argument to system call 12, while the expression (12:4, 10:2) is interpreted as the fifth output argument from system-call 12 is passed as the third input argument to system-call 10.

Trace	**From**	**out.idx**	**To**	**in.idx**	**assign type**	**edge type**
1:0,0:0	1	0	0	0	$sc_0.in(0) \longleftarrow sc_1.out(0)$	$sc_1 \longrightarrow sc_0$
2:0,0:0	2	0	0	1	$sc_0.in(1) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_0$
2:0,3:0	2	0	3	0	$sc_3.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_3$
2:0,4:0	2	0	4	0	$sc_4.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_4$
2:0,5:0	2	0	5	0	$sc_5.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_5$
2:0,6:0	2	0	6	0	$sc_6.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_6$
2:0,7:0	2	0	7	0	$sc_7.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_7$
2:0,8:0	2	0	8	0	$sc_8.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_8$
2:0,9:0	2	0	9	0	$sc_9.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_9$
2:0,10:0	2	0	10	0	$sc_{10}.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_{10}$
2:0,11:0	2	0	11	0	$sc_{11}.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_{11}$
2:0,12:0	2	0	12	0	$sc_{12}.in(0) \longleftarrow sc_2.out(0)$	$sc_2 \longrightarrow sc_{12}$
14:0,13:0	14	0	13	0	$sc_{13}.in(0) \longleftarrow sc_{14}.out(0)$	$sc_{14} \longrightarrow sc_{13}$
15:0,13:1	15	0	13	1	$sc_{13}.in(1) \longleftarrow sc_{15}.out(0)$	$sc_{15} \longrightarrow sc_{13}$
15:0,13:2	15	0	13	2	$sc_{13}.in(2) \longleftarrow sc_{15}.out(0)$	$sc_{15} \longrightarrow sc_{13}$
16:0,13:3	16	0	13	3	$sc_{13}.in(3) \longleftarrow sc_{16}.out(0)$	$sc_{16} \longrightarrow sc_{13}$
17:0,13:4	17	0	13	4	$sc_{13}.in(4) \longleftarrow sc_{17}.out(0)$	$sc_{17} \longrightarrow sc_{13}$

Table 5.2: System Call Dependencies

In Table 5.2 the first column (*trace*) represents the tuple as captured from the analysis, next from column 2 to column 5 we analyze to a further extent the column one disassembling the aforementioned tuple to its components, in column 6 we present the corresponding tuple as an assignment of the values from the output argument of the one system call to the input argument of the other one. Finally, in the column 7 we represent the resulting edges that has been created from this trace. So, in example, observing the data from the Table 5.2 we can proceed by constructing the System-Call Dependency Graph that is a directed acyclic graph (DAG) as presented in Figure 5.1. The vertex set of this graph is consisted from the system-call that took place during the execution of the sample and we have captured their trace (Table 5.1) and its edge set is consisted by their in between dependencies (Table 5.2)

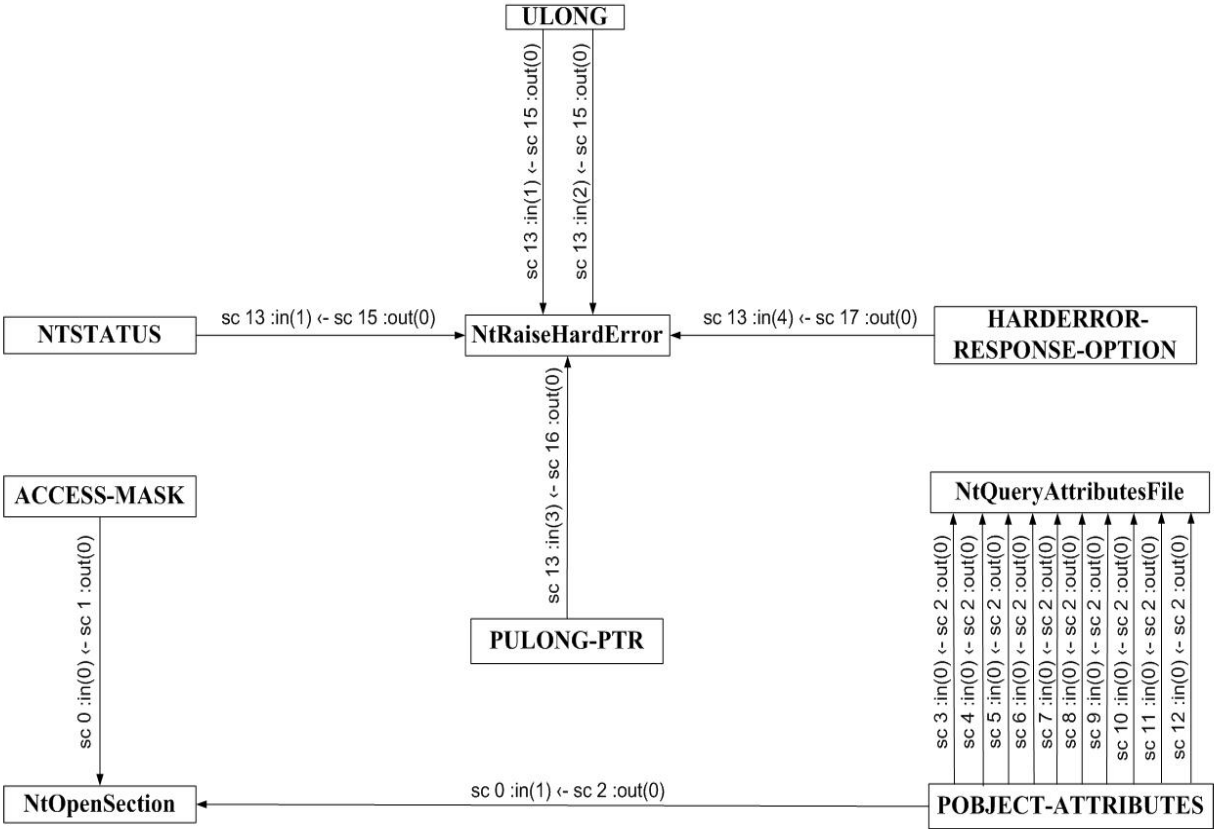

ULONG
sc 13 :in(1) <- sc 15 :out(0)
sc 13 :in(2) <- sc 15 :out(0)
NTSTATUS
sc 13 :in(1) <- sc 15 :out(0)
NtRaiseHardError
sc 13 :in(4) <- sc 17 :out(0)
HARDERROR-RESPONSE-OPTION
sc 13 :in(3) <- sc 16 :out(0)
PULONG-PTR
ACCESS-MASK
sc 0 :in(0) <- sc 1 :out(0)
NtOpenSection
sc 0 :in(1) <- sc 2 :out(0)
NtQueryAttributesFile
sc 3 :in(0) <- sc 2 :out(0)
sc 4 :in(0) <- sc 2 :out(0)
sc 5 :in(0) <- sc 2 :out(0)
sc 6 :in(0) <- sc 2 :out(0)
sc 7 :in(0) <- sc 2 :out(0)
sc 8 :in(0) <- sc 2 :out(0)
sc 9 :in(0) <- sc 2 :out(0)
sc 10 :in(0) <- sc 2 :out(0)
sc 11 :in(0) <- sc 2 :out(0)
sc 12 :in(0) <- sc 2 :out(0)
POBJECT-ATTRIBUTES

Figure 5.1: System Call Dependecy Graph

In Figure 5.1 we observe how the taint data are exchanged through the captured system calls and actually how the system call dependencies are created. So, in order to simplify this abstraction and to conclude to a final System-Call Dependency Graph, specific information is eliminated from the scheme resulting to the graph presented in Figure 5.2. In the resulting graph the vertex names are composed by the SC (stands fo system-call) followed by the corresponding system-call's ID.

To this point we ought to refer that the all the *distinct* dependencies to system-call *NtQueryAttributesFile* have been merged to one edge leading to one single vertex. However, we need to explain that we represent the graph in this way just for simplicity, because as we will refer next, the information of the number of edges from one system call to another (independently of if it is repeated) is quite valuable since we will need to use it for our model in the computation of similarity either for detection or classification.

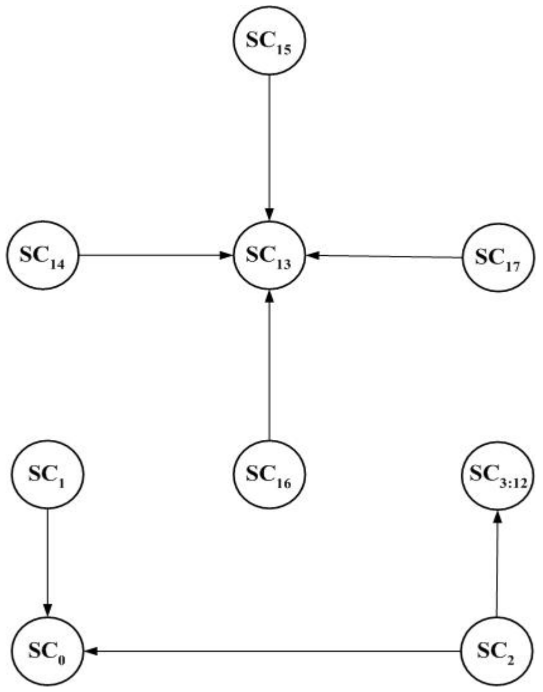


Figure 5.2: Simplified System Call Dependecy Graph

5.1.2 G^* an Auxiliary Hyper-Abstraction of SCDG

Through the literature, all the works that evolve the use of System-Call Dependency Graphs to perform malware detection and classification are utilizing the graph represented in Figure 5.2. However, despite the fact that this approach seems to works fine because of the abstraction it provides, in this work we decided to depart from the trodden, demanding even higher levels of abstraction and hence higher generalizability and elasticity. Thus, we decide to leverage the classification of each individual system-call of windows into classes of similar functionality concerning a specific resource. The aforementioned grouping of system-calls of course was not made randomly, when we used a specific tool for system-call capturing instead. To be more precise we utilized the grouping provided by the configuration file of NtTrace [45] where each system-call has a detailed description including its type. So, leveraging this quite valuable information we formed these data in order to have a mapping from each individual-system call to one group. Then having this mapping we are ready to proceed to the hyper-abstraction of each given System-Call Dependency Graph. To this point we ought to make clear that in our proposed model only the aforementioned hyper abstractions are used for both detection and classification.

Before we proceed with the definition of the Hyper-Abstraction of the System-Call Dependency Graph we ought to define the *mapping function* that is responsible for the mapping of the system calls to their corresponding groups and hence is responsible for the construction of super vertices.

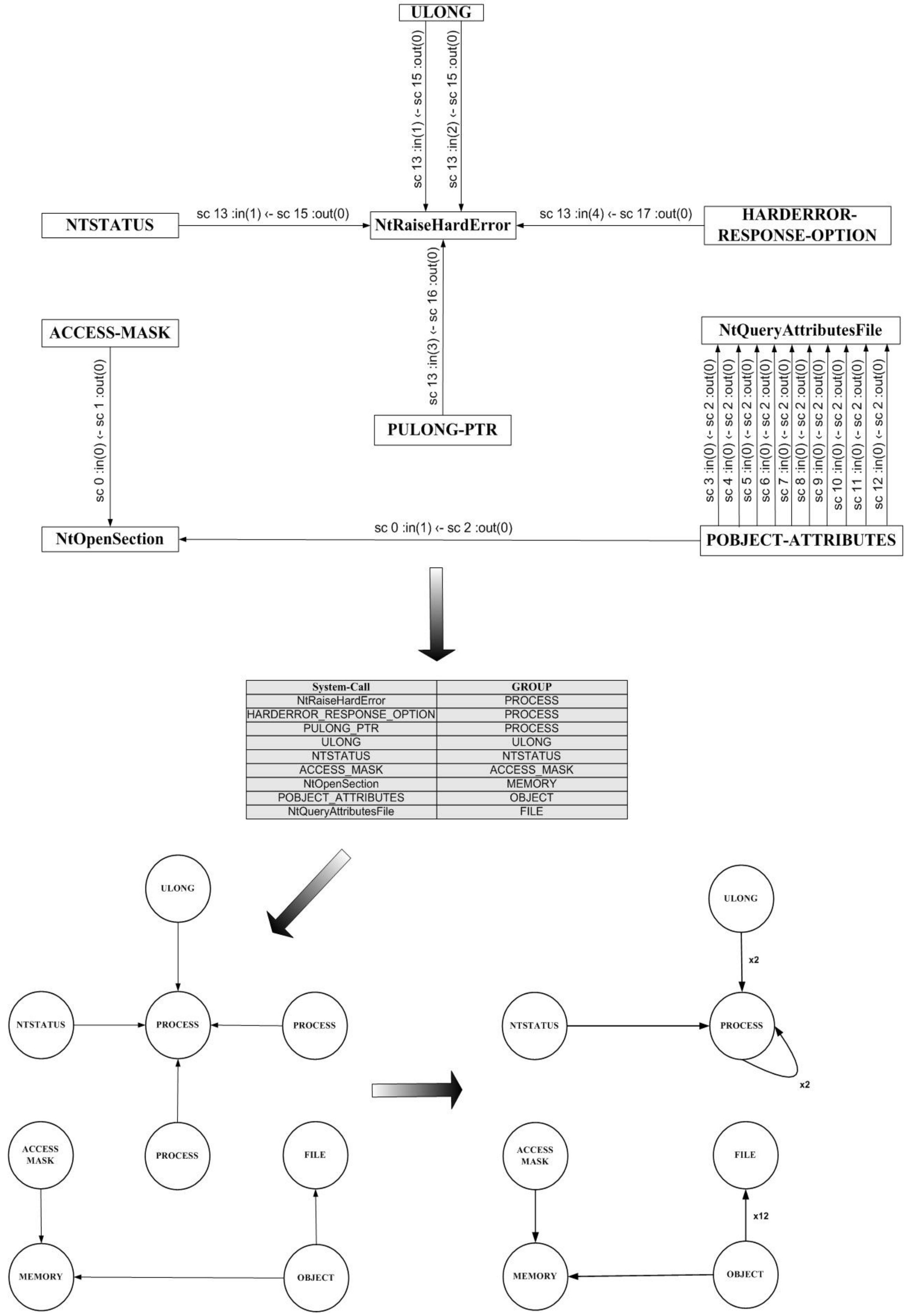
Definition 5.1 Let us assume that we are given a System-Call Dependency Graph lets say G, then a *mapping function* for the vertices of G is a transition where all vertices (System-Calls) are replaced by their group C (stands for class) resulting to multiple appearances of the same vertex.

Having already defined how the system-calls are mapped to their group the next step is to construct the G^* as a hyper-abstraction of G. In order to construct the G^* all the homonym vertices are merged to one super-vertex and any edges incoming or outgoing from or to other vertices are turned to edges between super-vertices with the same direction. Finally, duplicate edges are kept in order to indicate the importance of the intercommunication between any pair of system-call groups.

Definition 5.2 Let us assume that we are given a System-Call Dependency Graph lets say G, then a *hyper-abstraction* G^* is a graph that its vertex set is the number of distinct groups of system-calls appeared as vertices in G consisted actually by super-vertices and its edge set is the number of edges appeared in G and hence between the super-vertices in G*. Thus, formally speaking, a *hyper-abstraction* of $G = (V, E)$ is a graph $G^* = (V', E', m)$, where:

- the set of vertices V' corresponds to system-call groups from C that appeared in G,
- the set of edges $E' \subseteq V \times V$ and $E' = E$ correpsonds to *dependencies* between system-calls,
- the mapping function $m : V' = V \to C$ associates vertices (system-calls) with the system-call group that they belong to

Next, in Figure 5.3 we use as an example the System-Call Dependency Graph presented in Figure 5.1. So, given a System-Call Dependency Graph and utilizing a pre-classified set of system-calls into groups we are able to construct a hyper-abstraction of the given graph. To start with, we first substitute each vertex with his system-call's corresponding group and then merge all the vertices that are of the same group (homonym). To this point it is extremely significant to underline and make clear that the produced graph lacks of one property that traditional System-Call Dependency Graphs have and it is that the G^* is *not acyclic*. As easily one can understand, by merging vertices it is very probable to create circles because, while the number of edges remains the same, their end-points are finally *concentrated* between less vertices. Indicateve example consists the creation of a self-loop in Figure 5.3.


ULONG
sc 13 :in(1) <- sc 15 :out(0)
sc 13 :in(2) <- sc 15 :out(0)
NTSTATUS
sc 13 :in(1) <- sc 15 :out(0)
NtRaiseHardError
sc 13 :in(4) <- sc 17 :out(0)
HARDERROR-RESPONSE-OPTION
sc 13 :in(3) <- sc 16 :out(0)
ACCESS-MASK
sc 0 :in(0) <- sc 1 :out(0)
PULONG-PTR
NtQueryAttributesFile
sc 3 :in(0) <- sc 2 :out(0)
sc 4 :in(0) <- sc 2 :out(0)
sc 5 :in(0) <- sc 2 :out(0)
sc 6 :in(0) <- sc 2 :out(0)
sc 7 :in(0) <- sc 2 :out(0)
sc 8 :in(0) <- sc 2 :out(0)
sc 9 :in(0) <- sc 2 :out(0)
sc 10 :in(0) <- sc 2 :out(0)
sc 11 :in(0) <- sc 2 :out(0)
sc 12 :in(0) <- sc 2 :out(0)
NtOpenSection
sc 0 :in(1) <- sc 2 :out(0)
POBJECT-ATTRIBUTES
ULONG
NTSTATUS
PROCESS
PROCESS
PROCESS
ACCESS MASK
FILE
MEMORY
OBJECT
ULONG
x2
NTSTATUS
PROCESS
x2
ACCESS MASK
FILE
x12
MEMORY
OBJECT

System-Call	GROUP
NtRaiseHardError	PROCESS
HARDERROR_RESPONSE_OPTION	PROCESS
PULONG_PTR	PROCESS
ULONG	ULONG
NTSTATUS	NTSTATUS
ACCESS_MASK	ACCESS_MASK
NtOpenSection	MEMORY
POBJECT_ATTRIBUTES	OBJECT
NtQueryAttributesFile	FILE

Figure 5.3: Hyper-Abstraction G^*

Group ID	Group Name	Group Tag	Group Cardinality
1	ACCESS_MASK	AM	1
2	Atom	AT	5
3	BOOLEAN	BO	1
4	Debug	DB	17
5	Device	DE	31
6	Environment	EN	12
7	File	FI	44
8	HANDLE	HD	1
9	Job	JB	9
10	LONG	LN	1
11	LPC	LP	47
12	Memory	MM	25
13	NTSTATUS	NT	1
14	Object	OB	19
15	Other	OT	36
16	PHANDLE	PH	1
17	PLARGE_INTEGER	PI	1
18	Process	PR	49
19	PULARGE_INTEGER	PS	1
20	PULONG	PU	1
21	PUNICODE_STRING	UI	1
22	PVOID_SIZEAFTER	VS	1
23	PWSTR	WS	1
24	Registry	RG	40
25	Security	SC	36
26	Synchronization	SN	38
27	Time	TM	5
28	Transaction	TN	49
29	ULONG	UL	1
30	WOW64	WW	19

Table 5.3: System Call Groups

In Table 5.3 we present the groups of system calls and the number of system-calls that each group includes. To this point, we ought to notice that the vertex set of G^* depends on the *type of distribution* followed by the vertices of the primary System-Call Dependency Graph when they are arranged in groups. What we mean is that, if in example the system-calls appeared in the vertices of System-Call Dependency Graph are *uniformly arranged* in the defined groups then we have a larger shrinkage on the size of the produced graph when we transit form G to G^*, while when a *Power-Law* or a *Gaussian* distribution is followed, then the size of the produced graph will exhibit a lower shrinkage.

