Algorithmic Techniques for Detection and Classification of Digital Objects

A Dissertation

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Iosif Polenakis

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
June 2019

Dedication

Starting from the early stages of the learning process, this dissertation is dedicated to my grandmother and to my grandfather (he will glorify from above) who taught me the first letters and contributed most to my character. Then, in my beloved family, my mother, my father, my sister, and of course my ”beloved one”, who helped me by all means to my career. I want to dedicate this dissertation to my beloved ”spiritual father”, my good teacher, who is none other than the supervisor of this dissertation, constituting a scientific and educational hero, shaping several aspects of my character in the latest years, making me feel lucky to be his student. Among mixed thoughts springing out of my mind the time these words are written, yes, this dissertation is mainly dedicated to the above people who, as passengers on this trip, patiently supported my effort throughout my stormy days, but also to yourself dear friend, who will go into the process of reading the ’diary’ of this journey.

And once the storm is over,
you won't remember how you made it through,
how you managed to survive.
You won't even be sure, whether the storm is really over.
But one thing is certain.
When you come out of the storm,
you won't be the same person who walked in.
That's what this storm's all about.
`Haruki Murakami, Kafka on the Shore, 2015.`

Acknowledgements

Finishing this dissertation, I would like to begin by thanking my beloved family for its indisputable and unselfish contribution throughout my studies. A ”great thank you” for all the moments we had on this journey.

Then, from the deepest of my heart, I would like to thank my supervisor, Stavros D. Nikolopoulos, Professor of the Department of Computer Science and Engineering of the University of Ioannina, initially for the trust he has shown to me and my goals from the very first moment of our cooperation, making me want to go even farther, and then for all those “lessons” that he offered my the latest years, for his patience and persistence in my academic formation, as well as for all the moral and of intellectual support it gave me throughout the preparation of this thesis.

I would also like to thank the members of the advisory committee, professor Leonidas Palios, and professor Loukas Georgiadis, for their valuable advice and the support they provided me throughout this PhD thesis, as well as the members the examining committee, professor Christos Nomikos, professor Yiannis Stamatiou, professor Aris Pagourtzis and professor Dimitrios Tsolis, for their valuable remarks, which contributed to the improvement of this thesis.

Finally, I would like to thank my colleagues, but mostly friends, Anna, Maria, Achilleas, Akis, Vasilis and Yannis for all the good cooperation we had all these years, the patience and the tolerance they shown the latest years in my character, but also all the beautiful moments we had together against through difficult times.

Table of Contents

List of Figures

List of Tables

Abstract

Iosif Polenakis, Ph.D., Department of Computer Science and Engineering, University of Ioannina, Greece, June 2019.
Algorithmic Techniques for Detection and Classification of Digital Objects.
Advisor: Stavros D. Nikolopoulos, Full Professor.

In this PhD Thesis there have been studied the algorithmic techniques for the detection and classification of digital object. In the area of digital objects, this Thesis focuses mainly on the investigation of designing and proposing algorithmic techniques that detect and classify (in terms of indexing) a specific category of digital objects, the one of software, and more precisely the malicious software, providing finally an integrated algorithmic framework for protection against malicious software.

It is well known that malicious software consists a security threat of major importance. Especially the last years, where almost every device supports networked operations, several malicious attacks have been deployed targeting on the infringement of Confidentiality, Integrity and Availability of data stored into information systems or any other computing device. Hence, this thesis mainly focuses on the design and the development of efficient graph-based algorithmic techniques that detect malicious software samples and further classify them into known malware families, while on the other hand, the proposal of graph-based strategies for early warning, effectively prevent the pandemic spread of malicious software between interconnected mobile devices. The structure of the thesis is developed over two axes, namely, the design and development of protection techniques against the malicious software, regarding the detection and classification of malicious samples, and the development of graph-based techniques for pandemic prevention, regarding the definition of the maximum permit table time required for a countermeasure to suppress malware's spread.

Malicious authors, in order to avoid traditional detection methods, have developed highly sophisticated practices focusing on mutating their produced malicious samples, incorporating mutation engines that mutate the structure of the generated malicious samples (i.e., polymorphism and metamorphism). On the first development axis of this thesis, the research focuses on the design and the proposal of a mutation-tolerant graph-based representation of malicious software sample's behavior (behavioral graph) resulted from System-call Dependency Graphs, or, for short ScDG, a Directed Acyclic Graph produced through Dynamic Taint Analysis of the executed sample. So, in the first state we propose the Group Relation Graph, or, for short GrG, a Directed Weighted Graph that is an abstraction of ScDG resulting after grouping disjoint vertices of it, utilizing the property that system-calls can be merged into groups based on their similar functionality. Further, we extent this approach by proposing the Coverage Graph, or, for short CvG, where we investigating the dominating relations among the vertices of GrGs regarding the vertex weight and degree. Additionally, extending the potentials of the above graph-based representations, we also propose the Temporal Graphs, that actually depict the structural evolution of the previously proposed graphs (i.e., GrG and CvG) by depicting their structures through instances captured over specific periods. Among others, we propose a set of similarity metrics that utilize quantitative, relational and qualitative characteristics of the above graph-based representations of malicious software sample's behavior, utilizing them in order to experimentally evaluate the detection and classification potentials of our model.