5.2 Graph Similarity

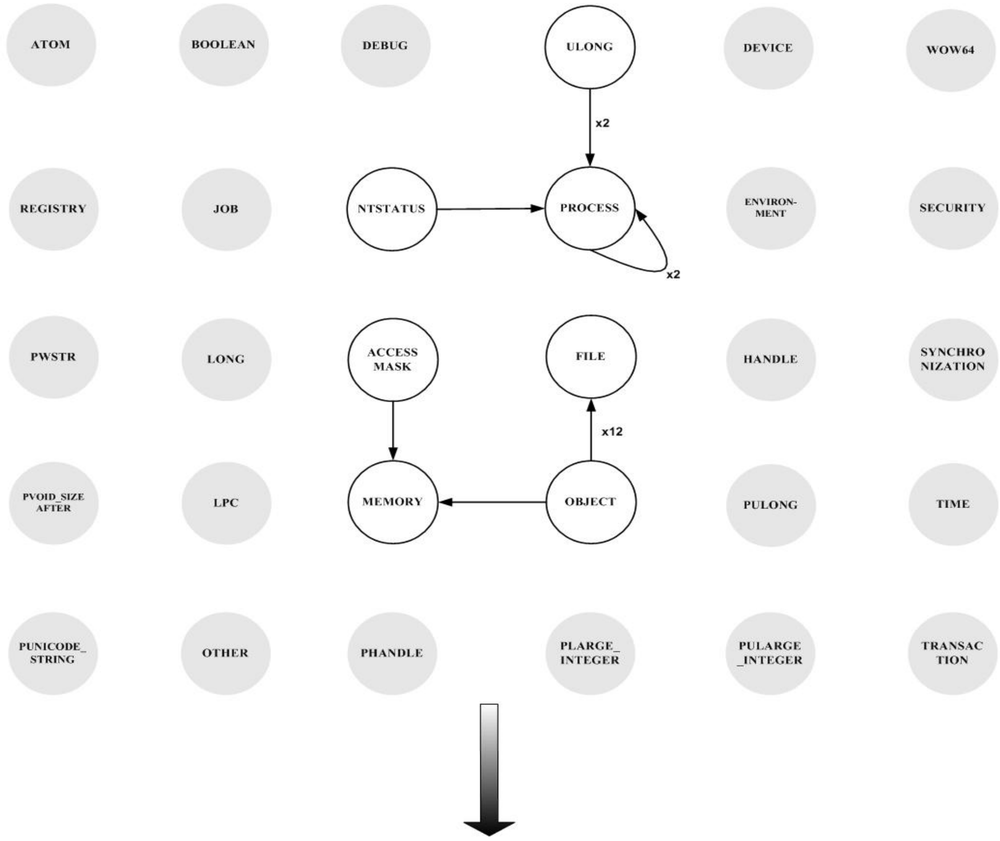
In this section we describe the representation of mawlare by its System-Call Depndency Graph, how malware is organized into malware families and make a brief introduction to knwon similarity metrics that we will utilize in our model.

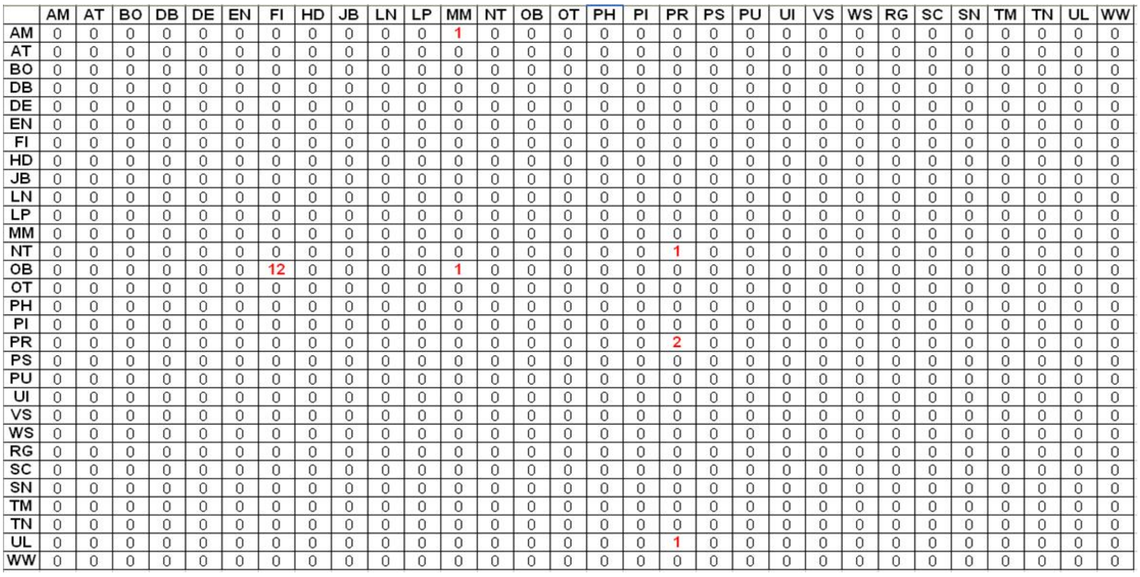
5.2.1 Graph Representation

As we referred above the auxiliary graph that consists hyper-abstraction of a given System-Call Dependency Graph (let us say G), the so called G^* has as its vertex set the groups of system calls that appeared in G. Additionally the number of all groups as depicted in Table 5.3 is 30. So, since the number of appeared vertices is fixed to at most 30 easily one can conclude that any given graph can be represented with a fixed size adjacency matrix with dimensions 30×30. thus next we cite an example of such a representation in Figure 5.4 keeping the paradigm of Figure 5.3.

As easily one can understand, since the graph is directed the resulting adjacency matrix is non symmetric. Additionally, we ought to notice that in the adjacency matrix are also included the isolated vertices, meaning the groups of system-calls that do not appear in the initial System-Call Dependency Graph. Thus, for a cell with coordinates x, y if it has a zero value it means that there is no edge from a system-call of group x to a system-call of group y, while if there is a non-zero value in that cell it respectively means that there are as many edges as the value in the cell from at least one system-call belonging to group x to at least one system-call belonging to group y.

To this point, we ought to repeat, in order to make clear, that the reason that we need any non-zero value and not only the ace, is the fact that this information is quite valuable so for the significance of the intercommunication of any two system-call group as for the utilization by metrics that take as input continuous values such as Bray-Curtis, Cosine Similarity and Tanimoto Coefficient that we use extensively in our formulas and we will discuss in the next subsection.





	AM	AT	BO	DB	DE	EN	FI	HD	JB	LN	LP	MM	NT	OB	OT	PH	PI	PR	PS	PU	UI	VS	WS	RG	SC	SN	TM	TN	UL	WW
AM	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
EN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HD	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
NT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
OB	0	0	0	0	0	0	12	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
OT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
PS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
UI	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
VS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
WS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
UL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
WW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.4: Adjacency Matrix from G^*

5.2.2 Malware Families and Sample Structure

Even though we have dedicated an individual chapter where we describe our experimental setup and our dataset's structure and indexing into malware families, in this section we ought to cite a brief description in order to be more easy for the reader to understand how our technique works. So, as depicted in Figure 5.5, while malware samples are

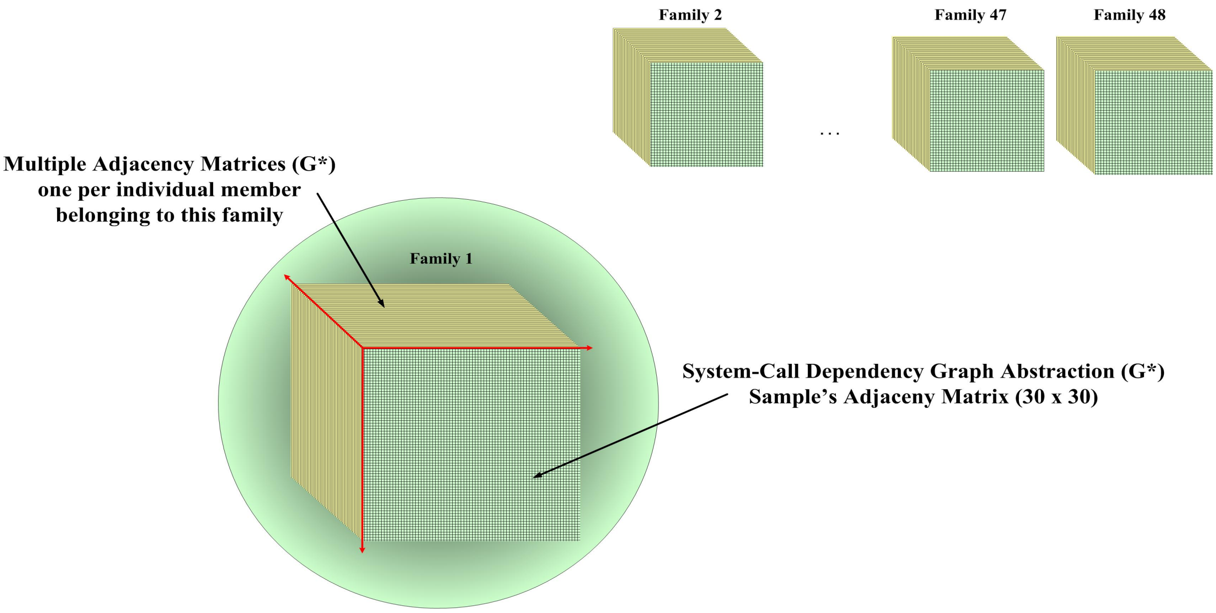

Family 2
Family 47
Family 48
...
Multiple Adjacency Matrices (G*)
one per individual member
belonging to this family
Family 1
System-Call Dependency Graph Abstraction (G*)
Sample's Adjaceny Matrix (30 x 30)

Figure 5.5: Organization of samples into malware families represented by G^* sets

organized into families and each sample is represented with an abstraction of its System-Call Dependecy Graph, the so called G^*, each mawlare family is consisted by a set of of G^*s with cardinality equal to her members.

5.2.3 Graph Similarity Metrics

Having already composed the theoretical background on about how we construct the hyper-abstraction graph G^* that we will utilize in our model, now we can proceed by discussing the similarity metrics that we will use in order to compute the similarity between any two graphs. In our approach we do not use directly one such metric in order to determine about the detection and the classification of an unknown sample. Instead, we combine either multiple metric in one formula or multiple formulas that already combine multiple metrics in order to provide results concerning the detection and classification, as we will show in later sections. However, to start with, in this section we make an brief introduction to the similarity metrics that we apply in order to compute the similarity between any two graphs and later we discuss how we leverage them by combining multiple similarity metrics in order to develop formulas that serve our purposes.

(A) Jaccard Index

Also known as Jaccard Similarity Coefficient. The Jaccard Index is used in order to compute the similarity between any two finite sample sets (vectors). However the main *drawback* of this similarity metric is the fact the it can be applied only on binary data (values 1 or 0) indicating respectively the existence or not of the i^{th} term in the two vectors. The result of the Jaccard similarity metric lies in the range $[0, 1]$. Jaccard Index can be computed as the size of the intersection divided by the size of the union of the sample sets. So for two given binary vectors (A, B) of length n both of them, the Jaccard Index is defined as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|(A_i = 1 \wedge B_i = 1)|}{|n|} \qquad \forall i \in [0, n-1] \tag{5.1}$$

So in order to compute the Jaccard index between any two adjacency matrices we sum every cell that has non-zero values to both matrices, as if a cell with coordinates x, y is non-zero into both adjacency matrices then the numerator is increased by one, while the denominator is the sum of all the cells that at least in one of adjacency matrices has non-zero value. However, to this point we ought to notice that if two corresponding cells have both zero values it is quite important as the inexistence of an edge may consist a qualitative characteristic of a family as we will discuss later.

(B) Bray-Curtis Dissimilarity

Bray-Curtis dissimilarity (Bray Curtis 1957) is a metric mostly used to derive relationships in ecology and environmental sciences.Bray-Curtis dissimilarity is defined as the sum of all differences of the values in each cell divided by the sum of all sums of the values in each cell. The result of the Bray-Curtis dissimilarity metric lies in the range $[0, 1]$. So, the Bray-Curtis dissimilarity for two bi-dimensional matrices A, B of size $(n \times n)$ is computed as follws:

$$BCD(A,B) = \frac{\sum\limits_{i=0}^{n-1}\sum\limits_{j=0}^{n-1} |A_{i,j} - B_{i,j}|}{\sum\limits_{i=0}^{n-1}\sum\limits_{j=0}^{n-1} |A_{i,j} + B_{i,j}|} \tag{5.2}$$

One main advantage that this metric provides, is the fact that it can be applied on continuous values, that is quite helpful for our approach since we can leverage the number of occurrences of each particular edge.

(C) Cosine Similarity

The Cosine similarity is mostly used for checking text document similarity where the vectors A and B are referred to term/words frequencies and each one is defined as the union of the words of the two texts. The Cosine Similarity measures the cosine of the angle between two vectors of an *inner product space.* Of course, like in almost all the similarity metrics its result lies in the range $[0, 1]$. The cosine of two vectors A, B (probably of different size) can be computed via the Euclidean dot product as $A \cdot B = \|A\|\|B\| cos\theta$ as:

$$similarity = cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum\limits_{i=1}^{n}(A_i \times B_i)}{\sqrt{\sum\limits_{i=1}^{n}(A_i)^2} \times \sqrt{\sum\limits_{i=1}^{n}(B_i)^2}} \tag{5.3}$$

Thus, easily one can understand that the cosine similarity can be applied by our model substituting the words and their corresponding frequencies with edges and the times of their appearance sthat are all stored in the adjacency matrix.

(D) Tanimoto Coefficient

The Tanimoto Coefficient is many time confused with Cosine Similarity since the both have similar algebraic forms. The Tanimoto coefficient is a mechanism for computing the Jaccard coefficient when the set under comparison are represented as bit vectors. Since the formula can be extended to be applied on vectors in general and since has similar properties with the Cosine similarity we utilize this metric to our model too. So, the Tanimoto coefficient fow two vectors A, B of length n can be computed as:

$$T(A,B) = \frac{A \cdot B}{\|A\|^2 + \|B\|^2 - (A \cdot B)} = \frac{\sum_{i=1}^{n}(A_i \times B_i)}{\sum_{i=1}^{n}(A_i)^2 + \sum_{i=1}^{n}(B_i)^2 - \sum_{i=1}^{n}(A_i \times B_i)} \tag{5.4}$$

5.3 Graph Based Malicious Software Detection

In this section we will describe our proposed technique for detecting unknown malware samples utilizing graph-based techniques that leverage System-Call Dependency Graphs. We present the development of a formula for calculating a value responsible for the detection of an unknown malware sample, that combines the information provided by known malware families according to qualitative characteristics resulting from its one, and similarity metrics such as the Jaccard index and the Bray-Curtis dissimilarity.

5.3.1 Detection Based on Family Qualitative Characteristics

As we referred across chapter 4, malware samples belonging to an individual malware family tend to share common characteristics. This is a quite valuable information, that we leveraged in order to develop a technique that will utilize these characteristics in order to result to if an unknown sample is malare or not.

To this point, we ought to refer a fact that intrigues our interesting and this is that the method we have developed for detection is family-based meaning that it utilizes information gathered across all the members of a family, while, instead, the method we have developed for classification is member-based since it is utilizing information gathered from a specific member in each family. However, through experiments, we observed that the family-based similarity metric we developed derived better results for detection while the member-based one derived better results for classification. As we referred above, members who belong to the same malware family tend to share common characteristics. So, based on this we developed the notion that these characteristics *should* be mirrored on the System-Call Dependency Graph and hence to its abstraction G^*.

Defining the term *characteristic* when working on G^*, we could claim that a characteristic is an edge between two system call classes, since in order for a specific task to be performed, system-calls of specific functionality need to be utilized and of course in different malware variants they can be substituted by equivalent ones. Thus we decided on focusing on edges that the most members in a family have them in their G^*s and hence they constitute a qualitative characteristic of a family. So, easily one can understand that if in a family of 100 members the 90 have a specific edge then this edge is of major importance, instead with another one that exists in only 10 members in the same family.

So, in order to append weights to each existed edge in every member of a family we should check how many times an individual edges appears across the members of a family. Thus, utilizing the adjacency matrixs that represent the G^* of each member, we created an auxiliary adjacency matrix for each family, where its each cell (edge) includes a value that represents its importance by depicting *in how many members' G^* this specific edge appears*, expressed in a percentage ratio.

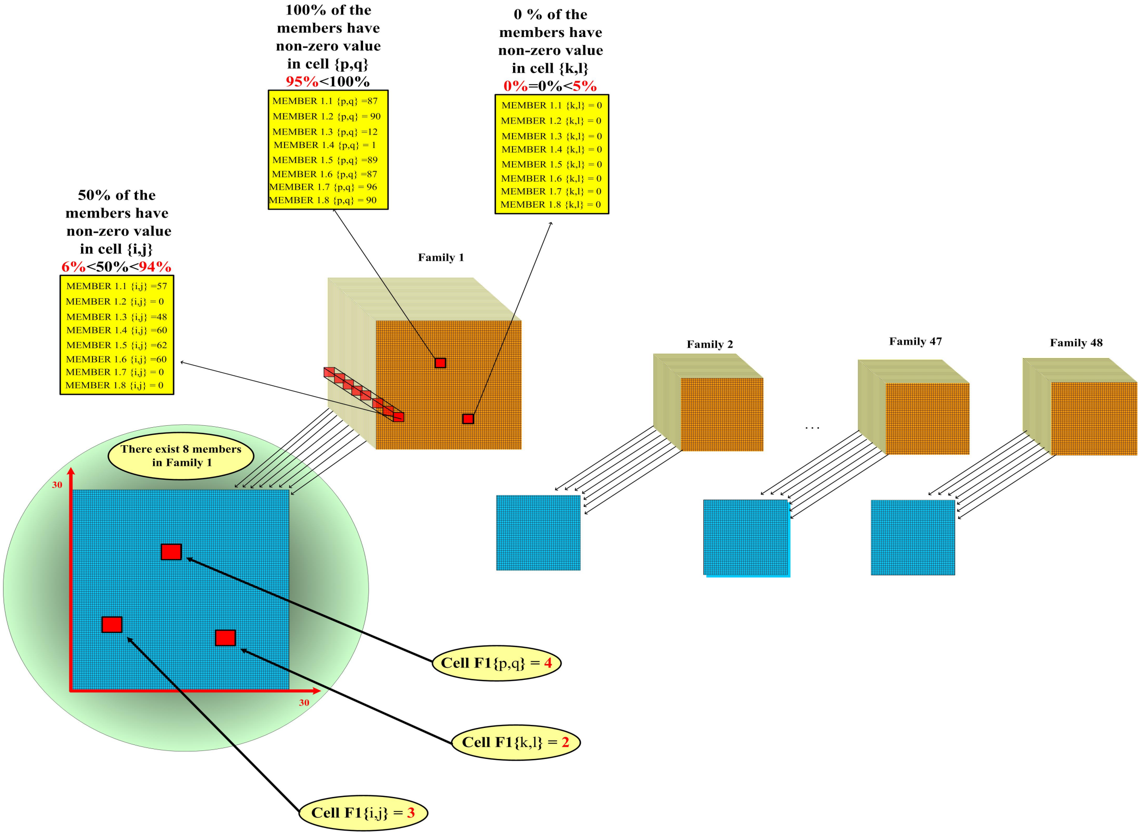


Figure 5.6: Zone Adjacency Matrix Construction

To this point we ought to underline that the values in each cell of each adjacency matrix vary in the range from 0 to the number of edges existed in the initial System-Call Dependency Graph, however we increment the counter only once if and only if there does exist non-zero value in the cell. So, having collected this valuable information we can proceed by filtering it as to decide the important edges that finally will consist the characteristics of each family.

Having compute the percentage of appearance of each edge in the auxiliary adjacency matrix we can proceed by assigning weights to each cell (edge) on this matrix. In order to assign weights we divide the values (ranging from 0 to 100) to three zones. However, before we assign the *importance tags* we ought to define the zone ranges. So, we first define a threshold about 95% and the zones are arranged based on this threshold. Thus, we mark each cell either with *important zone tag* that covers cells that include values in the range [0.95 - 1], or with *gray zone tag* that covers cells that include values in the range (0.05 - 0.95), or with *zone of inexistence tag* that covers cells that include values in the range [0- 0.05]. In Figure 5.6 we cite a simple example of how we assign the importance tags in the auxiliary family-level *zone* adjacency matrix.

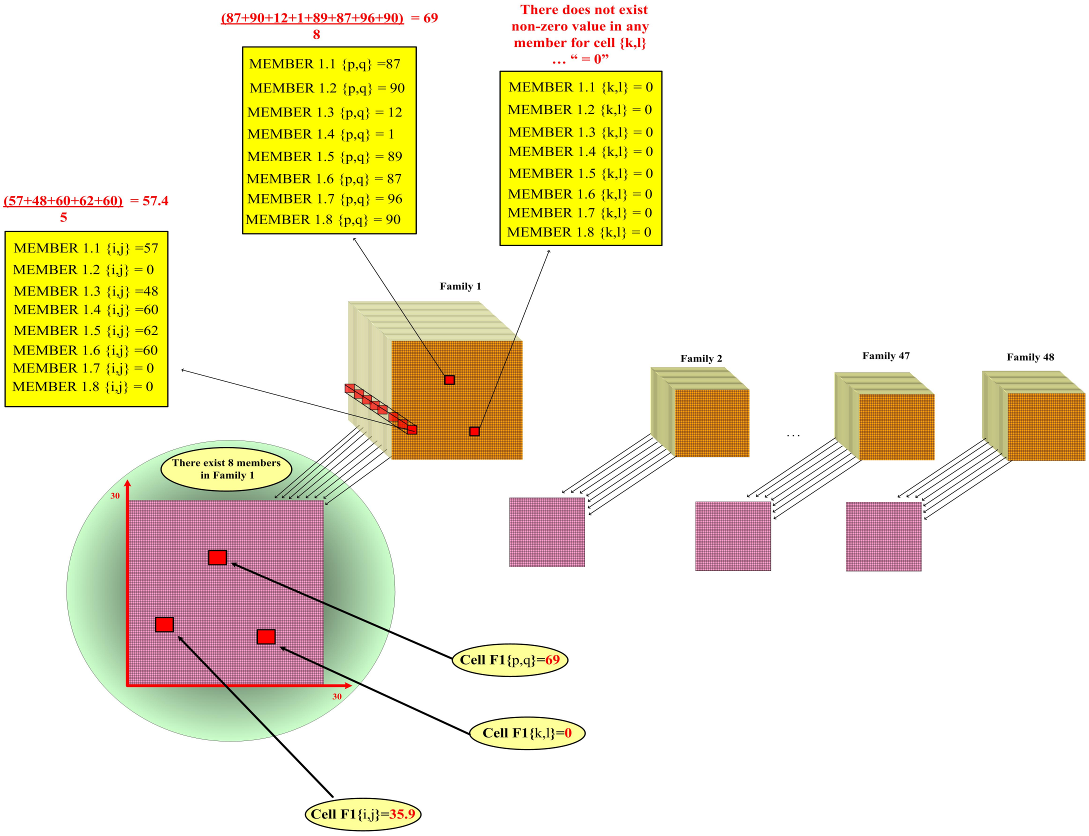


Figure 5.7: Accumulative Adjacency Matrix Construction

Once we have created our first auxiliary adjacency matrices that include the zones of importance for each family based on their members we can proceed by creating one more kind of auxiliary matrices that are accumulative concerning the values in each cell, meaning the mean value of the existed edges in each member's G^*. So, the main target is to find the non-zero values in each cell of each member's G^*, and then find the mean of them. To this point we ought to notice that we do not divide by the total number of cells (number of members in this family) but instead, we divide by the number of non-zero cell found during the search. A brief depiction of the whole process for the construction of the accumulative adjacency matrix is depicted in Figure 5.7.

5.3.2 Malware Detection Formula Components

Having already constructed our auxiliary adjacency matrices, we can then proceed by defining the component that compose our formula for malware detection. Next we enumerate them and provide a description based on what we have described until now.

(A) Family-side Fitting ($4 \rightarrow 1$ matching)

The Family-side fitting is a similarity metric that we have developed and focuses on compute the similarity between the test sample and any malware family based on the family's *zone adjacency matrix.* The main purpose of this metric is to compute the rate of satisfiability on the qualitative characteristics of any family by a test sample. Next we enumerate the steps followed for the computation of *Family-side Fitting* similarity.

1. In order to compare the test sample's adjacency matrix with the zone adjacency matrix of one family, we first need to make a cast on test sample's adjacency matrix. As we referred above the test sample's adjacency matrix includes cell that either have zero values or non-zero ones. So, we cast any non-zero values existed in test sample's adjacency matrix into aces.

2. The next step, that is the main process of this similarity metric is to *cover* in some fashion the family's zone adjacency matrix with the test sample's adjacency matrix and count how many aces ($TS[i,j] = 1$) of test sample's adjacency matrix fit on cells of family's zone adjacency matrix that have an important zone tag ($FM[i,j] = 4$).

3. Next, we count the total number of cells in family's zone adjacency matrix that have an important zone tag.

4. Finally, all we do is to divide the number of cells with non-zero value in test sample's adjacency matrix that fit on cells with important zone tag in family's zone adjacency matrix by the total number of cells with important zone tag in family's zone adjacency matrix. Thus, the formula for computing the *Family-side Fitting* similarity metric is as follows:

$$Sim_{4\rightarrow 1}(FM, TS) = \frac{|FM \cap_{4\rightarrow 1} TS|}{|FM[i,j] = 4|} \ , \qquad \forall \ \ 0 \leq i,j < n \ , \tag{5.5}$$

   where n is the size of familiy's zone adjacency matrix and test sample's casted adjacency matrix, FM is the family's zone adjacency matrix, TS is the test sample's casted adjacency matrix and $|FM \cap_{4\rightarrow 1} TS| = |FM[i,j] = 4 \wedge TS[i,j] = 1|_{i,j=0}^{i,j \leq n}$

(B) Sample-side Fitting ($1 \rightarrow 4$ matching)

The Sample-side fitting is also a similarity metric that we have developed and while it seems quite the same with the Family-side fitting similarity it differentiates in the fact that it focuses on computing the similarity between the test sample and any malware family based on the test sample's *adjacency matrix*. The main purpose of this metric is to compute the rate of satisfiability in terms of edge existence of a test sample's adjacency matrix by the qualitative characteristics of any family represented by her zone adjacency matrix. Next we enumerate the steps followed for the computation of *Sample-side Fitting* similarity. The steps are almost the same as in the ones followed in the computation of Family-side fitting similarity.

1. In order to compare the test sample's adjacency matrix with the zone adjacency matrix of one family, we first need to make a cast on test sample's adjacency matrix. So, we cast any non-zero values existed in test sample's adjacency matrix into aces.

2. The next step, that is the main process of this similarity metric is to *cover* in some fashion the test sample's adjacency matrix with the family's zone adjacency matrix and count how many cells of family's zone adjacency matrix that have an important zone tags ($FM[i,j] = 4$) fit on cells having aces ($TS[i,j] = 1$) on test sample's adjacency matrix.

3. Next, we count the total number of cells in test sample's adjacency matrix that have non-zero value ($TS[i,j] = 1$) .

4. Finally, all we do is to divide the number of cells with important zone tag in family's zone adjacency matrix that fit on cells with non-zero value in test sample's adjacency matrix by the total number of cells with non-zero value in test sample's adjacency matrix. Thus, the formula for computing the *Sample-side Fitting* similarity metric is as follows:

$$Sim_{1\rightarrow 4}(TS, FM) = \frac{|TS \cap_{1\rightarrow 4} FM|}{|TS[i,j] = 1|}\ , \qquad \forall\ \ 0 \leq i,j < n\ , \tag{5.6}$$

where n is the size of familiy's zone adjacency matrix and test sample's casted adjacency matrix, FM is the family's zone adjacency matrix, TS is the test sample's casted adjacency matrix and $|TS \cap_{1\rightarrow 4} FM| = |TS[i,j] = 1 \wedge FM[i,j] = 4|_{i,j=0}^{i,j\leq n}$

(C) Mean and Max Jaccard Similarity

One more component we utilize to empower our formula for malware detection is the Jaccard index. The reason that we choose to utilize the Jaccard similarity is the fact that since this similarity metric is mostly applied on binary vector and since an ace or a zero can indicate the existence or nonexistence respectively of an edge it seemed to work fine for the *qualitative comparison* between two objects in terms of edge existence, while the *quantitative* one can be measure using the Bray-Curtis similarity as we will discuss next.

So, we utilize two times the Jaccard index as to compute firstly the mean similarity between the sample and all the members of a malware family and then we keep only the maximum value produced by the most similar member of the family to the sample. Next we enumerate the steps followed for the computation of Jaccard similarity

1. Before we start with the computation of similarity, we ought to notice that since working with the Jaccard similarity we need to cast both the adjacency matrices the one of each member and the one of the test sample to having as values zeros or aces. So, we first cast each value greater than zero to ace, and then we leave as it has, each value that equals to zero.

2. To this point we can point that the computation of the mean and the max Jaccard similarity can be performed synchronously. So, we first compute the Jaccard similarity between the test sample and each member of a family by counting the number of cells that in both adjacency matrices have non-zero value (both having aces) and then dividing by the number of cells that have ace in at least one of the adjacency matrices (either the one o family member's or the one of test sample's) have aces.

   The formula for computing the *Jaccard* similarity between a test sample and a family member can be computed as follows:

$$J(TS, M) = \frac{|TS \cap_{1\to1} M|}{|TS \cap_{1\to1} M| + |TS \cap_{1\to0} M| + |TS \cap_{0\to1} M|} \ , \tag{5.7}$$

   where M is the member's casted adjacency matrix, TS is the test sample's casted adjacency matrix and

$$\begin{aligned} |TS \cap_{1\to1} M| &= |TS[i,j] = 1 \wedge M[i,j] = 1|, \\ |TS \cap_{1\to0} M| &= |TS[i,j] = 1 \wedge M[i,j] = 0|, \\ |TS \cap_{0\to1} M| &= |TS[i,j] = 0 \wedge M[i,j] = 1|, \quad \forall \ 0 \leq i,j < n. \end{aligned}$$

3. Then, we check the current value of the Jaccard similarity between the test sample and the current member of a malware family and we store it if it is the maximum computed until now for this family as:

$$J_{max}(TS, FM) = max[J(TS, M_i)]_{i=1}^{n} \ , \tag{5.8}$$

where n is the cardinality of the set (members in this family), M_i is the i^{th} member's casted adjacency matrix, FM is the malware family that the member belongs to, and TS is the test sample's casted adjacency matrix.

4. Finally, having computed the Jaccard similarity between the test sample and each member of a mawlre family, we sum the values and divide them by the number of members in this family. Thus, the formula for computing the *Mean Jaccard* similarity is as follows:

$$J_{mean}(TS, FM) = \frac{\sum\limits_{i=0}^{n} J(TS, M_i)}{n} \ , \tag{5.9}$$

   where FM is the family set, n is the cardinality of the set (members in this family), M_i is the i^{th} member's casted adjacency matrix and TS is the test sample's casted adjacency matrix.

(D) Mean and Max Bray-Curtis (Dis)Similarity

The last component we utilize to empower our formula for malware detection is the Bray-Curtis dissimilarity. The reason that we choose to utilize the Bray-Curtis dissimilarity is the opposite of that of why we used Jaccard index as Bray-Curtis dissimilarity metric is mostly applied on continuous data and since any non-zero value in the adjacency matrices indicate the existence and the cardinality of an edge it seemed to work fine for the *quantitative characteristic comparison* between two objects in terms of edge existence or nonexistence and in the case of edge existence of edge cardinality respectively.

So, as in the case of Jaccard index, we utilize two times the Bray-Curtis dissimilarity as to compute firstly the mean similarity (1-dissimilarity) between the sample and all the members of a malware family and then we keep only the maximum value produced by the most similar (least dissimilar) member of the family to the sample. Next we enumerate the steps followed for the computation of Bray-Curtis Dissimilarity. To this point, it is notable to refer that for the computation of the mean Bray-Curtis dissimilarity we can both utilize either each member's initial adjacency matrix or the family-level accumulative adjacency matrix we describe in the previous section. In the next step we describe both the computation of Bray-Curtis using either each member's initial adjacency matrix or the family-level accumulative adjacency matrix. Before we start we can point that the computation of the mean and the max Jaccard similarity can be performed synchronously.

1. So, we first compute the Bray-Curtis dissimilarity between the test sample and each member of a family by summing the differences between any pair of respective cells in both test sample's and member's initial adjacency matrix. Then we sum the sums of the values in any pair of respective cells in both test sample's and member's initial adjacency matrix. So,the formula for computing the *Jaccard* similarity between a test sample and a family member can be computed as follows:

$$BCD(TS, M) = 1 - \frac{\sum\limits_{i=0}^{n-1}\sum\limits_{j=0}^{n-1}(TS[i,j] - M[i,j])}{\sum\limits_{i=0}^{n-1}\sum\limits_{j=0}^{n-1}(TS[i,j] + M[i,j])}\ , \tag{5.10}$$

   where n is the size of the adjacency matrix, M is the member's initial adjacency matrix and TS is the test sample's initial adjacency matrix

2. Then, we check the current value of the Bray-Curtis dissimilarity between the test sample and the current member of a malware family and we store it if it is the maximum computed untill now for this family as:

$$BCD_{max}(TS, FM) = max[BCD(TS, M_i)]_{i=1}^{n}\ , \tag{5.11}$$

   where n is the cardinality of the set (members in this family), M_i is the i^{th} member's initial adjacency matrix, FM is the malware family that the member belongs to, and TS is the test sample's initial adjacency matrix.

3. Finally, having computed the Bray-Curtis dissimilarity between the test sample and each member of a malware family, we sum the values and divide them by the number of members in this family. Thus, the formula for computing the *Mean Bray-Curtis* dissimilarity is as follows:

$$BCD_{mean}(TS, FM) = \frac{\sum\limits_{i=0}^{n} BCD(TS, M_i)}{n}\ , \tag{5.12}$$

   where FM is the family set, n is the cardinality of the set (members in this family), M_i is the i^{th} member's initial adjacency matrix and TS is the test sample's initial adjacency matrix.

Alternatively, we can compute the *Mean Bray-Curtis* dissimilarity by computing the Bray-Curtis dissimilarity by modifying the equation 5.10 as to compare the test sample's initial adjacency matrix with the family's accumulative adjacency matrix as:

$$BCD(TS, FM) = 1 - \frac{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(TS[i,j] - FM[i,j])}{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(TS[i,j] + FM[i,j])}, \tag{5.13}$$

where n is the cardinality of the set (members in this family), FM is the family's accumulative adjacency matrix and TS is the test sample's initial adjacency matrix

5.3.3 Malware Detection using NP-Similarity

The *NP*-similarity metric is a formula that we have developed in order to be able to detect if an unknown sample is a malicious or a benign one. The comparison is performed between an unknown sample and a malware family. So given an unknown sample and a series of malware families we can decide according to the results of NP-similarity metric if the given sample is malware or not. Actually we compute the NP-similarity between the sample and each family and keep the maximum value computed from the sample and each family. Then if the maximum value is below a pre-specified threshold then the unknown sample is benign or malware otherwise.

To start with, we ought to remind that the formula is actually a complex type of all the aforementioned similarity metrics of the previous section. However, we decided to assign different weights on to each one as they provide information about the similarity on different areas such as the satisfiability of family or sample qualitative or quantitative characteristics. So next, we cite the steps for the composition of the formula for the computation of NP-similarity and finally we present the computation of the formula, consisted by three components let us say F_1, F_2 and F_3. To this point we must declare that we have define four factors: $a = 4, b = 2, F_A = 1.5$ and $F_B = 1.2$.

1. To start with, we compute the first component of the formula that describes the qualitative satisfiability between the sample and a malware family by computing the *Family-side Fitting* similarity and the *Sample-side Fitting* similarity, between the test sample and the current family as they presented in equations 5.5 amd 5.6 respectively. Then we apply the factors as follows:

$$F_1 = \begin{cases} ((a \times S_{4\to1}) + (b \times S_{1\to4})) \times F_A & \text{, iff } S_{4\to1} = S_{1\to4} = 1 \\ \\ ((a \times S_{4\to1}) + (b \times S_{1\to4})) \times F_B & \text{, otherwise} \end{cases}$$

where $S_{4\to1} = Sim_{4\to1}(FM, TS)$ and $S_{1\to4} = Sim_{1\to4}(TS, FM)$

2. Then we proceed by computing the second component of the formula that describes the existential satisfiability between the sample and a malware family as described by the Jaccard index. So, we compute and then assign the corresponding weights on the max Jaccard and mean Jaccard similarities between the test sample and the current malware family as presented in equations 5.8 and 5.9 respectively, as follows.

$$F_2 = (b \times \overline{J}\,) + (a \times J_{max}) \ , \tag{5.14}$$

where $\overline{J} = J_{mean}(TS, FM)$ and $J_{max} = J_{max}(TS, FM)$.

3. Then we proceed by computing the final component of the formula that describes the qualitative satisfiability between the sample and a malware family as described by the Bray-Curtis dissimilarity. So, we compute and then assign the corresponding weights on the max Bray-Curtis and mean Bray-Curtis (dis)similarities between the test sample and the current malware family as presented in equations 5.1 and 5.12 - 5.13 respectively, as follows.

$$F_3 = (b \times \overline{BCD}) + (a \times BCD_{max}) \ , \tag{5.15}$$

where $\overline{BCD} = BCD_{mean}(TS, FM)$ and $BCD_{max} = BCD_{max}(TS, FM)$.

4. Finally we combine the three pre-computed components F_1, F_2 and F_3 in order to compose the final type of the formula by the product of the equations 5.14, 5.15 and 5.16 as:

$$NP = F_1 \times F_2 \times F_3 \tag{5.16}$$

After computations clears that the NP_{sim} is maximized when all the included similarities result in ace, while its max value is 324 and so a further normalization can be performed when dividing by the maximum value as follows:

$$NP = \frac{F_1 \times F_2 \times F_3}{324} \tag{5.17}$$

So, as we will present in the corresponding chapter of experimental results this technique can result with extremely high detection rate while exhibiting low false positive rates. To this point it is worth notable to refer that, as we will discuss later, it is proven through experiments that NP-similarity is in position to perform a crystal clear distinction between malware and benign programs using a quite low threshold, since of course is leveraging existential, qualitative and quantitative characteristics expressed through known malwares' System-Call Dependency Graphs.

5.4 Graph Based Malicious Software Classification

In this chapter we will present a series of approaches we have utilized in order to classify an unknown test sample to exclusively one of a series of given malware families. We firstly present how each technique works and then we discuss how we compose them, and how we actually use them as a series of filters, in order to achieve the optimal possible classification of an unknown sample. To this point we ought to refer that just like in our proposal for malware detection we are based again on combinatorial approach we followed in NP-similarity. However, having already tried the NP-similarity in malware classification experiments we observed that it was not so effective as expected, and hence we should proceed by a more straightforward approach such as performing direct similarity computation on the initial and casted adjacency matrices using only the traditional similarity metrics.

5.4.1 Malware Classification Filters

In order to perform malware classification, we choose to omit the family's qualitative characteristics and proceed by compare only the test sample with each member inside each family. Our main target is to keep the highest similarity result exhibited by a member of a family as representative of this family and then to classify the test sample to the family in which belongs the representative that is most similar to the sample. In other words, given f malware families we measure the similarity between the test sample and each member in each family and keep the top similar result for each family resulting to f similarity results, one for each family. Then the approach is straightforward as we classify the test sample to the family that served the top value among f. However, in case of tie we proceed to next series of corresponding result produce by other similarity metrics (filters). So as easily one can understand, since the filter have different classification ability, the final classification results depend on the filter sequence. Next we present the components of our classification method.

(A) SaMe: Sample - Member Optimal Fitting

The most straightforward approach for classifying an unknown sample is to measure its similarity with all the known sample (members in each family) and then to classify it to the family that belongs the most similar known sample (member of this family). This approach is based on the intuition that it is probable enough that the unknown sample is directly correlated through phylogeny with its most similar known sample either as a descendant or as ancestor.

So, in order to capture so the quantitative as the existential characteristics we propose a combination of Jaccard index, Bray-Curtis Dissimilarity and Cosine similarity and Tanimoto coefficient, that we call *SaMe* (stands for SAmple-MEmber) similarity. Next, we cite the steps followed for the construction of the combination of metrics, just like in NP-similarity and finally how we compute the similarity between an unknown sample and each member of each family using the SaMe similarity. The steps presented below are followed in order to finally compute the SaMe similarity for each member of a family.

1. First, in order to measure the similarity in edge existence level we compute the Jaccard index between the test sample and the current member of a family $J(TS, M)$, utilizing their casted adjacency matrices as we already presented in equation 5.7

2. Next, in order to capture the qualitative characteristics of each member and to a greater extent of its corresponding family we proceed by computing the Bray-Curtis dissimilarity between the test sample and the current member of a family $BCD(TS, M)$, utilizing their initial adjacency matrices as we already as presented in equation 5.10

3. Emphasizing on capturing the qualitative characteristics we similarly compute the Cosine similarity between the test sample and the current member of a family utilizing their initial adjacency matrices as presented below:

$$CS(TS, M) = \frac{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(TS[i,j] \times M[i,j])}{\sqrt{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} TS[i,j]^2} \times \sqrt{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} M[i,j]^2}} \tag{5.18}$$

4. Insisting on capturing the qualitative characteristics, we additionally compute the Tanimoto coefficient between the test sample and the current member of a family utilizing their initial adjacency matrices as presented below:

$$T(TS,M) = \frac{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(TS[i,j] \times M[i,j])}{\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}TS[i,j]^2 + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}M[i,j]^2 - \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(TS[i,j] \times M[i,j])} \tag{5.19}$$

5. Having computed all four components $J(TS, M), BCD(TS, M), CS(TS, M)$ and $T(TS, M)$ in order to compose the final type of the formula by the product of the aforementioned similarity metrics as:

$$SaMe(TS, M) = J(TS, M) \times BCD(TS, M) \times CS(TS, M) \times T(TS, M) \tag{5.20}$$

6. Having completed the computation of the $SaMe$ similarity metric between the test sample and a member of a family we repeat the whole process for all the members of this family keeping the maximum value that appears in this family as.

$$SaMe_{max}(TS, FM) = max[SaMe(TS, M_i)]_{i=1}^{n} \ , \tag{5.21}$$

where n is the cardinality of the set (members in this family), M_i is the i^{th} member's initial adjacency matrix, FM is the malware family that the member belongs to, and TS is the test sample's initial adjacency matrix.