Moreover, since the usage of mobile devices exhibits a wide spread, dependently of te adoption of IoT, throughout this thesis, it has also been studied the development of graph-based algorithmic techniques that would integrate the overall algorithmic framework for protection against malicious software by investigating graph-based strategies for suppressing and finally avoiding potential pandemics caused by malware's spread. More precisely, we propose a set of graph-based techniques for modeling the topology of towns-planning, the node mobility patterns as also the propagation behavior, incorporating them to develop an algorithmic technique that defines the maximum permitted time required by a counter- measure to take effect removing the malware from an infected device (i.e., response time) in order to finally the pandemic spread. Finally, the precision of the proposed approach is tested throughout a series of repetitive (Monte Carlo) series of experiments of various epidemic models and set of factors that affect the malware's spread.

Εκτεταμενη Περιληψη

Ιωσήφ Πολενάκης, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2019.
Αλγοριθμικές Τεχνικές Ανίχνευσης και Κατάταξης Ψηφιακών Αντικειμένων.
Επιβλέπων: Σταύρος Δ. Νικολόπουλος, Καθηγητής.

Σε αυτή τη διδακτορική διατριβή διερευνήθηκαν οι αλγοριθμικές τεχνικές για την ανίχνευση και τη κατάταξη ψηφιακών αντικειμένων. Στον τομέα των ψηφιακών αντικειμένων, η παρούσα εργασία επικεντρώνεται κυρίως στη μελέτη του σχεδιασμού και της περαιτέρω ανάπτυξης αλγοριθμικών τεχνικών ανίχνευσης και κατάταξης μιας συγκεκριμένης κατηγορίας ψηφιακών αντικειμένων, αυτής του λογισμικού και πιο συγκεκριμένα του κακόβουλου λογισμικού, δημιουργώντας εν τέλει ένα ολοκληρωμένο αλγοριθμικό πλαίσιο για την προστασία ενάντια σε αυτό.
Είναι γνωστό ότι το κακόβουλο λογισμικό αποτελεί μία από τις σημαντικότερες απειλές για την ασφάλεια. Ιδίως τα τελευταία χρόνια, λόγω της εξέλιξης στη διασυνδεσιμότητα των υπολογιστικών συσκευών, αναπτύχθηκαν διάφορες κακόβουλες επιθέσεις με στόχο την παραβίαση της εμπιστευτικότητας, της ακεραιότητας και της διαθεσιμότητας των δεδομένων που εδράζονται τόσο στην ευρεία έκταση των πληροφοριακών συστημάτων, όσο και σε αυτόνομες υπολογιστικές συσκευές. Με αυτή την αφορμή, η συγκεκριμένη διατριβή προσανατολίζεται στον τομέα της Ασφάλειας Πληροφοριακών Συστημάτων, έχοντας ως κύριο μέλημα, αφενός το σχεδιασμό και την ανάπτυξη αποτελεσματικών γραφοθεωρητικών αλγοριθμικών τεχνικών, τα οποία αρχικά ανιχνεύουν κακόβουλα λογισμικά, ταξινομώντας τα εν συνεχεία σε γνωστές οικογένειες κακόβουλων λογισμικών, ενώ αφετέρου, βασισμένες στις αρχές της έγκυρης πρόληψης, αποτρέπουν αποτελεσματικά τα φαινόμενα πανδημίας από την εξάπλωση του κακόβουλου λογισμικού μεταξύ διασυνδεδεμένων φορητών συσκευών. Συγκεκριμένα, η παρούσα διατριβή αναπτύσσεται πάνω σε δύο άξονες, στον σχεδιασμό και την ανάπτυξη αλγοριθμικών τεχνικών προστασίας από το

κακόβουλο λογισμικό, αναφορικά με την ανίχνευση και τη περαιτέρω κατάταξη κακόβουλων λογισμικών και εν συνεχεία την ανάπτυξη γραφοθεωρητικών τεχνικών για την πρόληψη πανδημικών φαινομένων ορίζοντας το μέγιστο επιτρεπόμενο χρονικό όριο για την εφαρμογή των μέτρων προστασίας με στόχο την καταστολή της εξάπλωσης του κακόβουλου λογισμικού.
Οι δημιουργοί των κακόβουλων λογισμικών, προκειμένου να αποφύγουν τις καθιερωμένες μεθόδους ανίχνευσης, έχουν αναπτύξει ευφυείς τεχνικές που εστιάζουν στη μετάλλαξη των παραγόμενων κακόβουλων λογισμικών, ενσωματώνοντας μηχανισμούς μετάλλαξης που στόχο έχουν να τροποποιήσουν ριζικά τη δομή των παραγόμενων δειγμάτων. Ως εκ τούτου, στον πρώτο άξονα, η έρευνα επικεντρώνεται στο σχεδιασμό και την πρόταση μιας αναπαράστασης μέσω γραφήματος της συμπεριφοράς του δείγματος κακόβουλου λογισμικού (συμπεριφοριστικό γράφημα) ανθεκτικής σε μεταλλάξεις, η οποία προκύπτει από τα Γραφήματα Εξάρτησης Κλήσεων Συναρτήσεων Συστήματος (κατευθυνόμενα άκυκλα γραφήματα), τα οποία κατασκευάζονται αντλώντας πληροφορία από την εκτέλεση δυναμικής ανάλυσης του εκτελεσθέντος κακόβουλου λογισμικού. Έτσι, σε πρώτο επίπεδο, προτείνουμε το Γράφημα Συσχετίσεων Ομάδων (κατευθυνόμενο έμβαρο γράφημα) το οποίο προκύπτει έπειτα από ομαδοποίηση των κόμβων του γραφήματος Εξάρτησης Κλήσεων Συναρτήσεων Συστήματος, αξιοποιώντας την ιδιότητα ότι οι κλήσεις συναρτήσεων συστήματος μπορούν να συγχωνευθούν σε ομάδες ανάλογα με την ομοειδή λειτουργικότητά τους. Επιπρόσθετα, επεκτείνουμε αυτή την προσέγγιση προτείνοντας το Γράφημα Κάλυψης, όπου διερευνούμε τις “σχέσεις κυριαρχίας” μεταξύ των κόμβων του γραφήματος Συσχετίσεων Ομάδων, αναφορικά με το βάρος και το βαθμό αυτών. Επιπλέον, επεκτείνοντας τις δυνατότητες των παραπάνω γραφημάτων προτείνουμε επίσης τα Χρονικά Μεταβαλλόμενα Γραφήματα, τα οποία απεικονίζουν τη δομική εξέλιξη των προτεινόμενων γραφημάτων (δηλ. Γραφήματα Συσχετίσεων Ομάδων και Γραφήματα Κάλυψης) απεικονίζοντας την εξέλιξη στη δομή τους μέσω στιγμιότυπων αυτών, τα οποία καταγράφονται ανα συγκεκριμένες περιόδους. Μεταξύ άλλων, προτείνουμε ένα σύνολο μετρικών ομοιότητας, όπου αξιοποιούνται τα ποσοτικά, σχεσιακά και ποιοτικά χαρακτηριστικά των παραπάνω γραφημάτων αναφορικά με τη συμπεριφορά των κακόβουλων λογισμικών, αξιοποιώντας αυτές τις μετρικές για τη μετέπειτα αποτίμηση των δυνατοτήτων ανίχνευσης και κατάταξης των προταθέντων μοντέλων.

Επιπλέον, δεδομένου ότι η χρήση των φορητών συσκευών παρουσιάζει ευρεία

εξάπλωση, κατά την εκπόνηση της παρούσας διατριβής μελετήθηκε η ανάπτυξη αλγοριθμικών τεχνικών βασισμένων σε γραφήματα, οι οποίες θα ολοκληρώνουν το ευρύτερο αλγοριθμικό πλαίσιο προστασίας ενάντια στο κακόβουλο λογισμικό, με τη διερεύνηση στρατηγικών βασισμένων σε γραφήματα για την καταστολή και περαιτέρω αποφυγή εν δυνάμει πανδημικών φαινομένων που θα προκύψουν από την εξάπλωση του κακόβουλου λογισμικού. Πιο συγκεκριμένα, προτείνουμε μια σειρά τεχνικών για τη μοντελοποίηση της τοπολογίας του πολεοδομικού σχεδιασμού, των μοτίβων κίνησης των φορητών συσκευών καθώς επίσης και της συμπεριφοράς μετάδοσης (αναφορικά με το ακολουθούμενο επιδημιολογικό μοντέλο), συντονίζοντας τα μοντέλα αυτά στην πλαισίωση μιας αλγοριθμικής τεχνικής που θα καθορίζει τον μέγιστο επιτρεπόμενο χρονικό όριο που απαιτείται από ένα αντίμετρο προστασίας προκειμένου να απομακρυνθεί το κακόβουλο λογισμικό από μια μολυσμένη συσκευή (χρόνος απόκρισης), ώστε τελικά να αποφευχθεί η πανδημία. Τέλος, η ευρύτερη απόδοση της προτεινόμενης προσέγγισης παρουσιάζεται μέσα από μια σειρά επαναλαμβανόμενων πειραμάτων (Monte Carlo) ακολουθώντας διαφορετικά επιδημιολογικά μοντέλα, λαμβάνοντας παράλληλα υπόψη και ένα σύνολο παραγόντων που επηρεάζουν την εξάπλωση του κακόβουλου λογισμικού.

Chapter 1

Introduction

1.1 Modern Security Threats

The increasing security threats on the protection of privacy, integrity and confidentiality of systems as also of the data stored in them constitute the key incentives for research and thorough study on information system security. Security of information systems is one of the most important issues of concern in maintaining the smooth and persistent operation of IT. This research proposal is developed in the field of protection against malicious software and the prevention of its spread, which consists the dominant tactic of cyber-attacks. Therefore, the basic aim of the proposal is the in-depth and multi-level study on the protection against malicious software as also the prevention of its spread by proposing effective graph-based algorithmic techniques which ensure the protection of privacy, integrity and confidentiality of the systems.