7. Then we repeat again for all the members of all the other families as before and finally keep as the dominant family the one that includes the member that exhibits the maximum value of similarity with the test sample according to $SaMe$ similarity. The typical depiction of the above is the following:

$$F_{dominant} = max[SaMe_{max}(TS, FM_f)]_{f=1}^{N} \ , \tag{5.22}$$

where N is the number of all families.

(B) Conscripting NP-Similarty for Malware Classification

As we referred in the introduction of the section, the *NP* similarity by itself is not as effective as expected and hence it can not be utilized by itself for malware classification. However, we observed experimentally that if combined with *SaMe* similarity it can yield even more higher classification rates than *SaMe* by itself. So, since *NP* similarity enforces *SaMe* we decided to combine them in order to construct one more classification filter.

The most interesting point in the combination is the way it is done. Actually we measure the similarity between the test sample and a member of a malware family using *SaMe* similarity and then we *patch* in some fashion the difference from the perfect matching using *NP* similarity. So, in other words, we use for computig the similarity of the two objects while conscripting *NP* to recompute it but in the percentage their dissimilarity as described by the next steps:

1. Firstly we compute the NP similarity between the test sample and an individual mawlare family as described by the equation 5.16.
2. Next, we proceed by computing the $SaMe$ similarity between the test sample and the selected malware family as described by the equation 5.21.
3. Having already computed the NP and the $SaMe$ similarity between the test sample and a malware family, we can proceed by computing their combination as described by the next equation

$$S^{NP}_{SaMe}(TS,FM) = [SaMe_{max}(TS,FM)] + [SaMe^{'}_{max}(TS,FM) \times NP(TS,FM)] \ , \qquad (5.23)$$

   where $SaMe^{'}_{max}(TS,FM) = (1 - SaMe_{max}(TS,FM))$.

4. Finaly as before, we can keep as the *dominant* family the one that exhibits the maximum values in the equation 5.23 as:

$$F_{dominant} = max[S^{NP}_{SaMe}(TS,FM_f)]^{N}_{f=1} \ , \qquad (5.24)$$

   where N is the number of all families.

(C) Retrieving Malware's Kernel Computing The MSCC of $\mathbf{G}^*$

In this part we propose a quite alternative approach, where we leverage pure graph-theoretic background in order to develop an elaborate technique for malware classification. To be more precise, we take into account the Maximum Strongly Connected Component (MSCC) of a given graph and also all the Strongly Connected Components that appear in it and leverage them in order to capture characteristics that consecutively will be utilized in malware's classification.

Next we present the tow approaches that we follow starting from inside out, presenting firstly the *Kernel* similarity measuring the percentage in the mapping of vertices in each strongly connected component of a known samples graph on the test samples maximum strongly connected component and then the *Cover* similarity where we measure the percentage of common vertices that are linked to the common vertices in the maximum strongly connected components of a known and an unknown sample.

Before we start, we ought to refer that our approach of treating malware's System Call Dependency Graph as an object that has cover and kernel is definitely not random, as our intuition is based on the real-life biological cells that also have a kernel surrounded by a semi-permeable cellular membrane that works as a cover and is responsible for its intercommunication with its environment. Thus, making the parallelism, the kernel is consisted by the maximum strongly connected component and the cover is consisted by the rest of the vertices that, while they do not belong to the maximum strongly connected component, are linked with vertices in it either with incoming or outgoing edges.

(1) Kernel Similarity

Basically our approach for the kernel similarity is somehow test based since we focus of the mapping of vertices belonging to strongly connected components of a member's G^* on vertices belonging to the maximum strongly connected component of test sample's G^* as depicted in Figure 5.8.

Next we enumerate the steps for the computation of Kernel similarity:

1. Firstly we compute the Maximum Strongly Connected Component of test samples G^* using the Tarjan's SCC algorithm.

2. Then we compute all the Strongly Connected Components on the G^* of a family's member with whom we want to measure the similarity, using again the Tarjan's SCC algorithm.

3. Having computed all the Strongly Connected Components in the member's G^* we proceed by assigning a label on to each vertex in a way that vertices that belong to the same Strongly Connected Component to have the same label.

4. Next we return to the Maximum Strongly Connected Component of test sample's G^* and we match the vertices inside it with the vertices in every Strongly Connected Component of member's G^* while we count the occurrences of each group of vertices from the members' G^* inside test's Maximum Strongly Connected Component. In

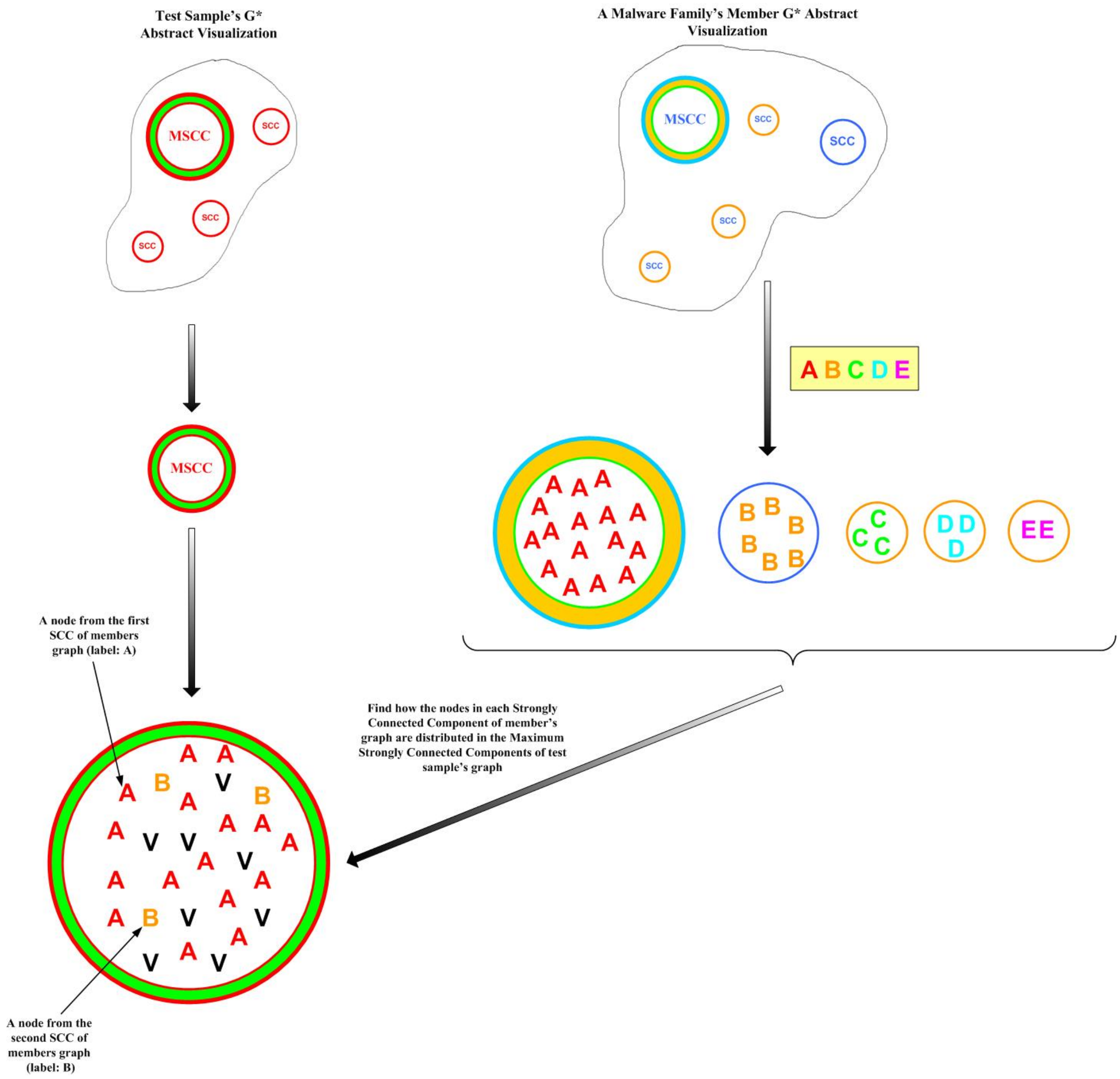

Test Sample's G* Abstract Visualization
A Malware Family's Member G* Abstract Visualization
MSCC
SCC
SCC
SCC
MSCC
SCC
SCC
SCC
SCC
A B C D E
MSCC
A node from the first SCC of members graph (label: A)
Find how the nodes in each Strongly Connected Component of member's graph are distributed in the Maximum Strongly Connected Components of test sample's graph
A node from the second SCC of members graph (label: B)

Figure 5.8: Kernel Similarity Visualization

other words, if the vertex v_i belongs to the Strongly Connected Component with label l_α in the member's G^*, and also exists inside the test's Maximum Strongly Connected Component, then the occurrences of group α are increased by one.

5. Finally we compute the $Kernel$ similarity as follows:

$$K(TS, M) = \frac{\sum_{g=1}^{k} \frac{|v : v \rightarrow \ell(g)|}{|MSCC(TS)| + |SCC_g(M)| - |v : v \rightarrow \ell(g)|}}{k}, \tag{5.25}$$

where k is the total number of Strongly Connected Components in a malware family member's G^*, $\ell(g)$ is the *label* of the g^{st} Strongly Connected Component of member's

G^*, $|v : v \rightarrow \ell(g)| = |v : v \in MSCC(TS) \wedge v \in SCC_g(M)|$ refers to the number of vertices that exist in test's G^* Maximum Strongly Connected Component and also exist in the g^{st} Strongly Connected Component of member's G^* and hence have the same label with this component's vertices, $|MSCC(TS)|$ is the total number of vertices in test's G^* Maximum Strongly Connected Component and $|SCC_g(M)|$ is the number of vertices in the g^{st} Strongly Connected Component of member's G^*.

6. So, in order to compute the Kernel similarity between the unknown test sample and a mawlare family we compute as presented in previous methods the maximum value that appears among all the members of the family as follows:

$$K_{max}(TS, FM) = max[K(TS, M_i)]_{i=1}^{n} \ , \tag{5.26}$$

   where n is the number of the members in this malware family.

7. And hence, we can keep as the *dominant* family the one that exhibits the maximum values in the equation 5.26 as:

$$F_{dominant} = max[K_{max}(TS, FM_f)]_{f=1}^{N} \ , \tag{5.27}$$

   where N is the number of all families.

(2) Cover Similarity

Our approach for the *cover* similarity is somehow inspired from the Jaccard index since its computation is based on the intersection of the two vertex sets in each Maximum Strongly Connected Components. Actually, Our main target is to compute the similarity between the vertices in the test's G^* and the member's G^* that while they do not belong in their corresponding Maximum Strongly Connected Component they have incoming or outgoing edges with vertices that exist in the Maximum Strongly Connected Component of test's G^* and the Maximum Strongly Connected Component of member's G^*. Envisaging these sets as covers we proceed by measuring the similarity between them based on their vertex sets as depicted in Figure 5.9, where the vertices that have in/out edge with the vertices in the two Maximum Strongly Connected Components are the green for the test's G^* and the red ones for the member's G^*.

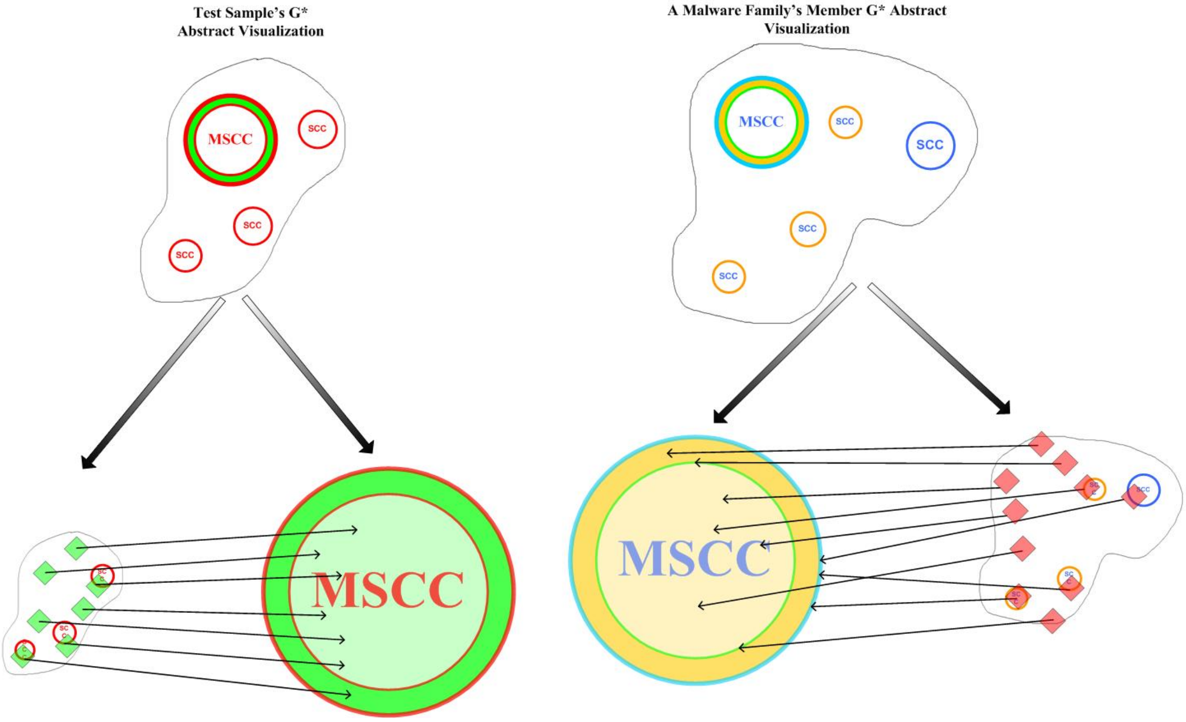


Figure 5.9: Cover Similarity Visualization

Next we enumerate the steps for the computation of Cover similarity:

1. Firstly, we compute the Maximum Strongly Connected Component of test samples G^* using the Tarjan's SCC algorithm.

2. Next, we compute the Maximum Strongly Connected Component of member's G^* using once more the Tarjan's SCC algorithm.

3. Having computed both the Maximum Strongly Connected Components in test's and member's G^*s, we proceed by storing separately the vertices in the test's G^* and the member's G^* that, while as we said before they do not belong to their corresponding Maximum Strongly Connected Component, they have incoming or outgoing edges with the vertices in test's Maximum Strongly Connected Component and the member's Maximum Strongly Connected Component (see green/red marks in Figure 5.9).

4. Having completed all the preparatory computation we can finally proceed with the computation of the *Cover* similarity as follows:

$$C(TS, M) = \frac{|\langle MSCC_{TS}\rangle_{cv} \cap \langle MSCC_M\rangle_{cv}|}{|\langle MSCC_{TS}\rangle_{cv}| + |\langle MSCC_M\rangle_{cv}| - |\langle MSCC_{TS}\rangle_{cv} \cap \langle MSCC_M\rangle_{cv}|}\ , \quad (5.28)$$

where the symbol $\langle\ \rangle_{cv}$ means the *cover* in terms of a vertex set that do not belong to any of the two Maximum Strongly Connected Components while it has in/out edges with vertices that exist in the Maximum Strongly Connected Components, or formally: $|v : (v \notin \{MSCC_{TS} \cup MSCC_M\}) \wedge (v \rightleftharpoons \{MSCC_{TS} \cup MSCC_M\})|$, where $|\langle MSCC_{TS}\rangle_{cv}|$ and $|\langle MSCC_M\rangle_{cv}|$ are the numbers of vertices that do not belong to their corresponding Maximum Strongly Connected Components but they have edges with vertices that exist in them and $MSCC_{TS}, MSCC_M$ are the Maximum Strongly Connected Components of test and member respectively.

5. So, in order to compute the Cover similarity between the unknown test sample and a mawlare family we compute as presented in previous methods the maximum value that appears among all the members of the family as follows:

$$C_{max}(TS, FM) = max[C(TS, M_i)]_{i=1}^{n} \ , \tag{5.29}$$

where n is the number of the members in this malware family.

6. Hence, we can keep as the *dominant* family the one that exhibits the maximum values in the equation 5.29 as:

$$F_{dominant} = max[C_{max}(TS, FM_f)]_{f=1}^{N} \ , \tag{5.30}$$

where N is the number of all families.

5.4.2 Malware Classification using Mutliple Filters

Having already discussed our proposed methods for classifying an unknown malware sample into a malware family, we can now proceed by presenting how we can combine all the aforementioned techniques in order to achieve an optimal malware classification result.

Our proposed method is based upon the serial application (see Figure 5.10) of multiple classification methods (*filters*) in order to achieve results that are firstly more rational since the problem is been treated in a multifaceted manner and secondly exhibits an increased classification accuracy ratio. So, generally speaking, our method is based upon the sorting of all families according to the value that they exhibit after the application of each filter, computed as $F_{dominant}$ as we have shown in equations (5.22, 5.23, 5.27 and 5.30).

However, even though the application of solely one filter could be quite convenient, we have faced with cases of tie where multiple families are exposing values that pose them as dominant according to a specific filter. So, in order to completely eliminate any such case we chose to apply a serial order of our techniques based on experimental results. To this point, we ought to refer that the experiments assured one of our intuitions that a false order of the techniques could results to the exclusion of the correct malware family and hence to a false classification result.

Additionally one more sophisticated tuning we performed in the serial application of multiple classification methods is the following. As we referred above the final result of a classification method is the family that includes a member (let us call it *representative*) that actually exhibits the highest similarity with the test sample (according to the applied metric) amongst all of the members in the same family, and additionally among all the other families' *representatives*, inorder to make its family dominant. However, if we change the similarity metric then the same family can be the dominant again but because of another completely different member of her. This lead us to the guarantee that independence of the metrics according to the members should be applied. So, we compute each dominant family without the notion of the member that produce the result. In other words we apply the sequence of metrics without concerning about what member produced the highest result in each family, instead of appliying all the metrics sequentially on each member demanding to be the most similar one across all the metrics.

So next we enumerate the application of sequential filters for the classification of an unknown malware sample into a known malware family, as we ordered them after eperimental verification :

1. Firstly we compute *dominant* family using the *Cover* similarity between the test sample and each of the known families, as presented in equation 5.30

2. If there exists more than one dominant families then we proceed by computing the *Kernel* similarity between the test sample and each of the *dominant* families left from the elimination caused by the application of the *Cover* similarity, as presented in equation 5.27, but by reducing the range only to the dominant families produced after the application of the *Cover* similarity.

3. If still there exists more than one dominant families then we proceed by computing the S^{NP}_{SaMe} similarity between the test sample and each of the *dominant* families left from the elimination caused by the application of the *Kernel* similarity, as presented in equation 5.24, but by reducing the range only to the dominant families produced after the application of the *Kernel* similarity.

4. Finally, if still there exists more than one dominant families then we proceed by computing the *SaMe* similarity between the test sample and each of the *dominant* families left from the elimination caused by the application of the S^{NP}_{SaMe} similarity, as presented in equation 5.22, but by reducing the range only to the dominant families produced after the application of the S^{NP}_{SaMe} similarity.