Approaching the problem, the methodology we follow develops an algorithmic framework consisting of two axes: protection against malicious software and prevention from its spreading between mobile devices. So, at the first level, we study, design and finally develop protection systems that implement a set of algorithmic methods of detecting and classifying malware using watermarking techniques as well as other graph-based approaches. On the second level, based on known epidemiological models represented by graphs depicting snapshots of the networks that are dynamically formed between the mobile devices, we use structural information about the location of the nodes and the topology of the spatial representation, developing algorithmic techniques to suppress the spread of malware between mobile devices preventing pandemics. In order to achieve this goal, the ongoing research is developed on two main axes, which are initially on the development of algorithmic techniques for protection against malicious software and then on the development of algorithmic techniques to prevent its spread between mobile devices as to avoid pandemics.

In the concept of a clearly critical threat on the security of IT operation, we are called upon to investigate, recommend and implement techniques that, approaching the problem algorithmically, will provide protection against malicious software and also suppress to its spread between computing devices.

One of the most important challenges in detecting and then classifying malicious software is the resilience of each technique against its mutations (strain variation), with significant success rates being occupied by the so-called behavioral techniques. In this thesis there have been studied and proposed such malware detection and classification techniques [7], utilizing System-call Dependency Graphs as representations of its behavior through specific abstractions of these graphs in mutation-resistant graphs.

It is widely accepted that prevention is an invaluable tactic, and therefore it is clearly intended to be applied in the case of suppressing the spread of malicious software between mobile computing devices. Having already studied the phenomenon, from the aspect of the influence of the counter-measure's response time to avoid pandemic spread, it is estimated that further study at the level of nodes of the network with the greatest influence on the spread of malicious software (critical nodes) would contribute significantly the research level of the field. Therefore, in the second part of the proposed algorithmic framework, we aim to develop innovative algorithmic techniques that, as an evolution of the already proposed ones, will initiate the launch of graph-based strategies targeting the immunization of critical nodes of the network, utilizing the position of computing devices in the dynamic network.

1.2 Malicious Software

Malware or malicious software is a software type intended to cause harm to end point computers, systems or networks [8]. In this work we design and propose a graph based model that develops an algorithmic technic for malware detection and classification. Our method is applied on unknown software samples in order to detect whether they are malicious or not, and further classify them to one of a set of known malware families (i.e., set of malicious mawlare samples with similar functionality), as they have been developed by various antivirus software vendors.

On the contrary part of our scientific field, malware authors, have developed and deployed various techniques in order to avoid the traditional byte-level signature based detection methods. Since such detection methods appear to be significantly fragile against even the least (i.e., bit-level) mutation of the initial subject (i.e., ancestor malware sample), they mutate their software products (malware) creating structurally different but functionally similar copies of them. Except from the mutation methods that leverage one, or more, levels of encryption, there also exist more advanced mutation methods. Some of the most applicable malware mutations are the oligomorphism which is achieved through obfuscation techniques, the polymorphism where the code is modified through encryption techniques and the metamorphism, in which multiple structurally different copies of a malware sample are generated.

More precisely an *oligomorphic* or *semi-polymorphic* malware, is a specific category of obfuscated malware disposing an encryption/decryption module for multi-layer encryption in order to avoid decryption body detection. On the other hand, a *polymorphic* malware can create an endless number of new decryptors that use different encryption methods to encrypt the body of the malware [9]. As referred in [10], the main principal is to modify the appearance of the code constantly across the copies. Finally, a *metamorphic* malware changes its structure while keeps its functionality each time it replicates itself [11]. 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 a constant signature for virus scanning [11].

Hence, while the main functionality of a malware sample remains immutable during its mutations, malware samples can be merged into groups of malware samples with common functionality, the so called malware families. So, in this work we developed an algorithmic technic that not only detects if a program is malicious or not, but additionally, given a malicious software it can decide the malware family that it belongs to.

1.3 Defense Line Countermeasures

Since malicious software poses a major threat, several protection approaches have been proposed and implemented in order to eliminate such threats. The main corpus of the defense line is mainly developed over three axes, namely malware analysis, malware detection and malware classification:

- **Malware Analysis.** Malware analysis [12] is the process of determining the purpose and the functionality or, abstractly, the behavior of a given malicious code. Such a process is a necessary prerequisite in order to develop efficient and effective detection and also classification methods, and is mainly divided into two main categories, namely *Static* and *Dynamic* analysis [8].

- **Malware Detection.** The process of *malware detection* describes the method which focuses on determining whether a given program P is malicious or benign according to an *a priori knowledge* [13, 14, 15]. Specifically with the term *a priori knowledge* we are referred to something that is known to be malicious or a characteristic that owned by something that is malicious, at a given time. However, an efficient malware detection is strongly related to *malware analysis*, during which, the analyst collects all the required information.

- **Malware Classification.** The term *malware classification* refers to the process of determining the malware family to which a particular malware sample, let M belongs to. Malware classification is a quite important procedure, since the indexing of malware samples into families provides the ability to generalize detection signatures from sample level to family level. Through the indexing of a malware sample to a malware family, the construction of a new sample-specific signature is omitted, since the sample can be detected by the signature of its family.

- ☐ **Prevention.** One of the most important procedures incorporated to ensure the suppression of malware's spread is the deployment of efficient prevention methods or strategies that focus on the limitation of malware's propagation over their early stages. Most of such approaches include the deployment of an underlying compartmental epidemic model as also approaches that implement "early warning" or immunization strategies.

1.3.1 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.

A. 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 [16, 8, 17, 18, 12, 19]. 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 [16], 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 [12] and in most of the cases is faster that the dynamic one.

A.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 [8] 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 [8] 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 [17] 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 [17] 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 [8], 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 [8].

- **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 [17]. 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 [8]. 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 [8, 17].

- **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* [8].

  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 [8], 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 [8], 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 [17]. 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.

A.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 [8] 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 [8].

B. 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 [12]. 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 [16] 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 [17] 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 [20].

B.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 [21] 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 [16]. In order to analyze a program's behavior it is needed to intercept in some fashion between function calls, a process called *function hooking*[21]. Consecutively a dummy function that is responsible for that procedure is called *hooking function* [16]. 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* [16]. 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 [16, 8].
  - 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 [16, 8].
  - Windows Native API: Finally, Windows Native API resides between the system calls and the Windows API. As referred in [16]. 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 [16], 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 [16], 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[16]. 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 [16], 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 [5]. 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 [5, 22], 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.

B.2. Dynamic Analysis Tools

In this section we present some tools widely used in dynamic malware analysis as they described in [8]. 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.

1.3.2 Malware Detection

As we mentioned previously, malicious software samples are intended to compromise the privacy, the confidentiality or the integrity of a system, of data or any other cyber-source constituting hence an intrusion. To this end, Intrusion Detection Systems, or, for short IDS, are deployed in order to monitor the execution of applications, the traffic of networks or whole systems, aiming on spotting malicious activity patterns [23]. The system supervision through an IDS can be performed through the application of *malware detection* techniques, that reference file comparisons against signatures of malicious software [24], behavior monitoring of malicious patterns and system supervision [23]. However, the increasing birth-rate of new or mutated malware samples has raised the need for efficient and elaborated malware detection techniques that can effectively detect new malware strains in reasonable amounts of time. The detection approaches are strongly connected to the features set provided through the previous stage of malware analysis, and are distinguished to static and dynamic features, respectively. Static features may include, statistical analysis on n-grams or opcodes, properties of control flow graphs, while dynamic features are obtained the execution time of a program and concern its general behavior (i.e., interaction with the host-environment - O.S.), access events or any other interconnection patterns [25].

Malware detection as a general term is the process of determining, if a given program is malicious or benign [14, 15, 26, 13], 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 [13].

Malware detection is implemented through the utilization of a series of specific malware detection techniques [26]. In the latest approaches, malware detection is implemented with two approaches, *signature-based detection* and *behaviour* or *anomaly-based detection*.

Malware detection methods can be categorized into signature-based detection and anomaly or behavior-based detection [27, 28, 29, 30, 31, 32, 33, 34], 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.

A. Signature-based Malware Detection

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* [18]. If malware detector find such signature then the program under inspection is probably 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 [13].

Signature-based malware detection is one of the major techniques deployed by antivirus software products due to its time efficacy that provides real-time protection against malicious threats [35]. A byte-level signature is a sequence (i.e., pattern) of bytes used to identify each newly discovered malware, using a scanning scheme of exact correlation and a repository of signatures in order to detect malicious software samples [28]. A signature may represent a byte-code sequence, a binary assembly instruction, an imported Dynamic Link Library (DLL), or function and system calls. [29, 30]. Novel malware detection approaches using machine learning can be deployed through two methods, namely, assembly features and binary features [27]. However, signature-based detection techniques can easily be evaded through code obfuscation techniques that even the least modification on the code sequence would lead to a completely different byte-sequence [28]. A major characteristic of signature-based malware detection is the exhibited precision so through object scanning utilizing efficient meta-heuristic algorithms as in the uniqueness signature creation. This characteristic regarding its precision may turn to a drawback, since such methods can not detect obfuscated or mutated (e.g., polymorphic) malware samples, as their signature does not match the stored one [27].

As referred in [13], 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 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 [14], 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 [8]. 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 [26], 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 malware, 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.

B. Behavior-based Malware Detection

Another approach deployed for malware, gaining remarkable research interest during the last yeas is behavioral detection, or more formally, behavior-based malware detection [36]. Behavior-based malware detection mainly focuses on capturing the interaction (in terms of interconnection, relations or dependencies between system-elements i.e., system-calls or API calls) between the executed software and the system (i.e., Operating System) [5, 37, 14, 6, 12, 30, 38, 39, 40]. From an abstract machine learning aspect, the behavior-based systems are trained over a learning phase with behaviors exhibited during the execution of known malicious software samples, while in the monitoring phase the trained behavior-based system decides if an unknown software sample is malicious or not [28]. Behavior-based detection systems as expected require the execution of the software sample in order to extract dynamically (see, Dynamic Malware Analysis) the exhibited behaviors. In order for these dynamic systems to perform the mining of the specified behaviors they utilize software and hardware virtualization technologies, alongside with imitation conditions [27], providing the test sample with an environment as close to reality in order to evade the sandbox-detection mechanisms deployed occasionally by malicious software samples, and letting them exhibit their intentions. Despite the fact that such techniques deploy quite elaborate algorithms on their implementations, the incident that malware families tend to evolve in order to avoid detection [29], results to the need of the development of more elastic and mutation resilient techniques like the one we propose in this work.