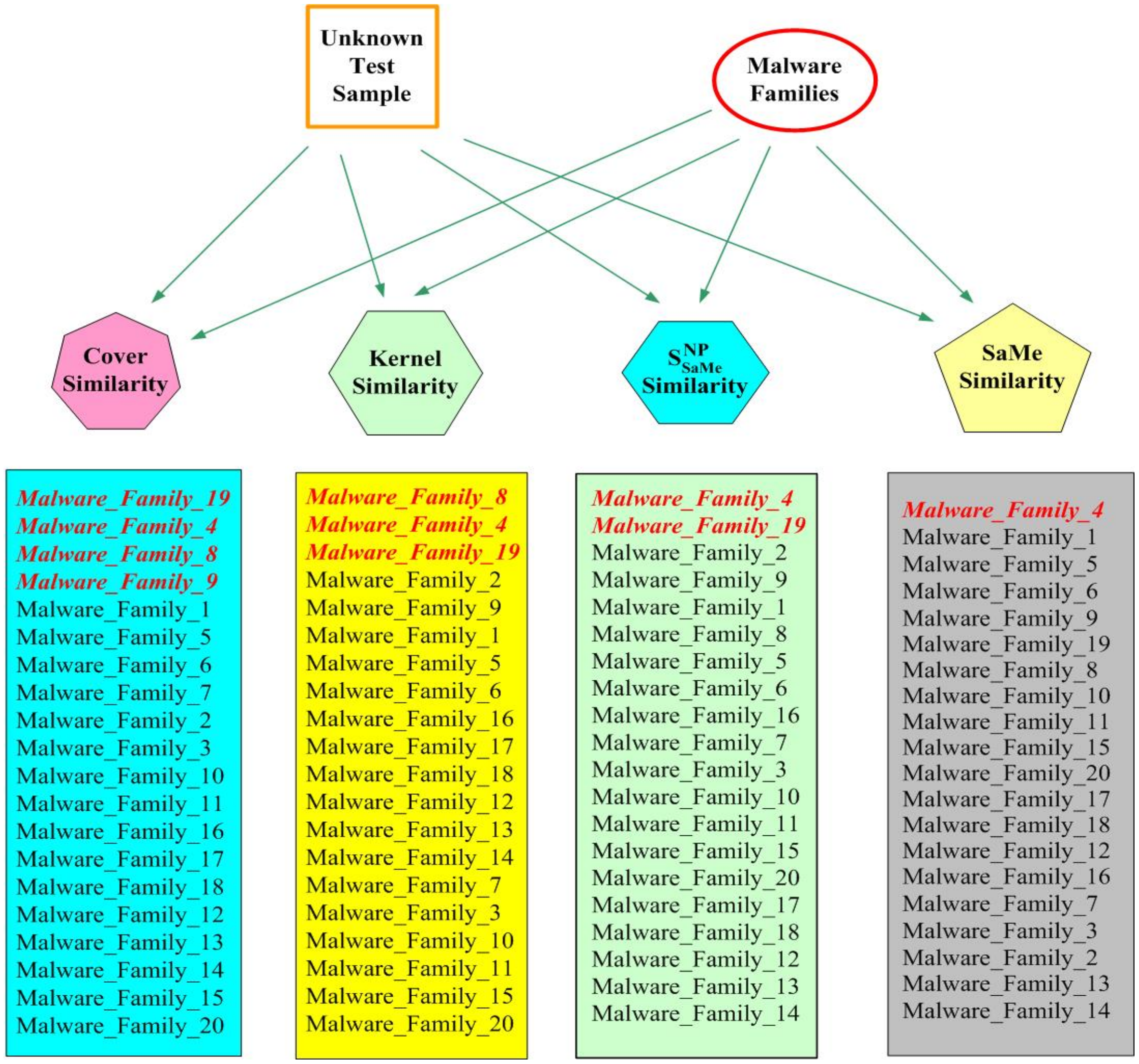


Figure 5.10: Visualization of Malware Classification using Multiple Filters

5.5 Other Approaches for Detection And Classification

In the last section of this chapter we cite a few techniques for malware detection and classification that we tried but failed because, with high probability, it caused due to the phenomenon of phylogeny that takes place across the malware families. However, even though the techniques where inadequate, we feel the duty to refer them in order to prevent other researchers from dedicating time to work with them.

5.5.1 Failed Malware Detection Methods

Starting from the development of our whole detection and classification procedure we began by applying a straightforward approach simply by measuring similarity metric between the test sample and any member of any malware family. However, measuring directly the Jaccard Coefficient, Cosine Similarity, Hamming or Canberra distances between the test sample and any member raised low detection rates and high false positives that lead us to exclude any such approach. Additionally we applied the in, out or in/out degree distribution of the vertices in the G^* graphs in order to observe any similarities that exist between the malware samples while however lack in any of the known benign ones. This approach failed too in an effort to distinguish malicious from benign samples.

5.5.2 Failed Malware Classification Methods

For malware classification, until we reach our final proposed approach, the whole situation was even harder while the existence of existential, qualitative and quantitative characteristics spread among the members of each family posed valuable information that we should leverage. So, our first failed try was to compute the Jaccard coefficient between the test sample and the zone adjacency matrix of a family reducing our range only on computing one distinct Jaccard between aces in the sample's casted adjacency matrix and fours in the family's zone adjacency matrix and one distinct Jaccard between zeros in the sample's casted adjacency matrix and twos in the family's zone adjacency matrix. Finally, one of our last tries that failed too, was the computation of the topological sorting of the vertices of the test samples and a member's G^*'s and the comparison of the produced sequences.

Chapter 6

Results

In this chapter we will discuss our results, starting from presenting our data-set that we used for the evaluation of our proposed model for malware detection and classification. Additionally in this chapter we present the design of our experiments and the methods we decided to utilize in order to evaluate our model, and finally we provide a comparison with other proposed models either for malware detection or for malware classification or for both of them. However, even though, as we will refer later, the other approaches use different evaluation techniques and obviously different data-set we will proceed by comparing our result with the ones produced by other models in order to accomplish a properly documented view of our model's effectiveness.

6.1 Data Set

For the evaluation of our proposed malware detection method we used a dataset of 2631 malware samples pre-classified into 48 malware families where each family contains from 3 to 317 malware members and a set of 33 benign programs. To this point we ought to refer that with the term sample we actually refer to the graph representation of it as it is been achieved by the construction of its System-Call Dependency Graph constructed by processing traces botained through taint analysis during the 120 min execution of any sample in a virtual machine running Windows XP sp2 and having 2.66GHz Intel Core i7 CPU and 8GB RAM . So, having this dataset we proceed by transforming all the 2631

System-Call Dependency Graphs from malware samples and the 33 from the benign ones to G^*s in order to feed our method with a proper input.

Family Name	Members	Family Name	Members
ABU,Banload	16	Hupigon,AWQ	219
Agent,Agent	42	IRCBot,Sdbot	66
Agent,Small	15	LdPinch,LdPinch	16
Allaple,RAHack	201	Lmir,LegMir	23
Ardamax,Ardamax	25	Mydoom,Mydoom	15
Bactera,VB	28	Nilage,Lineage	24
Banbra,Banker	52	OnLineGames,Delf	11
Bancos,Banker	46	OnLineGames,LegMir	76
Banker,Banker	317	OnLineGames,Mmorpg	19
Banker,Delf	20	OnLineGames,OnLineGames	23
Banload,Banker	138	Parite,Pate	71
BDH,Small	5	Plemood,Pupil	32
BGM,Delf	17	PolyCrypt,Swizzor	43
Bifrose,CEP	35	Prorat,AVW	40
Bobax,Bobic	15	Rbot,Sdbot	302
DKI,PoisonIvy	15	SdBot,SdBot	75
DNSChanger,DNSChanger	22	Small,Downloader	29
Downloader,Agent	13	Stration,Warezov	19
Downloader,Delf	22	Swizzor,Obfuscated	27
Downloader,VB	17	Viking,HLLP	32
Gaobot,Agobot	20	Virut,Virut	115
Gobot,Gbot	58	VS,INService	17
Horst,CMQ	48	Zhelatin,ASH	53
Hupigon,ARR	33	Zlob,Puper	64

Table 6.1: Malware Families

Additionally it is of major importance to mention that we did not perform any taint malware analysis on the samples since firstly the development of such processes is out of the scope of this thesis and secondly, and much more important, because we lack of the extremely high levels of expertise demanded when performing procedures like dynamic malware analysis due to the risk posed to the systems connected to the same network. So, we downloaded the initial System Call Dependency Graphs produced by taint analysis from the web-page of Domagoj Babic [2] and proceeded by transforming each sample's System Call Dependency Graph into its auxiliary hyper-abstraction, the so called G^*, based on the grouping on system-calls as presented in Table 5.3 of section 5.1.2. In Table

6.1 we cite the mawlare families and their corresponding number of members in each one of them.

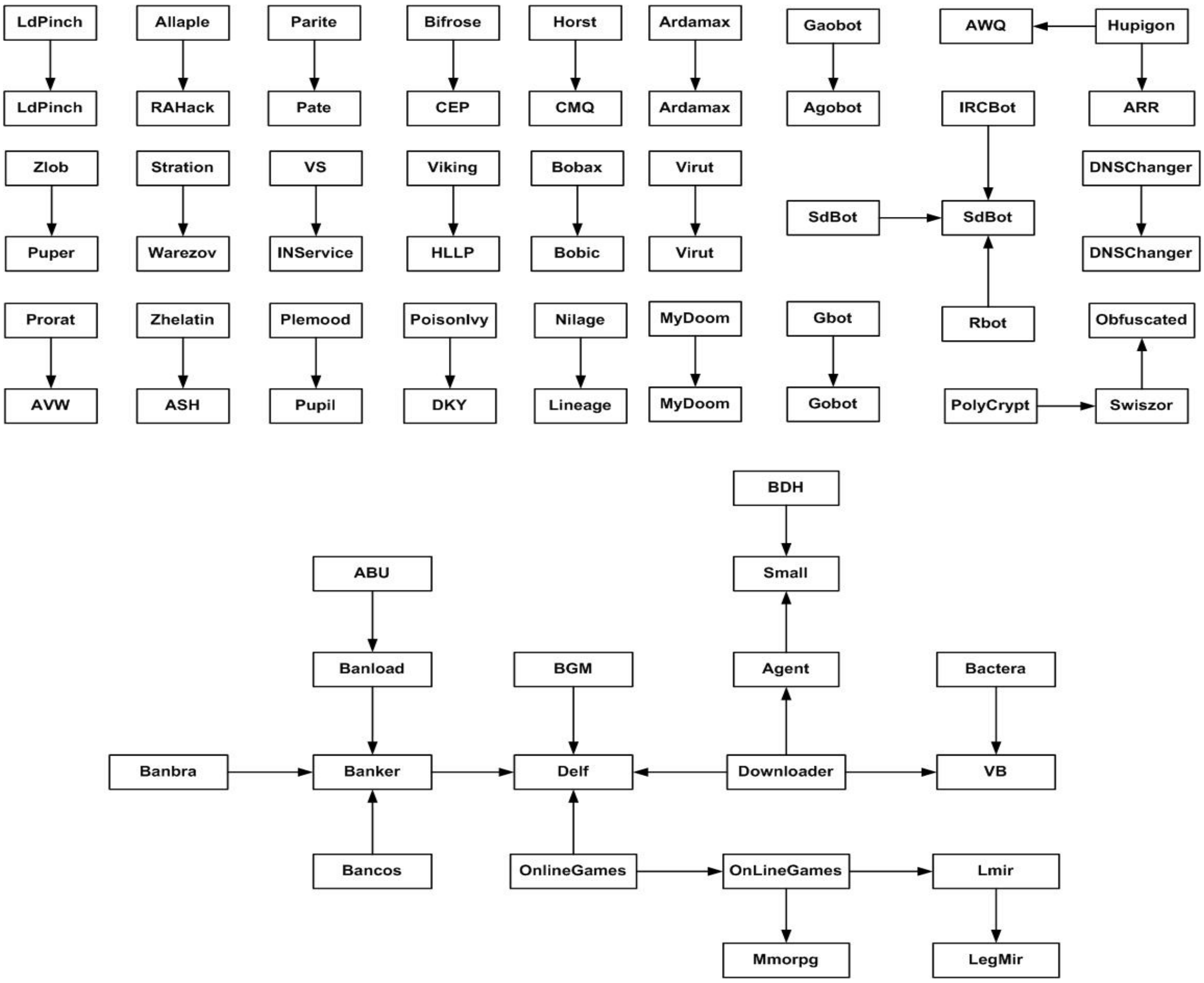


Figure 6.1: Malware Families Connected by Name Commonalities

Finally, it is worth to mention the as we can see there exist families that have common names either appearing in their first or in their second half of their names. This happens because these families are somehow relatives. So, as easily one can understand that this fact consists an obvious instance of phylogeny exhibited between malware families and except from the fact in naming (see Figure 6.1), that is depicted also in our results after the application of the classification method on a test sample from one of the relative families, this phenomenon is observed to happen even in families that even though having totally different names their members exhibit similar functionality and hence conclude in some fashion into a false classification result. However we ought to trust this classification and not increase our classification rate at will.

6.2 Experimental Design

In this section we present our experimental design and discuss the reasons that we choose to proceed with the proposed setup. Additionally we will discuss how we divided our dataset into train-set and test-set and how we tuned our threshold parameters according to feedback produced by sequential experiments.

In order to evaluate our model we performed *5-fold cross validation* utilizing the dataset we described in the previous section. More precisely, we divided each one of the 48 malware families into 5 segments and each time we iterated our experiment by using as test-set each distinct i^{th} 20% of the members of each family and the rest 80% as train-set. So the test-set consists of a compilation of all the 20% of the members of all malware families where th train-set consisted by compiling all the corresponding 80% of the members of all the malware families.

In other words we run series of 5-set experiments each time using a train set consisted of 2100 test sample divided into 48 families and around 500 tests samples that we had hidden their family tag. In each experiment we test both the detection and the classification procedures, while keeping the total detection and classification ratios for each experiment with all the 500 samples. Once an experiment is finished we proceed by checking firstly the detection tag assigned by our system (*benign* or *malware*) counting the false-positive and the true-positives and then we checked the classification tag as to compare if the family that our system has assigned this sample to is the same with its hidden family tag.

To this point, we offer to underline that we performed three types of correct classification counting as we show in the corresponding table in the next section. The first one refers to the *exact* matching in both parts of the names between the name of the family that our system assigned the sample to and the name of the family that exists on the sample's hidden family tag. The second one is the *partial* matching. In this case we count as a correct classification the exact matching in *any* part of the names between the name of the family that our system assigned the sample to and the name of the family that exists on the sample's hidden family tag. Finally the third one is the so called *directed* matching. In this case we count as a correct classification the exact matching in *at least one* part of the names between the name of the family that our system assigned the sample to and the name of the family that exists on the sample's hidden family tag.

Classified as:	**Exact Matchig**	**Partial Matchig**	**Directed Matchig**
Banker,Delf	0	1	0
Banbra,Banker	0	1	1
Banker,Banker	0	1	1
Bancos,Banker	1	1	1

Table 6.2: Classification: Matching Process and Results Accuracy

A more clear representation of these metrics is shown in Table 6.2, where we cite a simple example explaining how these classification accuracy metrics work. Let as assume that we have a sample from family *Bancos,Banker* that has been detected as malware and we classify it into a malware family that is presented in the first column. So, in columns 2,3 and 4 we can observe what would be the result (1 for correct or 0 for wrong classification) according to if we demand *exact*, *partial* or *directed* matching.

Finally, we ought to refer that the system we developed and deploys the graph-based detection and classification methods needs only one tuning, and this is only for the threshold of the detection method (NP-similarity). Actual, threshold resulted after sequential experiments focusing on maximizing the ratio of the number of true-positives divided by the number of false-positives. Actually we achieved by slowing the threshold in 0.57 ($NP(ts, m) \geq 0.57 \rightarrow malware$ or $NP(ts, m) < 0.57 \rightarrow benign$) to minimize the false-positives and false-negatives while maximizing the true-positives and true-negatives.

6.3 Result Comparison

In this section we present our results after a series of experiments and the compare separately our detection rates with the ones achieved by other approaches and our classification rate with rates achieved by other approaches to. However, we ought to notice that as far as we know there have not been published results concerning both detection and classification or the one that have they do not include results comparable to ours.

6.3.1 Detection and Classification Results

Next, we present our results after a series of 5-fold cross validation experiments that we performed using 2631 malware samples from 48 known malware families and 33 benign samples from commodity programs. Next, in Table 6.3 we cite our results from on of a series of 5-fold cross validation experiments. Each line in the table refers to an experiment while the last one refers to the mean value obtained from all 5 experiments.

Experiment	**Detection**	**Exact Matchig**	**Partial Matchig**	**Directed Matchig**
fold 1	99.70 %	70.10 %	82.40 %	81.20 %
fold 2	99.40 %	69.90 %	83.20 %	82.00 %
fold 3	99.50 %	67.40 %	83.00 %	81.40 %
fold 4	99.90 %	68.20 %	84.50 %	83.00 %
fold 5	99.70 %	66.40 %	81.10 %	80.00 %
total	**99.64 %**	**68.40 %**	**82.84 %**	**81.58 %**

Table 6.3: Malware Detection and Classification Results

The first column refers to the detection ratio (true-positives). To this point we ought to refer that the false positives are computed independently of each experiments solely on the 33 benign sample and results a 10%. The second column refers to the Classification ration demanding *exact* matching, while the third and the fourth ones correspond to the *partial* and *directed* matching respectively.

6.3.2 Detection Rate Comparison

Below in Table 6.4, we compare our detection rates (true-positives) and the fail detections (false-positives) against those presented in other works independently of if they are achieved using similar techniques (graph-based) or techniques from other fields and of course independently of the fact that they used different data-sets.

in:	**Technique**	**True Positives**	**False Positives**
[1]	*SVM classifier*	89.74 %	9.74 %
[3]	*SCDG, Tree Automata Inference*	80.00 %	5.00 %
[18]	*CFG, templates*	97.50 %	0.00%
[23]	*SCDG, graph mining*	94.26 %	15.58 %
[35]	*SCDG, sequence matching*	64.00 %	0.00 %
[39]	*SCDG, grading*	80.09 %	11.00%
[66]	*API-sequences, OOA rules*	97.19 %	-
this thesis	*SCDG, NP-similarity*	99.64 %	10.00 %

Table 6.4: Malware Detection Results Comparison

So, in Table 6.4 we present an accumulative view of the aforementioned result comparison where the first column refers to the work that are published the result the second one refers to the utilized technique and the third and fourth columns refer to the detection and false-positive ratios respectively.

In [1], Alazab *et al.*, developed a fully automated system that disassemble and extracts API-call features from executables and then by using $n-gram$ statistical analysis is able to distinguish malicious from benign executables. The mean detection rate exhibited was 89.74% with 9.72% false positives when used a Support Vector Machine (SVM) classifier by applying $n-grams$.

Babic *et al.*, in [3] the malware detection is achieved by k-testable tree automata inference from system call data flow dependence graphs. To this point we ought to underline that in this work Babic *et al.* use the same data-set that we borrow from Domagoj Babic's official web-page, so, this work consists an optimal instance to compare our model's results. However, while 2-fold cross validation was performed in [3] using the first half of data as train-set and the second one as test-set at random we exhibited even better detection rates (almost 20% more) while unfortunately our model had double false-positives (5% more).

Next, Christodorescu *et al.*, in [18] there is presented a malware detection algorith (A_{MD}) based on instruction semantics in order to detect malicious programs. Actually templates of Control Flow Graphs are built in order to demand their satisfiability when a program is malicious. While it seems to exhibit better results than the ones produced by our model, since it exhibits 0 false-positives, however it is a model based on static analysis and hence it would not be proper to compare two methods that while look similar they have a deep edge in their theoretical background.

Fredrikson *et al.*, in [23] is been proposed an automatic technique for extracting optimally discriminative specifications based on graph mining and concept analysis that when used by a behavior based malware detector can distinguish malicious from benign programs. As referred in the corresponding work it can yield an 86% detection rate with 0 false-positives, however we substitutes with the mean of the values as presented in this work as other experiments exhibits higher detection rates but with higher false positives, that is a fair substitution.

Kolbitch *et al.*, in [35] there is been proposed an *effective* and *efficient* approach for malware detection based on behavioral graph matching by detecting string matches in system-call sequences, and that is able to substitute the traditional anti-virus system at the end hosts. The main drawback of this proposed approach is the fact that even if no false-positives where exhibited and even if its is flexible to malware obfuscation, their detection rates are too low in contrast with the those of other approaches.

In [39] Luh and Tavolato, present one more algorithm based on behavioral graphs that distinguishes malicious from benign programs by grading the sample based on report generated from monitoring tools when it is executed in a protected environment. While the produced false-positives are very close to ours, the corresponding detection ration is even lower ensuring the reliability of our model.

Finally, in [66] Ye *et al.*, have developed an integrated system for malware detection based on API-sequences. This is an also different model from ours since the detection process is abased on matching the API-sequences on Objective-Oriented Association rules in order to decide the maliciousness or not of a test program. However, even if the detection rates are high enough there are not false-positive rates mentined in this work

6.3.3 Classification Rate Comparison

Below in Table 6.5, we compare our classification rates against those presented in other works independently of if they are achieved using similar techniques (graph-based) or techniques from other fields and of course independently of the fact that they used different data-sets. So, in Table 6.5 we present an accumulative view of the aforementioned result comparison where the first column refers to the work that are published the result the second one refers to the utilized technique and the third column refers to classification rates.

To this point we ought to refer that, in contrast with the detection results comparison, when we tried to compare our results in classification we realised that the whole situation was even harder because despite the diversity in techniques (they are not many graph-based ones) in most of the works we observed that the result concerned the classification into families measuring the per-family classification rate and not the overall one like we do. However, we proceed in Table 6.5 by citing a few results from other works in order to compare with them our results.

in:	**Technique**	**Classification Ratio**
[6]	*SCDG, behavior profiles,*	95.9 %
[27]	*FCG, Knn*	78,78 %
this thesis	*SCDG, multi-filters(exact)*	68.40 %
this thesis	*SCDG, multi-filters(partial)*	82.84 %
this thesis	*SCDG, multi-filters(directed)*	81.58 %

Table 6.5: Malware Classification Results Comparison

In [6] Bayer *et al.*, propose a scalable clustering approach to identify and group malware samples that exhibit similar behavior, based on *profiles* that characterizing programs activity in a more abstract manner. Since they also use control flow dependencies between system-calls, their work is proper to be compared with ours, even if they do not use direct use of System-Call Dependency Graphs.

Finally, in [27] Hu *et al.*, design implement and evaluate the Symantec's Malware Indexing Tree, that classifies malwares based on their function call graphs useing K nearest neighbor algorithm. Even if their mean classification rates are lower than ours (at least in cases of partial and directed matching) we suppose that this is caused due to the limitations posed by the static analysis performed in that model since they use Function Call Graphs.

6.4 Advantages and Limitations

In this sections we cite our advantages and limitations of our proposed model after having presented a result comparison in the sections of malware detection and malware classification with models from other works that implement either graph-based methods or pure methods from other fields such as data mining.

One of the main advantage is the fact that our model provides high generalization ability since through the hyper-abstraction of System-Call Dependency Graph (G^*)utilized by our model we are able to detect and classify except from the traditionally mutated malware those ones that have been mutated by code arrangements that result to equivalent system-call substitutions in the resulting System-Call Dependency Graph. Additionally, excluding the time demanded during the process of taint analysis our model is claimed to be even faster than the proposed one (the time comparison is to be examined in the future) since it operates on hyper-abstraction of graphs that are consisted by less vertices and less edges.

On the other hand, the main drawback of our proposed model is that the algorithm of inferring the hyper-abstraction (G^*) can be applied only on *labeled graphs* such as the of System-Call Dependency Graphs, thus reducing the range of its application.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we dealt with the topic of detection and classification of mutated malware, by proposing a sophisticated approach for the use of System-Call Dependency Graphs for malware detection and classification. Actually, we leveraged the grouping of system-calls in order to construct a hyper-abstraction of System-Call Dependency Graphs produced during malware execution through taint analysis by tracing system-call dependencies between them, the so called G^*. Then, we developed the NP-similarity metric for malware detection that combines, a relation between an unknown sample System-Call Dependency Graph-hyper abstraction and a combination of System-Call Dependency Graph hyper-abstractions of known malware samples together with a set of similarity metrics between them in order to distinguish if the unknown sample is malicious or not based on a predefined threshold. Next, we developed a series of filter in order to classify any unknown malware sample into one of a set of known malware families based on graph-based similarity metrics similar to the aforementioned ones.

We evaluated our model's detection and classification results performing the corresponding processes on a set of 2630 malware samples from 48 malware families and 33 benign commodity programs. The detection process exhibited a 99.64% rate with 10% false positives while our classification ratio reaches the 82.84% and could be even higher if our dataset was consisted by strictly distinctive families,in terms of phylogeny lack . Finally, we compared our proposed model against other models either graph-based or not, and since there are only slight differences we can claim that that our approach can stand competing against the other approaches.

7.2 Future Work

As we referred in section 6.4, the limitation of our model is the fact that the underlying algorithm for constructing in this case System-Call Dependency Graph hyper-abstraction (G^*) demands as input a labeled graph, such as a System-Call Dependency Graph which its vertices are system-call that can be distinguished by their names. Hence, the application of this model into other topics demand the object of representation to be a labeled graph. Initially this seems to reduce the ranges of our further research, however, below we cite a few paradigms that our model can be applied.

- **Text:** The first application of our algorithm that came to our minds, and consists the core idea of application, was the one of applying our model into measuring similarity between texts, which is very close to plagiarism check . The main idea is that text contains words that can be grouped into classes of *synonyms* (just like the system-call groups). Then, linking groups of synonyms of words that coexist in the same sentence we can construct labeled graphs that are in proper form to be compared by our model.

- **Sound:** Similarly to text, the sound contains notes instead of words. So, if we want to check the similarity between two sound (i.e. songs) we can use our model to compare the labeled graphs produced when grouping notes that are in the same position in every scale.

- **Image:** Similarly to text, images contain colors instead of words. So, if we want to check the similarity between two images (RGB scale) we can use again our model to compare the labeled graphs produced when grouping pixels in groups of same hue and by linking pair of groups if their corresponding pixels co-appear within a range in an area of the image.

- **Chemical unions:** As we know, chemical unions are represented as graphs, where the vertex set consists by chemical elements and edges appear if two elements are compatible according to the number of electrons in the outer layer. Additionally since all the elements in a column of the periodic table have the same number of electrons in the outer layer, then anyone of them can be substituted in a chemical union by anyone that exists in the same column with it in the periodic table. So envisaging as element groups the columns of periodic table we can apply our model to measure the similarity between chemical unions that include different elements based upon the column grouping. Finally, extending this idea we could extent further by applying the same strategy to a more complex type, the bio-molecules. Since bio-molecules are constructed by chemical unions, can be down-casted to complexes of chemical unions and hance to be measured in a similar manner. Now as easily one can understand, extending even further this approach can be applied to biological viruses since they are constructed by biomolecules and so on.

Bibliography

[1] M. Alazab, R. Layton, S. Venkataraman and P. Watters, "Malware detection based on structural and behavioural features of API calls," *Proc. Int'l Cyber Resilience Conference (CR'10)*, pp. 1–10, 2010.

[2] D. Babic, *www.domagoj-babic.com/index.php/ResearchProjects/MalwareAnalysis*

[3] D. Babic, D. Reynaud and D. Song, "Malware analysis with tree automata inference," *Proc. Computer Aided Verification (CAV'11)*, pp. 116–131, 2011.

[4] M. Bailey, J. Oberheide, J. Andersen, Z.M. Mao, F. Jahanian and J. Nazario, "Automated classification and analysis of internet malware," *Proc. Recent Advances in Intrusion Detection, Springer Berlin Heidelberg (RAID'11)*, pp. 78–197, 2007.

[5] H.A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," *Proc. ACM SIGSOFT Software Engineering Notes,* pp. 156–165, 2005.

[6] U. Bayer, P.M. Comparetti, C. Hlauschek, C. Kruegel and E. Kirda, "Scalable behavior-based malware clustering," *Proc. Network and Distributed System Security Symposium (NDSS'09)*, pp. 8–11, 2009.

[7] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda and C. Kruegel, "A view on current malware behaviors," *Proc. 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET'09)*, 2009.

[8] U. Bayer, A. Moser, C. Kruegel and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology* 2 (2006) 67–77 .

[9] G. Bonfante, M. Kaczmarek and J.Y. Marion, "Control flow graphs as malware signatures," *Int'l Workshop on the Theory of Computer Viruses (TCV'07)*, 2007.

[10] L. Cavallaro, P. Saxena and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," *Proc. Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'08)*, pp. 143–163, 2008.

[11] S. Cesare and X. Yang, "A fast flowgraph based classification system for packed and polymorphic malware on the endhost," *Proc. 24th IEEE Int'l Conference on Advanced Information Networking and Applications (AINA'10)*, pp. 721–728, 2010.

[12] S. Cesare and X. Yang, “Malware variant detection using similarity search over sets of control flow graphs,” *Proc. 10th Int’l Conference on Trust, Security and Privacy in Computing and Communications (TrustCom’11)*, pp. 181–189, 2011.

[13] S. Chaumette, O. Ly and R. Tabary, “Automated extraction of polymorphic virus signatures using abstract interpretation,” *Proc. 4th IEEE Int’l Conference on Network and System Security (NSS’11)*, pp. 41–48, 2011.

[14] I. Chionis, S.D. Nikolopoulos and I. Polenakis, “A survey on algorithmic techniques for malware detection,” *Proc. 2nd Int’l Symposium on Computing in Informatics and Mathematics (ISCIM’13)*, pp. 29–34, 2013.

[15] M. Christodorescu and S.Jha, “Static analysis of executables to detect malicious patterns,” *Proc. 12th USENIX Security Symposium,* 2006.

[16] M. Christodorescu and S.Jha, “Testing malware detectors,” *In ACM SIGSOFT Software Engineering Notes* 29 (2004) 34–44.

[17] M. Christodorescu, S. Jha and C. Kruegel, “Mining specifications of malicious behavior,” *Proc. 1st* ACM India Software Engineering Conference, pp. 5–14, 2008.

[18] M. Christodorescu, S. Jha, S.A. Seshia, D. Song and R.E. Bryant, “Semantics-aware malware detection,” *Proc. 2005 IEEE Symposium on Security and Privacy (SP’05)*, pp. 32–46, 2005.

[19] B. Danilo, L. Martignoni and M. Monga, “Using code normalization for fighting self-mutating malware,” *Proc. Int’l Symposium on Secure Software Engineering (ES-SoS’06)*, pp. 37–44, 2006.

[20] G. Debin, M.K.Reiter and D.Song, “Behavioral distance for intrusion detection,” *Proc. Recent Advances in Intrusion Detection (RAID’06)*, pp. 63–81, 2006.

[21] A. Dinaburg, P. Royal, M. Sharif and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” *Proc. 15th ACM Conference on Computer and Communications Security (CCS’08)*, pp. 51-62, 2008.

[22] M. Egele, T. Scholte, E. Kirda and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Computing Surveys 44* (2012), Article 6.

[23] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer and X. Yan, “Synthesizing near-optimal malware specifications from suspicious behaviors,” *Proc. IEEE Symposium on Security and Privacy (SP’10)*, pp. 45–60, 2010.

[24] J. Gregoire, H. Debar and E. Filiol, “Behavioral detection of malware: from a survey towards an established taxonomy,” *Journal in Computer Virology* 4 (2008) 251-266.

[25] B. Guillaume, K. Matthieu and J.Y. Marion, "Architecture of a morphological malware detector," *Journal in Computer Virology* 5 (2009) 263–270.

[26] S.A. Hofmeyr, S. Forrest and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security* 6 (1998) 151–180.

[27] X. Hu, T. Chiueh and K.G. Shin, "Large-scale malware indexing using function-call graphs," *Proc. 16th ACM Conference on Computer and Communications Security (CCS'09)*, pp. 611–620, 2009.

[28] N. Idika and A.P. Mathur, "A survey of malware detection techniques," *Technical Report, Department of Computer Science, Purdue University TR-48,* 2007.

[29] M.E. Karim, A. Walenstein, A. Lakhotia and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology* 1 (2005) 3–23.

[30] M.E. Karim, A. Walenstein, A. Lakhotia and L. Parida, "Malware phylogeny using maximal pi-patterns," *Proc. EICAR Conference (EICAR'05)*, pp. 156–174, 2005.

[31] K. Keehyung and B.R.Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," *Proc. 12th ACM Annual Conference on Genetic and Evolutionary Computation (GECCO'10)*, pp. 1211–1218, 2010.

[32] K. Kendall and C. McMillan, "Practical malware analysis," *In Black Hat Conference, USA*, 2007.

[33] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology* 7 (2011) 233–245.

[34] J. Kinder, S. Katzenbeisser, C. Schallhart and H. Veith, "Detecting malicious code by model checking," *Proc. Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'05)*, pp. 174–187, 2005.

[35] C. Kolbitsch, C.P.M. Comparetti, C. Kruegel, E. Kirda, X.Y. Zhou and X. Wang, "Effective and efficient malware detection at the end host," *Proc. USENIX Security Symposium,* pp. 351–366, 2009.

[36] J.Z. Kolter and M.A. Maloof, "Learning to detect and classify malicious executables in the wild," *The Journal of Machine Learning Research* 7 (2006) 2721–2744.

[37] C. Kruegel, E. Kirda, D. Mutz, W. Robertson and G. Vigna, "Polymorphic worm detection using structural information of executables," *Proc. Recent Advances in Intrusion Detection (RAID'06)*, pp. 207–226, 2006.

[38] C. Kruegel, D. Mutz, F. Valeur and G. Vigna, "On the detection of anomalous system call arguments," *Proc. ESORICS Computer Security (ESORICS'03)*, pp. 326–343, 2003.

[39] R. Luh and P. Tavolato, “Behavior-based malware recognition,” *Fachhochschule St. Polten University of Applied Sciences TR-79-84,* 2012.

[40] K. Mathur and S. Hiranwal, “A survey on techniques in detection and analyzing malware executables,” *Int’l Journal of Advanced Research in Computer Science and Software Engineering* 3 (2013) 422–428.

[41] G. McGraw and G. Morrisett, “Attacking malicious code: A report to the infosec research council,” *Proc. IEEE Software,* pp. 33–41, 2000.

[42] A. Moser, C. Kruegel and E. Kirda, “Exploring multiple execution paths for malware analysis,” *Proc. IEEE Symposium in Security and Privacy (SP’07)*, pp. 231–245, 2007.

[43] A. Moser, C. Kruegel and E. Kirda, “Limits of static analysis for malware detection,” *Proc. 23rd Annual Conference on Computer Security Applications (ACSAC’2007)*, pp. 421-430, 2007.

[44] M. Mungale and M. Stamp, “Software similarity and metamorphic detection,” *Proc. 11th Int’l Conference on Security and Management (SAM’12)*, 2012.

[45] NtTrace, *www.howzatt.demonco.uk/NtTrace/*

[46] Y. Park, D. Reeves,V. Mulukutla and B. Sundaravel, “Fast malware classification by automated behavioral graph matching,” *Proc. 6th ACM Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW’10)*, pp. 45–49, 2010.

[47] B.B. Rad, M. Maslin and I. Suhaimi, “Camouflage in malware: from encryption to metamorphism,” *Int’l Journal of Computer Science and Network Security* 12 (2012) 74–83.

[48] B.B. Rad and M. Masrom, “Metamorphic virus variants classification using opcode frequency histogram,” *ArXiv preprint ArXiv:1104.3228*, 2011.

[49] k. Rieck, H. Thorsten, W. Carsten, D. Patrick and P. Laskov, “Learning and classification of malware behavior,” *Proc. Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA’08)*, pp. 108–125, 2008.

[50] I. Santos, Y.K. Penya, J. Devesa and P.G. Bringas, “N-grams-based file signatures for malware detection,” *Proc. 2nd International Conference on Enterprise Information Systems (ICEIS’09)*, pp. 317–320, 2009.

[51] E.J. Schwartz, T. Avgerinos and D. Brumley, “ All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” *Proc. IEEE Symposium on Security and Privacy (SP’10)*, pp. 317–331, 2010.

[52] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni, “A fast automaton-based method for detecting anomalous program behaviors,” *Proc. IEEE Symposium in Security and Privacy (SP’01)*, pp. 144–155, 2001.

[53] M.K. Shankarapani, S. Ramamoorthy, R.S. Movva and S. Mukkamala, “Malware detection using assembly and API call sequences,” *Journal in Computer Virology* 7 (2011) 107–119.

[54] M.I. Sharif, A. Lanzi, J.T. Giffin and W. Lee, “Impeding malware analysis using conditional code obfuscation,” *Proc. Symposium in Network and Distributed System Security (NDSS’08)*, 2008.

[55] K.A. Al-Sheshtawi, H.M. Abdul-Kader and N.A. Ismail, “Artificial immune clonal selection classification algorithms for classifying malware and benign processes using API call sequences,” *Int’l Journal of Computer Science and Network Security* 10 (2010) 31–39.

[56] M. Sikorski and A. Honig, “Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software,” *No Starch Press*, 2012.

[57] P. Szor and P. Ferrie, “Hunting for metamorphic,” *Proc. Virus Bulletin Conference (VB’01)*, pp. 123–143, 2001.

[58] R.M.H. Ting and J. Bailey, “Mining minimal contrast subgraph patterns,” *Proc. Secure Data Management (SDM’13)*, pp. 639–643, 2006.

[59] P. Vinod, R. Jaipur, V. Laxmi and M. Gaur, “Survey on malware detection methods,” *Proc. 3d Hackers Workshop on Computer and Internet Security (IITKHACK’09)*, pp. 74–79, 2009.

[60] C. Warrender, S. Forrest and B. Pearlmutter, “Detecting intrusions using system calls: Alternative data models,” *Proc. IEEE Symposium in Security and Privacy (SP’99)*, pp. 133–145, 1999.

[61] A. Walenstein and A. Lakhotia, “The software similarity problem in malware analysis,” *Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software,* pp. 10–20, 2007.

[62] A. Walenstein, R. Mathur, M.R. Chouchane and A. Lakhotia, “The design space of metamorphic malware,” *Proc. 2nd Int’l Conference on i-Warfare and Security (ICIW’07)*, pp. 241–248, 2007.

[63] C. Willems, T. Holz and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *Proc. IEEE Security and Privacy (SP’07)*, pp. 32–39, 2007.

[64] J.Y. Xu, A.H. Sung, P. Chavez and S. Mukkamala, "Polymorphic malicious executable scanner by API sequence analysis," *Proc. 4th IEEE Int'l Conference on Hybrid Intelligent Systems (HIS'04)*, pp. 378–383, 2004.

[65] M. Xu, L. Wu, S. Qi, J. Xu, H. Zhang, Y. Ren and N. Zheng, "A similarity metric method of obfuscated malware using function-call graph," *Journal of Computer Virology and Hacking Techniques* 9 (2013) 35–47.

[66] Y. Ye, W. Dingding, L. Tao and Y. Dongyi, "IMDS: Intelligent malware detection system," *Proc. 13th ACM Int'l Conference on Knowledge Discovery and Data Mining (SIGKDD'07)*, pp. 1043–1047, 2007.

[67] Y. Yi, Y. Lingyun, W. Rui, S. Purui and F. Dengguo, "DepSim: a dependency-based malware similarity comparison system," *Proc. Information Security and Cryptology (ISC'11)*, pp. 503–522, 2011.

[68] H. Yin, D. Song, M. Egele, C. Kruegel and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," *Proc. 14th ACM conference on Computer and Communications Security (CCS'07)*, pp. 116–127, 2007.

[69] I.You and K. Yim, "Malware obfuscation techniques: A brief survey," *Proc. Int'l Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA'10)*, pp. 297–300, 2010.

[70] Q. Zhang and D.S. Reeves, "Meta-aware: Identifying metamorphic malware," *Proc. Twenty-Third Annual IEEE Conference on Computer Security Applications (AC-SAC'07)*, pp. 411–420, 2007.