As referred in [26] 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 [41]. 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.

B.1 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* [26]. 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 part 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 [41]. 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 [26]. 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.

B.2 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 positive 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 [26]. In this type of behavioral detection the whole process relies on, either manually written or through static analysis, application-specific models [41]. So, the main goal is the development of a *rule set* specifying the valid behavior that should be exhibited by any running application [26]. 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.

1.3.3 Malware Classification

Next, 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.

Particularly, malware authors, in order to avoid traditional detection methods, produce new (mostly mutated) malware samples rapidly, utilizing existing ones in order for the new strains to preserve the functionality inherited from their ancestors. As referred in [30] mutated malware samples are generated from existing ones utilizing automated techniques [42, 43] or integrated tools, generating new samples from libraries and code parts from code exchange networks.

Through the literature, the term *malware classification* has been confused several times with *malware detection*. Distinguishing precisely these two procedures, it can be stated that malware detection is a *binary classification*, where a a set of unknown samples is classified against a collection of malware and goodware samples, while malware classification is a *multinomial classification* on whether an already detected malware sample belongs to a particular family or type [44]. As described in [45], malicious software samples that belong to the same malware family tend to exhibit similar behavioral and structural profiles. Additionally, malware classification augments the analysis of new, or mutated, malicious samples where their signatures have not been constructed yet [46].

A. Philogeny

Another field of malware analysis applied in malware classification is *malware phylogeny*[47], which aims on inferring evolutionary relationships between instances of families. The major profit from creating a phylogeny model is the fact that newly developed elaborated detection systems that deploying such techniques can detect that a sample that has not been previously seen can be related to a malware family, when analyzed along an evolution path [48]. Throughout this process the main target is to reveal similarities and relations among a set of specific malware samples coexist and are exhibited by all the members of the set (i.e., malware family) [49], distinguishing its type or family. Such approaches can be utilized to identify evolution trends in over a set of malware samples [30], constituting hence valuable tool for more generalized signatures or, in general, more elaborated detection-techniques. The models applied on phylogeny, using mostly phylogenetic networks, model evolutionary relations among malware families, describing temporal ordering among samples, defining ancestor-descendent relations, as also relationships between families, augmenting hence malware classification [] and unveiling evolutionary trends [50].

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 (e.g., 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 [51, 52], 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.

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* [51]. A quite convenient representation of malware's evolution could be a tree-like one as a *dendrogram* [53] (see Figure 4.1) where malware samples have been clustered according to a technique that detects commonalities between specimens.

B. 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 [2]. 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 [53].

C. 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 [3, 38].

D. Behavior-Based Classification Methods

As we referred previously, malware samples that belong to the same family 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.

D.1 Malware Classification using Function Call Graphs

In [3] 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.

D.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].

Next we present an overview of the fundamental elements that rule the concept of malware mutation procedures, and how the properties of the whole concept could be leveraged in order to develop robust and global detection techniques that based on the procedure of malware classification would be able to efficiently be applied to detect any variation (i.e., mutated strain) of a malware sample.

- Mutations: As it was discussed previously, malware authors in order to avoid traditional signature based detection techniques tend to mutate their produced malware samples through various mutations techniques (i.e., polymorphic or metamorphic engines etc.). The produced mutated malware samples actually are generated through a technique that is deployed over an initial (ancestor) malware sample that inherits to the new strain its main functionality but with a differentiated structure regarding mostly its implementation.

- Phylogeny: The tree structure generated over several evolution on various initial malware samples is called *Philogenetic Tree* and provides valuable information concerning the development of generalized malware detection techniques that would be able to leverage the shared functional commonalities of malware strains that have been generated sharing a common ancestor malware sample.

- Family Indexing: Investigating the previous approach, it easily follows that, since several malware strains that have been produced as evolution of a previous seen, and hence already known, malware specimen, are produced based on this ancestor, they can construct a group of functionally similar siblings (i.e., malware family).

- Generalization Models: Leveraging the properties exhibited so far, the approach of developing detection techniques that utilize functional commonalities shared among the malware strains produced from a common ancestor malware strain, would result to techniques that actually would detect any malware strain produced from any previously known malware sample, degrading hence the ability of malware authors to avoid detection through the deployed mutations techniques.

1.3.4 Epidemic Models and Digital Epidemiology

Epidemic models can be applied to any network structures to describe the propagation of a disease despite of its type (i.e., biological virus or computer virus) between a set of entities. The overall propagation can be described as a branching process, e.g., a tree that its root is the initial infected population and every level contains child nodes representing the population infected by the nodes of the previous level.

Compartmental Models

Briefly speaking, such epidemic models describe the nodes - entities by a set of potential states or conditions they can go through the course of the epidemic, namely `Susceptible, Infected, Repaired, Removed, Immune`. In the `Susceptible` state a node is potentially vulnerable to a disease, while when the node gets infected (probably by its neighbors) then it goes to `Infected` state. On the other hand, depending on the modeled cases, if the disease is destructive for its host then after a period of time the infected node goes to `Removed` state, while if a cure exist and is been applied to an infected node then after a period of time (throughout this paper we shall call it sanitize-time) the node goes to `Repaired` state, where, depending again on the modeling demands, it can be either `Immune` or not. Next, we briefly present various epidemic models that can be deployed according to the needs of the simulated problem.

A. Epidemic Model SI

It the most trivial model containing only two states (`Susceptible, Infected`). Once a node is susceptible and gets infected (and hence infectious), then it remains forever in this state. The following epidemic models considered as variations of the SI model [54, 55, 56, 57].

B. Epidemic Model SIRp

In this model an infected node can be repaired in some fashion [54, 55, 57]. To this point, it is worth noting that a node repair may provide immunization to the host against the disease or not. Depending on this fact, the following specifications arise as special cases of SIRp model:

B.1. Epidemic Model SII

The SII model (last I stands for Immune) results as a solution when we need to formally describe the propagation of a disease where there exists a cure that immunizes/sanitizes [58] the infected hosts after their treat [57, 59, 60].

B.2. Epidemic Model SIS

On the other hand, the SIS model (last S stands for Susceptible) is suitable for the cases where even though exists a cure for the infected node, it still stays susceptible on getting the disease again [57]. Another extension of the SIS epidemic model is the **SIRS Epidemic Model** where an infected host may get cured by a repair and get susceptible again. More precisely, the SIS model is also referred as SIRS where the 'R' stands for `Repaired` (i.e. 'Rp' in our case). A further specification may be appeared extending SIS model depending on the case and the demands of the situation under modeling.

C. Other Epidemic Models

Other epidemic models may contains less common states (i.e., compartments), e.g., Removed or Exposed like the ones presented below.

C.1. Epidemic Model SEIS

Finally, if the modeled disease is destructive for the infected host, i.e., no cure exists, then SIRm model (last Rm stands for Removed) is suitable for application in such case to model the epidemic.

C.2. Epidemic Model SEIS

The SEIS model introduces a new state (i.e. `Exposed`)takeing into account the latent period of a disease where a node may be exposed to the disease by, for example, a close contact with an infected node. In this model, any immunity has been left to an infected node leaving an infected node to be susceptible again in the time after the infection.

C.3. Epidemic Model SEIR

However, similarly to the case of SIRp model, the SEIR epidemic model formally describes the propagation of a disease where there exists a cure that immunizes or simply repairs an infected host, once has been firstly exposed and then infected by the disease.

Control of Malware Spread and Pandemic Prevention

In the wider area of epidemic models there are several properties that rule the propagation of both a biological or digital threat regarding the Susceptible population over the compartmental model that describes the behavioral characteristics of an epidemic. Some of them, exhibiting a greater impact on affecting the spread, are the so called “birth rate” which describes the rate on which the Susceptible population gets Infected, as also the initial Infected population.

Combining these two properties, and trying to manipulate the factors that affect the spread on a greater magnitude, it follows that the propagation of a spreading threat (in this case a malicious software that spreads between mobile devices) could be avoided if there would exist a method, or an approach in general, that would give the ability to manipulate in some fashion so the rate on which the Susceptible population turns to Infected as also the initial Infected population. In both cases, the critical situation is indicated at the point where the Infected population gets an extremely high exponential grow, the so called “grow-level”, which leads to either a faster grow of the Infected population in the next time as also, if it is located chronologically at the start of the epidemic, to an increase of the initial Infected population, which rationally also increase the rate of infections.

The main goal on attempting to prevent pandemic situations is to investigate, propose and finally implement graph based strategies that will be able, by leveraging the aforementioned properties, to prevent pandemic outbreaks, specifically for the case of digital threats, e.g., a malicious software spreading between proximal mobile devices. Hence, concentrating over the reduction of the initial Infected population it is rational to expect that early warning, or in other words a quick response time, would significantly reduce the initial Infected population, or at least to delay the appearance of "grow-level", and through this achievement, would significantly affect malware's spread leading finally to pandemic prevention. Specifically, in this thesis, the effect of response time in terms of the maximum permitted period of time need by a counter-measure (i.e., AntiVirus or removal tool) to take effect by sanitizing the Infected population (i.e., mobile devices) is investigated and a graph based model for this purpose is proposed, implemented and experimentally tested.

1.4 Related Work

In the following, there are presented the related works regarding the proposed approaches on malware detection and malware classification as also the literature relevant to malware pandemics.

1.4.1 Related Work on Malware Detection

Next there are presented several graph-based and non-graph based malware detection techniques proposed through the literature over the last years and have consisted the base of the theoretical background on this work regarding the malware classification procedure.