APPENDIX

Alphabetically Ordered System-Calls and Corresponding Groups

A. Name		Group
1) ACCESS_MASK	→	ACCESS_MASK
2) ATOM_INFORMATION_CLASS	→	Atom

D. Name		Group
1) DEBUG_CONTROL_CODE	→	Debug

E. Name		Group
1) EVENT_INFORMATION_CLASS	→	Synchronization
2) EVENT_TYPE	→	Synchronization

F. Name		Group
1) FILE_INFORMATION_CLASS	→	File
2) FS_INFORMATION_CLASS	→	File

H. Name		Group
1) HANDLE	→	HANDLE
2) HARDERROR_RESPONSE_OPTION	→	Process

J. Name		Group
1) JOBOBJECTINFOCLASS	→	Job

K. Name		Group
1) KEY_INFORMATION_CLASS	→	Registry
2) KEY_VALUE_INFORMATION_CLASS	→	Registry

L. Name		Group
1) LONG	→	LONG

M. Name		Group
1) MEMORY_INFORMATION_CLASS	→	Memory

N. Name		Group
1) NtAcceptConnectPort	→	LPC
2) NtAccessCheck	→	Security
3) NtAccessCheckAndAuditAlarm	→	Security
4) NtAccessCheckByType	→	Security
5) NtAccessCheckByTypeAndAuditAlarm	→	Security
6) NtAccessCheckByTypeResultList	→	Security
7) NtAccessCheckByTypeResultListAndAuditAlarm	→	Security
8) NtAccessCheckByTypeResultListAndAuditAlarmByHandle	→	Security
9) NtAcquireCMFViewOwnership	→	Other
10) NtAddAtom	→	Atom
11) NtAddBootEntry	→	Device
12) NtAddDriverEntry	→	Device
13) NtAdjustGroupsToken	→	Security
14) NtAdjustPrivilegesToken	→	Security
15) NtAlertResumeThread	→	Process
16) NtAlertThread	→	Process
17) NtAllocateLocallyUniqueId	→	Other
18) NtAllocateReserveObject	→	Object
19) NtAllocateUserPhysicalPages	→	Memory
20) NtAllocateUuids	→	Other
21) NtAllocateVirtualMemory	→	Memory
22) NtAlpcAcceptConnectPort	→	LPC
23) NtAlpcCancelMessage	→	LPC
24) NtAlpcConnectPort	→	LPC
25) NtAlpcCreatePort	→	LPC

N.	Name		Group
26)	NtAlpcCreatePortSection	→	LPC
27)	NtAlpcCreateResourceReserve	→	LPC
28)	NtAlpcCreateSectionView	→	LPC
29)	NtAlpcCreateSecurityContext	→	LPC
30)	NtAlpcDeletePortSection	→	LPC
31)	NtAlpcDeleteResourceReserve	→	LPC
32)	NtAlpcDeleteSectionView	→	LPC
33)	NtAlpcDeleteSecurityContext	→	LPC
34)	NtAlpcDisconnectPort	→	LPC
35)	NtAlpcImpersonateClientOfPort	→	LPC
36)	NtAlpcOpenSenderProcess	→	LPC
37)	NtAlpcOpenSenderThread	→	LPC
38)	NtAlpcQueryInformation	→	LPC
39)	NtAlpcQueryInformationMessage	→	LPC
40)	NtAlpcRevokeSecurityContext	→	LPC
41)	NtAlpcSendWaitReceivePort	→	LPC
42)	NtAlpcSetInformation	→	LPC
43)	NtApphelpCacheControl	→	Process
44)	NtAreMappedFilesTheSame	→	File
45)	NtAssignProcessToJobObject	→	Job
46)	NtCallbackReturn	→	Other
47)	NtCancelDeviceWakeupRequest	→	Device
48)	NtCancelIoFile	→	File
49)	NtCancelIoFileEx	→	File
50)	NtCancelSynchronousIoFile	→	File
51)	NtCancelTimer	→	Other
52)	NtClearAllSavepointsTransaction	→	Transaction
53)	NtClearEvent	→	Synchronization
54)	NtClearSavepointTransaction	→	Transaction
55)	NtClose	→	File
56)	NtCloseObjectAuditAlarm	→	Security
57)	NtCommitComplete	→	Transaction
58)	NtCommitEnlistment	→	Transaction
59)	NtCommitTransaction	→	Transaction
60)	NtCompactKeys	→	Registry
61)	NtCompareTokens	→	Security
62)	NtCompleteConnectPort	→	LPC
63)	NtCompressKey	→	Registry
64)	NtConnectPort	→	LPC
65)	NtContinue	→	Process
66)	NtCreateChannel	→	LPC

N (cont). Name		Group
67) NtCreateDebugObject	→	Object
68) NtCreateDirectoryObject	→	Object
69) NtCreateEnlistment	→	Transaction
70) NtCreateEvent	→	Synchronization
71) NtCreateEventPair	→	Synchronization
72) NtCreateFile	→	File
73) NtCreateIoCompletion	→	File
74) NtCreateJobObject	→	Job
75) NtCreateJobSet	→	Job
76) NtCreateKey	→	Registry
77) NtCreateKeyedEvent	→	Synchronization
78) NtCreateKeyTransacted	→	Registry
79) NtCreateMailslotFile	→	File
80) NtCreateMutant	→	Synchronization
81) NtCreateNamedPipeFile	→	File
82) NtCreatePagingFile	→	File
83) NtCreatePort	→	LPC
84) NtCreatePrivateNamespace	→	Object
85) NtCreateProcess	→	Process
86) NtCreateProcessEx	→	Process
87) NtCreateProfile	→	Debug
88) NtCreateProfileEx	→	Debug
89) NtCreateResourceManager	→	Transaction
90) NtCreateSection	→	Memory
91) NtCreateSemaphore	→	Synchronization
92) NtCreateSymbolicLinkObject	→	Object
93) NtCreateThread	→	Process
94) NtCreateThreadEx	→	Process
95) NtCreateTimer	→	Other
96) NtCreateToken	→	Security
97) NtCreateTransaction	→	Transaction
98) NtCreateTransactionManager	→	Transaction
99) NtCreateUserProcess	→	Process
100) NtCreateWaitablePort	→	LPC
101) NtCreateWorkerFactory	→	Process
102) NtCurrentTeb	→	Other
103) NtDebugActiveProcess	→	Debug
104) NtDebugContinue	→	Debug
105) NtDelayExecution	→	Process
106) NtDeleteAtom	→	Atom
107) NtDeleteBootEntry	→	Device
108) NtDeleteDriverEntry	→	Device

N (cont). Name		Group
109) NtDeleteFile	→	File
110) NtDeleteKey	→	Registry
111) NtDeleteObjectAuditAlarm	→	Security
112) NtDeletePrivateNamespace	→	Object
113) NtDeleteValueKey	→	Registry
114) NtDeviceIoControlFile	→	File
115) NtDisableLastKnownGood	→	Device
116) NtDisplayString	→	Other
117) NtDrawText	→	Other
118) NtDuplicateObject	→	Object
119) NtDuplicateToken	→	Security
120) NtEnableLastKnownGood	→	Device
121) NtEnumerateBootEntries	→	Device
122) NtEnumerateDriverEntries	→	Device
123) NtEnumerateKey	→	Registry
124) NtEnumerateSystemEnvironmentValuesEx	→	Environment
125) NtEnumerateTransactionObject	→	Transaction
126) NtEnumerateValueKey	→	Registry
127) NtExtendSection	→	Memory
128) NtFilterToken	→	Security
129) NtFindAtom	→	Atom
130) NtFlushBuffersFile	→	File
131) NtFlushInstallUILanguage	→	Other
132) NtFlushInstructionCache	→	Memory
133) NtFlushKey	→	Registry
134) NtFlushProcessWriteBuffers	→	Memory
135) NtFlushVirtualMemory	→	Memory
136) NtFlushWriteBuffer	→	Memory
137) NtFreeUserPhysicalPages	→	Memory
138) NtFreeVirtualMemory	→	Memory
139) NtFreezeRegistry	→	Registry
140) NtFreezeTransactions	→	Transaction
141) NtFsControlFile	→	File
142) NtGetContextThread	→	Process
143) NtGetCurrentProcessorNumber	→	Environment
144) NtGetDevicePowerState	→	Device
145) NtGetMUIRegistryInfo	→	Other
146) NtGetNextProcess	→	Process
147) NtGetNextThread	→	Process
148) NtGetNlsSectionPtr	→	Other
149) NtGetNotificationResourceManager	→	Transaction
150) NtGetPlugPlayEvent	→	Device

N (cont). Name		Group
151) NtGetTickCount	→	Other
152) NtGetWriteWatch	→	Other
153) NtImpersonateAnonymousToken	→	Security
154) NtImpersonateClientOfPort	→	LPC
155) NtImpersonateThread	→	Security
156) NtInitializeNlsFiles	→	Other
157) NtInitializeRegistry	→	Registry
158) NtInitiatePowerAction	→	Device
159) NtIsProcessInJob	→	Job
160) NtIsSystemResumeAutomatic	→	Environment
161) NtIsUILanguageComitted	→	Other
162) NtListenChannel	→	LPC
163) NtListenPort	→	LPC
164) NtListTransactions	→	Transaction
165) NtLoadDriver	→	Device
166) NtLoadKey	→	Registry
167) NtLoadKey2	→	Registry
168) NtLoadKeyEx	→	Registry
169) NtLockFile	→	File
170) NtLockProductActivationKeys	→	Other
171) NtLockRegistryKey	→	Registry
172) NtLockVirtualMemory	→	Memory
173) NtMakePermanentObject	→	Object
174) NtMakeTemporaryObject	→	Object
175) NtMapCMFModule	→	Other
176) NtMapUserPhysicalPages	→	Memory
177) NtMapUserPhysicalPagesScatter	→	Memory
178) NtMapViewOfSection	→	Memory
179) NtMarshallTransaction	→	Transaction
180) NtModifyBootEntry	→	Device
181) NtModifyDriverEntry	→	Device
182) NtNotifyChangeDirectoryFile	→	File
183) NtNotifyChangeKey	→	Registry
184) NtNotifyChangeMultipleKeys	→	Registry
185) NtNotifyChangeSession	→	Other
186) NtOpenChannel	→	LPC
187) NtOpenDirectoryObject	→	Object
188) NtOpenEnlistment	→	Transaction
189) NtOpenEvent	→	Synchronization
190) NtOpenEventPair	→	Synchronization
191) NtOpenFile	→	File
192) NtOpenIoCompletion	→	File

N (cont). Name		Group
193) NtOpenJobObject	→	Job
194) NtOpenKey	→	Registry
195) NtOpenKeyedEvent	→	Synchronization
196) NtOpenKeyEx	→	Registry
197) NtOpenKeyTransacted	→	Registry
198) NtOpenKeyTransactedEx	→	Registry
199) NtOpenMutant	→	Synchronization
200) NtOpenObjectAuditAlarm	→	Object
201) NtOpenPrivateNamespace	→	Object
202) NtOpenProcess	→	Process
203) NtOpenProcessToken	→	Security
204) NtOpenProcessTokenEx	→	Security
205) NtOpenResourceManager	→	Transaction
206) NtOpenSection	→	Memory
207) NtOpenSemaphore	→	Synchronization
208) NtOpenSession	→	Other
209) NtOpenSymbolicLinkObject	→	Object
210) NtOpenThread	→	Process
211) NtOpenThreadToken	→	Security
212) NtOpenThreadTokenEx	→	Security
213) NtOpenTimer	→	Other
214) NtOpenTransaction	→	Transaction
215) NtOpenTransactionManager	→	Transaction
216) NtPlugPlayControl	→	Device
217) NtPowerInformation	→	Device
218) NtPrepareComplete	→	Transaction
219) NtPrepareEnlistment	→	Transaction
220) NtPrePrepareComplete	→	Transaction
221) NtPrePrepareEnlistment	→	Transaction
222) NtPrivilegeCheck	→	Security
223) NtPrivilegedServiceAuditAlarm	→	Security
224) NtPrivilegeObjectAuditAlarm	→	Security
225) NtPropagationComplete	→	Transaction
226) NtPropagationFailed	→	Transaction
227) NtProtectVirtualMemory	→	Memory
228) NtPullTransaction	→	Transaction
229) NtPulseEvent	→	Synchronization
230) NtQueryAttributesFile	→	File
231) NtQueryBootEntryOrder	→	Device
232) NtQueryBootOptions	→	Device
233) NtQueryDebugFilterState	→	Debug
234) NtQueryDefaultLocale	→	Environment

N (cont). Name		**Group**
235) NtQueryDefaultUILanguage	→	Environment
236) NtQueryDirectoryFile	→	File
237) NtQueryDirectoryObject	→	Object
238) NtQueryDriverEntryOrder	→	Device
239) NtQueryEaFile	→	File
240) NtQueryEvent	→	Synchronization
241) NtQueryFullAttributesFile	→	File
242) NtQueryInformationAtom	→	Atom
243) NtQueryInformationEnlistment	→	Transaction
244) NtQueryInformationFile	→	File
245) NtQueryInformationJobObject	→	Job
246) NtQueryInformationPort	→	LPC
247) NtQueryInformationProcess	→	Process
248) NtQueryInformationResourceManager	→	Transaction
249) NtQueryInformationThread	→	Process
250) NtQueryInformationToken	→	Security
251) NtQueryInformationTransaction	→	Transaction
252) NtQueryInformationTransactionManager	→	Transaction
253) NtQueryInformationWorkerFactory	→	Process
254) NtQueryInstallUILanguage	→	Environment
255) NtQueryIntervalProfile	→	Debug
256) NtQueryIoCompletion	→	File
257) NtQueryKey	→	Registry
258) NtQueryLicenseValue	→	Other
259) NtQueryMultipleValueKey	→	Registry
260) NtQueryMutant	→	Synchronization
261) NtQueryObject	→	Object
262) NtQueryOleDirectoryFile	→	File
263) NtQueryOpenSubKeys	→	Registry
264) NtQueryOpenSubKeysEx	→	Registry
265) NtQueryPerformanceCounter	→	Debug
266) NtQueryPortInformationProcess	→	LPC
267) NtQueryQuotaInformationFile	→	File
268) NtQuerySection	→	Memory
269) NtQuerySecurityAttributesToken	→	Security
270) NtQuerySecurityObject	→	Security
271) NtQuerySemaphore	→	Synchronization
272) NtQuerySymbolicLinkObject	→	Object
273) NtQuerySystemEnvironmentValue	→	Environment
274) NtQuerySystemEnvironmentValueEx	→	Environment
275) NtQuerySystemInformation	→	Other
276) NtQuerySystemInformationEx	→	Other

N (cont). Name		Group
277) NtQuerySystemTime	→	Time
278) NtQueryTimer	→	Time
279) NtQueryTimerResolution	→	Time
280) NtQueryValueKey	→	Registry
281) NtQueryVirtualMemory	→	Memory
282) NtQueryVolumeInformationFile	→	File
283) NtQueueApcThread	→	Process
284) NtQueueApcThreadEx	→	Process
285) NtRaiseException	→	Process
286) NtRaiseHardError	→	Process
287) NtReadFile	→	File
288) NtReadFileScatter	→	File
289) NtReadOnlyEnlistment	→	Transaction
290) NtReadRequestData	→	LPC
291) NtReadVirtualMemory	→	Memory
292) NtRecoverEnlistment	→	Transaction
293) NtRecoverResourceManager	→	Transaction
294) NtRecoverTransactionManager	→	Transaction
295) NtRegisterProtocolAddressInformation	→	Transaction
296) NtRegisterThreadTerminatePort	→	Debug
297) NtReleaseCMFViewOwnership	→	Other
298) NtReleaseKeyedEvent	→	Synchronization
299) NtReleaseMutant	→	Synchronization
300) NtReleaseSemaphore	→	Synchronization
301) NtReleaseWorkerFactoryWorker	→	Process
302) NtRemoveIoCompletion	→	File
303) NtRemoveIoCompletionEx	→	File
304) NtRemoveProcessDebug	→	Debug
305) NtRenameKey	→	Registry
306) NtRenameTransactionManager	→	Transaction
307) NtReplaceKey	→	Registry
308) NtReplacePartitionUnit	→	Device
309) NtReplyPort	→	LPC
310) NtReplyWaitReceivePort	→	LPC
311) NtReplyWaitReceivePortEx	→	LPC
312) NtReplyWaitReplyPort	→	LPC
313) NtReplyWaitSendChannel	→	LPC
314) NtRequestDeviceWakeup	→	Device
315) NtRequestPort	→	LPC
316) NtRequestWaitReplyPort	→	LPC
317) NtRequestWakeupLatency	→	Device
318) NtResetEvent	→	Synchronization

N (cont). Name		Group
319) NtResetWriteWatch	→	Other
320) NtRestoreKey	→	Registry
321) NtResumeProcess	→	Process
322) NtResumeThread	→	Process
323) NtRollbackComplete	→	Transaction
324) NtRollbackEnlistment	→	Transaction
325) NtRollbackSavepointTransaction	→	Transaction
326) NtRollbackTransaction	→	Transaction
327) NtRollforwardTransactionManager	→	Transaction
328) NtSaveKey	→	Registry
329) NtSaveKeyEx	→	Registry
330) NtSaveMergedKeys	→	Registry
331) NtSavepointComplete	→	Transaction
332) NtSavepointTransaction	→	Transaction
333) NtSecureConnectPort	→	LPC
334) NtSendWaitReplyChannel	→	LPC
335) NtSerializeBoot	→	Device
336) NtSetBootEntryOrder	→	Device
337) NtSetBootOptions	→	Device
338) NtSetContextChannel	→	LPC
339) NtSetContextThread	→	Process
340) NtSetDebugFilterState	→	Debug
341) NtSetDefaultHardErrorPort	→	Process
342) NtSetDefaultLocale	→	Environment
343) NtSetDefaultUILanguage	→	Environment
344) NtSetDriverEntryOrder	→	Device
345) NtSetEaFile	→	File
346) NtSetEvent	→	Synchronization
347) NtSetEventBoostPriority	→	Synchronization
348) NtSetHighEventPair	→	Synchronization
349) NtSetHighWaitLowEventPair	→	Synchronization
350) NtSetHighWaitLowThread	→	Synchronization
351) NtSetInformationDebugObject	→	Debug
352) NtSetInformationEnlistment	→	Transaction
353) NtSetInformationFile	→	File
354) NtSetInformationJobObject	→	Job
355) NtSetInformationKey	→	Registry
356) NtSetInformationObject	→	Object
357) NtSetInformationProcess	→	Process
358) NtSetInformationResourceManager	→	Transaction
359) NtSetInformationThread	→	Process
360) NtSetInformationToken	→	Security

N (cont). Name		Group
361) NtSetInformationTransaction	→	Transaction
362) NtSetInformationTransactionManager	→	Transaction
363) NtSetInformationWorkerFactory	→	Process
364) NtSetIntervalProfile	→	Debug
365) NtSetIoCompletion	→	File
366) NtSetIoCompletionEx	→	File
367) NtSetLdtEntries	→	Other
368) NtSetLowEventPair	→	Synchronization
369) NtSetLowWaitHighEventPair	→	Synchronization
370) NtSetLowWaitHighThread	→	Synchronization
371) NtSetQuotaInformationFile	→	File
372) NtSetSecurityObject	→	Security
373) NtSetSystemEnvironmentValue	→	Environment
374) NtSetSystemEnvironmentValueEx	→	Environment
375) NtSetSystemInformation	→	Other
376) NtSetSystemPowerState	→	Device
377) NtSetSystemTime	→	Time
378) NtSetThreadExecutionState	→	Device
379) NtSetTimer	→	Other
380) NtSetTimerEx	→	Other
381) NtSetTimerResolution	→	Time
382) NtSetUuidSeed	→	Other
383) NtSetValueKey	→	Registry
384) NtSetVolumeInformationFile	→	File
385) NtShutdownSystem	→	Other
386) NtShutdownWorkerFactory	→	Process
387) NtSignalAndWaitForSingleObject	→	Synchronization
388) NtSinglePhaseReject	→	Transaction
389) NtStartProfile	→	Debug
390) NtStartTm	→	Transaction
391) NTSTATUS	→	NTSTATUS
392) NtStopProfile	→	Debug
393) NtSuspendProcess	→	Process
394) NtSuspendThread	→	Process
395) NtSystemDebugControl	→	Debug
396) NtTerminateJobObject	→	Job
397) NtTerminateProcess	→	Process
398) NtTerminateThread	→	Process
399) NtTestAlert	→	Process
400) NtThawRegistry	→	Registry
401) NtThawTransactions	→	Transaction
402) NtTraceControl	→	Other

N (cont). Name		Group
403) NtTraceEvent	→	Synchronization
404) NtTranslateFilePath	→	File
405) NtUmsThreadYield	→	Process
406) NtUnloadDriver	→	Device
407) NtUnloadKey	→	Registry
408) NtUnloadKey2	→	Registry
409) NtUnloadKeyEx	→	Registry
410) NtUnlockFile	→	File
411) NtUnlockVirtualMemory	→	Memory
412) NtUnmapViewOfSection	→	Memory
413) NtVdmControl	→	Device
414) NtWaitForDebugEvent	→	Debug
415) NtWaitForKeyedEvent	→	Synchronization
416) NtWaitForMultipleObjects	→	Synchronization
417) NtWaitForMultipleObjects32	→	Synchronization
418) NtWaitForSingleObject	→	Synchronization
419) NtWaitForWorkViaWorkerFactory	→	Process
420) NtWaitHighEventPair	→	Synchronization
421) NtWaitLowEventPair	→	Synchronization
422) NtWorkerFactoryWorkerReady	→	Process
423) NtWow64CallFunction64	→	WOW64
424) NtWow64CsrAllocateCaptureBuffer	→	WOW64
425) NtWow64CsrAllocateMessagePointer	→	WOW64
426) NtWow64CsrCaptureMessageBuffer	→	WOW64
427) NtWow64CsrCaptureMessageString	→	WOW64
428) NtWow64CsrClientCallServer	→	WOW64
429) NtWow64CsrClientConnectToServer	→	WOW64
430) NtWow64CsrFreeCaptureBuffer	→	WOW64
431) NtWow64CsrGetProcessId	→	WOW64
432) NtWow64CsrIdentifyAlertableThread	→	WOW64
433) NtWow64CsrVerifyRegion	→	WOW64
434) NtWow64DebuggerCall	→	WOW64
435) NtWow64GetCurrentProcessorNumberEx	→	WOW64
436) NtWow64GetNativeSystemInformation	→	WOW64
437) NtWow64InterlockedPopEntrySList	→	WOW64
438) NtWow64QueryInformationProcess64	→	WOW64
439) NtWow64QueryVirtualMemory64	→	WOW64
440) NtWow64ReadVirtualMemory64	→	WOW64
441) NtWow64WriteVirtualMemory64	→	WOW64
442) NtWriteFile	→	File
443) NtWriteFileGather	→	File
444) NtWriteRequestData	→	LPC

N (cont). Name		Group
445) NtWriteVirtualMemory	→	Memory
446) NtYieldExecution	→	Process

O. Name		Group
1) OBJECT_INFORMATION_CLASS	→	Object

P. Name		Group
1) PCLIENT_ID	→	Process
2) PCONTEXT	→	Process
3) PEXCEPTION_RECORD	→	Process
4) PGENERIC_MAPPING	→	Security
5) PHANDLE	→	PHANDLE
6) PIO_APC_ROUTINE	→	File
7) PLARGE_INTEGER	→	PLARGE_INTEGER
8) POBJECT_ATTRIBUTES	→	Object
9) PPORT_MESSAGE	→	LPC
10) PPORT_VIEW	→	LPC
11) PROCESSINFOCLASS	→	Process
12) PSECURITY_DESCRIPTOR	→	Security
13) PSECURITY_QUALITY_OF_SERVICE	→	Security
14) PTIMER_APC_ROUTINE	→	Other
15) PTOKEN_PRIVILEGES	→	Security
16) PULARGE_INTEGER	→	PULARGE_INTEGER
17) PULONG	→	PULONG
18) PULONG_PTR	→	Process
19) PUNICODE_STRING	→	PUNICODE_STRING
20) PUSER_STACK	→	Process
21) PVOID_SIZEAFTER	→	PVOID_SIZEAFTER
22) PWSTR	→	PWSTR

S. Name		Group
1) SECTION_INFORMATION_CLASS	→	Memory
2) SECTION_INHERIT	→	Memory
3) SECURITY_INFORMATION	→	Security
4) SYSTEM_INFORMATION_CLASS	→	Other

T. Name		Group
1) THREADINFOCLASS	→	Process
2) TIMER_TYPE	→	Other
3) TOKEN_INFORMATION_CLASS	→	Security
4) TOKEN_TYPE	→	Security

U. Name		Group
1) ULONG	→	ULONG

W. Name		Group
1) WAIT_TYPE	→	Synchronization

Alphabetically Ordered Groups and Corresponding System-Calls

1. ACCESS_MASK

1) ACCESS_MASK

2. Atom

1) ATOM_INFORMATION_CLASS
2) NtAddAtom
3) NtDeleteAtom
4) NtFindAtom
5) NtQueryInformationAtom

3. BOOLEAN

1) BOOLEAN

4. DEBUG

1) DEBUG_CONTROL_CODE
2) NtCreateProfile
3) NtCreateProfileEx
4) NtDebugActiveProcess
5) NtDebugContinue
6) NtQueryDebugFilterState
7) NtQueryIntervalProfile
8) NtQueryPerformanceCounter
9) NtRegisterThreadTerminatePort
10) NtRemoveProcessDebug
11) NtSetDebugFilterState
12) NtSetInformationDebugObject
13) NtSetIntervalProfile
14) NtStartProfile
15) NtStopProfile
16) NtSystemDebugControl
17) NtWaitForDebugEvent

5. DEVICE

1) NtAddBootEntry
2) NtAddDriverEntry
3) NtCancelDeviceWakeupRequest
4) NtDeleteBootEntry
5) NtDeleteDriverEntry
6) NtDisableLastKnownGood
7) NtEnableLastKnownGood
8) NtEnumerateBootEntries
9) NtEnumerateDriverEntries
10) NtGetDevicePowerState
11) NtGetPlugPlayEvent
12) NtInitiatePowerAction
13) NtLoadDriver
14) NtModifyBootEntry
15) NtModifyDriverEntry
16) NtPlugPlayControl
17) NtPowerInformation
18) NtQueryBootEntryOrder
19) NtQueryBootOptions
20) NtQueryDriverEntryOrder
21) NtReplacePartitionUnit
22) NtRequestDeviceWakeup
23) NtRequestWakeupLatency
24) NtSerializeBoot
25) NtSetBootEntryOrder
26) NtSetBootOptions

5. DEVICE (cont.)

27) NtSetDriverEntryOrder
28) NtSetSystemPowerState
29) NtSetThreadExecutionState
30) NtUnloadDriver
31) NtVdmControl

6. ENVIRONMENT

1) NtEnumerateSystemEnvironmentValuesEx
2) NtGetCurrentProcessorNumber
3) NtIsSystemResumeAutomatic
4) NtQueryDefaultLocale
5) NtQueryDefaultUILanguage
6) NtQueryInstallUILanguage
7) NtQuerySystemEnvironmentValue
8) NtQuerySystemEnvironmentValueEx
9) NtSetDefaultLocale
10) NtSetDefaultUILanguage
11) NtSetSystemEnvironmentValue
12) NtSetSystemEnvironmentValueEx

7. FILE

1) FILE_INFORMATION_CLASS
2) FS_INFORMATION_CLASS
3) NtAreMappedFilesTheSame
4) NtCancelIoFile
5) NtCancelIoFileEx
6) NtCancelSynchronousIoFile
7) NtClose
8) NtCreateFile
9) NtCreateIoCompletion
10) NtCreateMailslotFile
11) NtCreateNamedPipeFile
12) NtCreatePagingFile
13) NtDeleteFile
14) NtDeviceIoControlFile
15) NtFlushBuffersFile
16) NtFsControlFile
17) NtLockFile
18) NtNotifyChangeDirectoryFile
19) NtOpenFile
20) NtOpenIoCompletion
21) NtQueryAttributesFile
22) NtQueryDirectoryFile
23) NtQueryEaFile
24) NtQueryFullAttributesFile
25) NtQueryInformationFile
26) NtQueryIoCompletion
27) NtQueryOleDirectoryFile
28) NtQueryQuotaInformationFile
29) NtQueryVolumeInformationFile
30) NtReadFile
31) NtReadFileScatter
32) NtRemoveIoCompletion
33) NtRemoveIoCompletionEx
34) NtSetEaFile
35) NtSetInformationFile
36) NtSetIoCompletion
37) NtSetIoCompletionEx
38) NtSetQuotaInformationFile
39) NtSetVolumeInformationFile
40) NtTranslateFilePath
41) NtUnlockFile
42) NtWriteFile
43) NtWriteFileGather
44) PIO_APC_ROUTINE

8. HANDLE

1) HANDLE

9. JOB

1) JOBOBJECTINFOCLASS
2) NtAssignProcessToJobObject
3) NtCreateJobObject
4) NtCreateJobSet
5) NtIsProcessInJob
6) NtOpenJobObject
7) NtQueryInformationJobObject
8) NtSetInformationJobObject
9) NtTerminateJobObject

10. LONG

1) LONG

11. LPC

1) NtAcceptConnectPort
2) NtAlpcAcceptConnectPort
3) NtAlpcCancelMessage
4) NtAlpcConnectPort
5) NtAlpcCreatePort
6) NtAlpcCreatePortSection
7) NtAlpcCreateResourceReserve
8) NtAlpcCreateSectionView
9) NtAlpcCreateSecurityContext
10) NtAlpcDeletePortSection
11) NtAlpcDeleteResourceReserve
12) NtAlpcDeleteSectionView
13) NtAlpcDeleteSecurityContext
14) NtAlpcDisconnectPort
15) NtAlpcImpersonateClientOfPort
16) NtAlpcOpenSenderProcess
17) NtAlpcOpenSenderThread
18) NtAlpcQueryInformation
19) NtAlpcQueryInformationMessage
20) NtAlpcRevokeSecurityContext
21) NtAlpcSendWaitReceivePort
22) NtAlpcSetInformation
23) NtCompleteConnectPort
24) NtConnectPort
25) NtCreateChannel
26) NtCreatePort
27) NtCreateWaitablePort
28) NtImpersonateClientOfPort
29) NtListenChannel
30) NtListenPort
31) NtOpenChannel
32) NtQueryInformationPort
33) NtQueryPortInformationProcess
34) NtReadRequestData
35) NtReplyPort
36) NtReplyWaitReceivePort
37) NtReplyWaitReceivePortEx
38) NtReplyWaitReplyPort
39) NtReplyWaitSendChannel
40) NtRequestPort
41) NtRequestWaitReplyPort
42) NtSecureConnectPort
43) NtSendWaitReplyChannel
44) NtSetContextChannel
45) NtWriteRequestData
46) PPORT_MESSAGE
47) PPORT_VIEW

12. MEMORY

1) MEMORY_INFORMATION_CLASS
2) NtAllocateUserPhysicalPages
3) NtAllocateVirtualMemory
4) NtCreateSection
5) NtExtendSection
6) NtFlushInstructionCache
7) NtFlushProcessWriteBuffers
8) NtFlushVirtualMemory
9) NtFlushWriteBuffer
10) NtFreeUserPhysicalPages

12. MEMORY (cont.)

11) NtFreeVirtualMemory
12) NtMapUserPhysicalPagesScatter
13) NtMapViewOfSection
14) NtOpenSection
15) NtProtectVirtualMemory
16) NtQuerySection
17) NtLockVirtualMemory
18) NtMapUserPhysicalPages
19) NtQueryVirtualMemory
20) NtReadVirtualMemory
21) NtUnlockVirtualMemory
22) NtUnmapViewOfSection
23) NtWriteVirtualMemory
24) SECTION_INFORMATION_CLASS
25) SECTION_INHERIT

13. NTSTATUS

1) NTSTATUS

14. OBJECT

1) NtAllocateReserveObject
2) NtCreateDebugObject
3) NtCreateDirectoryObject
4) NtCreatePrivateNamespace
5) NtCreateSymbolicLinkObject
6) NtDeletePrivateNamespace
7) NtDuplicateObject
8) NtMakePermanentObject
9) NtMakeTemporaryObject
10) NtOpenDirectoryObject
11) NtOpenObjectAuditAlarm
12) NtOpenPrivateNamespace
13) NtOpenSymbolicLinkObject
14) NtQueryDirectoryObject
15) NtQueryObject
16) NtQuerySymbolicLinkObject
17) NtSetInformationObject
18) OBJECT_INFORMATION_CLASS
19) POBJECT_ATTRIBUTES

15. OTHER

1) NtAcquireCMFViewOwnership
2) NtAllocateLocallyUniqueId
3) NtAllocateUuids
4) NtCallbackReturn
5) NtCancelTimer
6) NtCreateTimer
7) NtCurrentTeb
8) NtDisplayString
9) NtDrawText
10) NtFlushInstallUILanguage
11) NtGetMUIRegistryInfo
12) NtGetNlsSectionPtr
13) NtGetTickCount
14) NtGetWriteWatch
15) NtInitializeNlsFiles
16) NtIsUILanguageComitted
17) NtLockProductActivationKeys
18) NtMapCMFModule
19) NtNotifyChangeSession
20) NtOpenSession
21) NtOpenTimer
22) NtQueryLicenseValue
23) NtQuerySystemInformation
24) NtQuerySystemInformationEx
25) NtReleaseCMFViewOwnership
26) NtResetWriteWatch

15. OTHER (cont.)

27) NtSetLdtEntries
28) NtSetSystemInformation
29) NtSetTimer
30) NtSetTimerEx
31) NtSetUuidSeed
32) NtShutdownSystem
33) NtTraceControl
34) PTIMER_APC_ROUTINE
35) SYSTEM_INFORMATION_CLASS
36) TIMER_TYPE

16. PHANDLE

1) PHANDLE

17. LARGE_INTEGER

1) PLARGE_INTEGER

18. PROCESS

1) HARDERROR_RESPONSE_OPTION
2) NtAlertResumeThread
3) NtAlertThread
4) NtApphelpCacheControl
5) NtContinue
6) NtCreateProcess
7) NtCreateProcessEx
8) NtCreateThread
9) NtCreateThreadEx
10) NtCreateUserProcess
11) NtCreateWorkerFactory
12) NtDelayExecution
13) NtGetContextThread
14) NtGetNextProcess
15) NtGetNextThread
16) NtOpenProcess
17) NtOpenThread
18) NtQueryInformationProcess
19) NtQueryInformationThread
20) NtQueryInformationWorkerFactory
21) NtQueueApcThread
22) NtQueueApcThreadEx
23) NtRaiseException
24) NtRaiseHardError
25) NtReleaseWorkerFactoryWorker
26) NtResumeProcess
27) NtResumeThread
28) NtSetContextThread
29) NtSetDefaultHardErrorPort
30) NtSetInformationProcess
31) NtSetInformationThread
32) NtSetInformationWorkerFactory
33) NtShutdownWorkerFactory
34) NtSuspendProcess
35) NtSuspendThread
36) NtTerminateProcess
37) NtTerminateThread
38) NtTestAlert
39) NtUmsThreadYield
40) NtWaitForWorkViaWorkerFactory
41) NtWorkerFactoryWorkerReady
42) NtYieldExecution
43) PCLIENT_ID
44) PCONTEXT
45) PEXCEPTION_RECORD
46) PROCESSINFOCLASS
47) PULONG_PTR
48) PUSER_STACK
49) THREADINFOCLASS

19. PULARGE_INTEGER

1) PULARGE_INTEGER

20. PULONG

1) PULONG

21. PUNICODE_STRING

1) PUNICODE_STRING

22. PVOID_SIZEAFTER

1) PVOID_SIZEAFTER

23. PWSTR

1) PWSTR

24. REGISTRY

1) KEY_INFORMATION_CLASS
2) KEY_VALUE_INFORMATION_CLASS
3) NtCompactKeys
4) NtCompressKey
5) NtCreateKey
6) NtCreateKeyTransacted
7) NtDeleteKey
8) NtDeleteValueKey
9) NtEnumerateKey
10) NtEnumerateValueKey
11) NtFlushKey
12) NtFreezeRegistry
13) NtInitializeRegistry
14) NtLoadKey
15) NtLoadKey2
16) NtLoadKeyEx
17) NtLockRegistryKey
18) NtNotifyChangeKey
19) NtNotifyChangeMultipleKeys
20) NtOpenKey
21) NtOpenKeyEx
22) NtOpenKeyTransacted
23) NtOpenKeyTransactedEx
24) NtQueryKey
25) NtQueryMultipleValueKey
26) NtQueryOpenSubKeys
27) NtQueryOpenSubKeysEx
28) NtQueryValueKey
29) NtRenameKey
30) NtReplaceKey
31) NtRestoreKey
32) NtSaveKey
33) NtSaveKeyEx
34) NtSaveMergedKeys
35) NtSetInformationKey
36) NtSetValueKey
37) NtThawRegistry
38) NtUnloadKey
39) NtUnloadKey2
40) NtUnloadKeyEx

25. SECURITY

1) NtAccessCheck
2) NtAccessCheckAndAuditAlarm
3) NtAccessCheckByType
4) NtAccessCheckByTypeAndAuditAlarm
5) NtAccessCheckByTypeResultListAndAuditAlarmByHandle
6) NtAccessCheckByTypeResultListAndAuditAlarm
7) NtAccessCheckByTypeResultList
8) NtAdjustGroupsToken
9) NtAdjustPrivilegesToken
10) NtDeleteObjectAuditAlarm
11) NtCompareTokens
12) NtCreateToken

25. SECURITY (cont.)

13) PSECURITY_QUALITY_OF_SERVICE
14) NtDuplicateToken
15) NtFilterToken
16) NtImpersonateAnonymousToken
17) NtImpersonateThread
18) NtOpenProcessToken
19) NtOpenProcessTokenEx
20) NtOpenThreadToken
21) NtOpenThreadTokenEx
22) NtPrivilegeCheck
23) NtPrivilegedServiceAuditAlarm
24) NtPrivilegeObjectAuditAlarm
25) NtQueryInformationToken
26) NtQuerySecurityAttributesToken
27) NtQuerySecurityObject
28) NtSetInformationToken
29) NtSetSecurityObject
30) PGENERIC_MAPPING
31) PSECURITY_DESCRIPTOR
32) NtCloseObjectAuditAlarm
33) PTOKEN_PRIVILEGES
34) SECURITY_INFORMATION
35) TOKEN_INFORMATION_CLASS
36) TOKEN_TYPE

26. SYNCHRONIZATION

1) EVENT_INFORMATION_CLASS
2) EVENT_TYPE
3) NtClearEvent
4) NtCreateEvent
5) NtCreateEventPair
6) NtCreateKeyedEvent
7) NtCreateMutant
8) NtCreateSemaphore
9) NtOpenEvent
10) NtOpenEventPair
11) NtOpenKeyedEvent
12) NtOpenMutant
13) NtOpenSemaphore
14) NtPulseEvent
15) NtQueryEvent
16) NtQueryMutant
17) NtQuerySemaphore
18) NtReleaseKeyedEvent
19) NtReleaseMutant
20) NtReleaseSemaphore
21) NtResetEvent
22) NtSetEvent
23) NtSetEventBoostPriority
24) NtSetHighEventPair
25) NtSetHighWaitLowEventPair
26) NtSetHighWaitLowThread
27) NtSetLowEventPair
28) NtSetLowWaitHighEventPair
29) NtSetLowWaitHighThread
30) NtSignalAndWaitForSingleObject
31) NtTraceEvent
32) NtWaitForKeyedEvent
33) NtWaitForMultipleObjects
34) NtWaitForMultipleObjects32
35) NtWaitForSingleObject
36) NtWaitHighEventPair
37) NtWaitLowEventPair
38) WAIT_TYPE

27. TIME

1) NtQuerySystemTime
2) NtQueryTimer
3) NtQueryTimerResolution
4) NtSetSystemTime
5) NtSetTimerResolution

28. TRANSACTION

1) NtClearAllSavepointsTransaction
2) NtClearSavepointTransaction
3) NtCommitComplete
4) NtCommitEnlistment
5) NtCommitTransaction
6) NtCreateEnlistment
7) NtCreateResourceManager
8) NtCreateTransaction
9) NtCreateTransactionManager
10) NtEnumerateTransactionObject
11) NtFreezeTransactions
12) NtGetNotificationResourceManager
13) NtListTransactions
14) NtMarshallTransaction
15) NtOpenEnlistment
16) NtOpenResourceManager
17) NtOpenTransaction
18) NtOpenTransactionManager
19) NtPrepareComplete
20) NtPrepareEnlistment
21) NtPrePrepareComplete
22) NtPrePrepareEnlistment
23) NtPropagationComplete
24) NtPropagationFailed
25) NtPullTransaction
26) NtQueryInformationEnlistment
27) NtQueryInformationResourceManager
28) NtQueryInformationTransaction
29) NtQueryInformationTransactionManager
30) NtReadOnlyEnlistment
31) NtRecoverEnlistment
32) NtRecoverResourceManager
33) NtRecoverTransactionManager
34) NtRegisterProtocolAddressInformation
35) NtRenameTransactionManager
36) NtRollbackComplete
37) NtRollbackEnlistment
38) NtRollbackSavepointTransaction
39) NtRollbackTransaction
40) NtRollforwardTransactionManager
41) NtSavepointComplete
42) NtSavepointTransaction
43) NtSetInformationEnlistment
44) NtSetInformationResourceManager
45) NtSetInformationTransaction
46) NtSetInformationTransactionManager
47) NtSinglePhaseReject
48) NtStartTm
49) NtThawTransactions

29. ULONG

1) ULONG

30. WOW64

1) NtWow64CallFunction64
2) NtWow64CsrAllocateCaptureBuffer
3) NtWow64CsrAllocateMessagePointer
4) NtWow64CsrCaptureMessageBuffer
5) NtWow64CsrCaptureMessageString
6) NtWow64CsrClientCallServer
7) NtWow64CsrClientConnectToServer
8) NtWow64CsrFreeCaptureBuffer
9) NtWow64CsrGetProcessId
10) NtWow64CsrIdentifyAlertableThread
11) NtWow64CsrVerifyRegion
12) NtWow64DebuggerCall
13) NtWow64GetCurrentProcessorNumberEx
14) NtWow64GetNativeSystemInformation
15) NtWow64InterlockedPopEntrySList
16) NtWow64QueryInformationProcess64
17) NtWow64QueryVirtualMemory64
18) NtWow64ReadVirtualMemory64
19) NtWow64WriteVirtualMemory64

Author's Publications

I. Chionis, S.D. Nikolopoulos and I. Polenakis. A Survey on Algorithmic Techniques for Malware Detection *2nd International Symposium on Computing in Informatics and Mathematics* (2013).

Short Vita

Born in Athens in (1990), Iosif Polenakis received his B.Sc. degree in Informatics (2012) from the Department of Informatics of the Ionian University. His B.Sc. thesis was the development and experimental evaluation of a simulator for malware spread in wireless connected devices. He received his M.Sc. degree from the Department of Computer Science and Engineering (2014) of the University of Ioannina and his M.Sc. thesis is on the development of algorithmic techniques for malware detection and classification based on system-call dependency graphs. During his M.Sc. studies he was member of the "Algorithms Engineering Lab" doing research on the application of algorithmic theory on graph-based techniques for malware analysis. His research interests focus mainly on graph-similarity, data-mining, malware analysis, and cryptography.