Graph-based Malware Detection Techniques

Kolbitch *et al.* [4] proposed an effective and efficient approach for malware detection, based on behavioral graph matching by detecting string matches in system-call sequences, that is able to substitute the traditional anti-virus system at the end hosts. The main drawback of this approach is the fact that although no false-positives where exhibited, their detection rates are too low compared with other approaches.

Luh and Tavolato [61] presented one more detection algorithm based on behavioral graphs that distinguishes malicious from benign programs by grading the sample based on reports generated from monitoring tools. While the produced false-positives are very close to ours, the corresponding detection ratio is even lower.

Babic *et al.* [5] 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.

Christodorescu *et al.* [37, 14] 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 [6], Fredrikson *et al.* proposed an automatic technique for extracting optimally discriminative behavioral specifications, based on graph mining and concept analysis, that have a low false positive rate and at the same time are general enough, when used by a behavior based malware detector, to efficiently distinguish malicious from benign programs.

In [62], Makandar and Patrot focus on detection and classification of the Trojan viruses using image processing techniques. In their proposed algorithm Gabor wavelet is used for key of feature extraction method and their experimental results are analyzed compared with two classifications such as KNN and SVM.

In [63], Hassen and Chan investigate a linear time function call graph (FCG) vector representation based on function clustering that has significant performance gains in addition to improved classification accuracy. They also show how this representation can enable using graph features together with other non-graph features.

Recently, Sikora and Zelinka [64] investigate how behavior of malicious software can be connected with evolution and visualization of its spreading as the network. Their approach is based on hypothetical swarm virus and its dynamics of spread in PC and they show that its dynamics can be then modeled as the network structure and thus likely controlled and stopped, as their experiments suggest.

Later, Souri and Hosseini [27] present a systematic and detailed survey of the malware detection mechanisms using data mining techniques. Additionally, it classifies the malware detection approaches in two main categories including signature-based methods and behavior-based detection.

Based on the dependency graphs of malware samples, Ding et al. [65] propose an algorithm to extract the common behavior graph for each known malware, which is used to represent the behavioral features of a malware family. In addition, a graph matching algorithm that is based on the maximum weight subgraph is used to detect malicious code.

In [66], Mukesh et al. propose a machine learning based architecture to distinguish existing and recently developing malware by utilizing network and transport layer traffic features.

Non Graph-based Malware Detection Techniques

In malware detection, there have been proposed similar models utilizing different non graph-based techniques like the one proposed by Alazab *et al.* [13], who developed a fully automated system that disassembles and extracts API-call features from executables and then, 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.

In [67], Ye *et al.* described an integrated system for malware detection based on API-sequences. This is also a different model from ours since the detection process is based on matching the API-sequences on OOA rules (i.e., Objective-Oriented Association) in order to decide the maliciousness or not of a test program.

In [67], Ye *et al.* described an integrated system for malware detection based on API-sequences. This is also a different model from ours since the detection process is based on matching the API-sequences on OOA rules (i.e., Objective-Oriented Association) in order to decide the maliciousness or not of a test program.

Finally, an important work of Christodorescu *et al.*, presented in [14], proposes a malware detection algorithm, called $\mathcal{A}_{MD}$, based on instruction semantics. More precisely, templates of control flow graphs are built in order to demand their satisfiability when a program is malicious. Although their detection model exhibits better results than the ones produced by our model, since it exhibits 0 false-positives, it is a model based on static analysis and hence it would not be fair to compare two methods that operate on different objects.

1.4.2 Related Work on Malware Classification

Next there are presented several graph-based and non-graph based malware classification techniques proposed through the literature over the last years and have consisted the base of the theoretical background on this work regarding the malware classification procedure.

Graph-based Malware Classification Techniques

In [38] Bayer *et al.*, propose a scalable clustering approach to identify and group malware samples that exhibit similar behavior, serving as input to an efficient clustering algorithm *profiles* that characterize programs activity in more abstract terms. 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. However, the model proposed in [38] mainly aims on clustering malware samples rather that classifying unknown ones to known malware families, that is a slightly different process.

Hu *et al.* in [68], design implement and evaluate the Symantec's Malware Indexing Tree (SMIT), that classifies malwares based on their function call graphs using k nearest-neighbor search. While, as referred in [68], their success rate reaches the 91.3%, it is worth mentioning that this classification rate refers in the case where the actual labeling of test samples family in included in the k nearest families resulted by the model. Hence, since our model returns only one dominant family we compare our results (i.e., 83.47%) with the results referred in [68] as Dominant Family Rate (i.e., 69.9%), that is defined as the percentage where the most prevalent family among k returned nearest neighbors is also the family to which the query malware belongs.

A model for malware classification utilizing discriminative behavior specifications extracted by the samples is presented by Rieck *et al.* in [39]. Specifically, by monitoring malware samples in sandbox, they collect behaviors, and based on a corpus of malware labeled by an anti-virus scanner a malware behavior classifier is trained using learning techniques. Finally, discriminative features of the behavior models are ranked for explanation of classification decisions. To this point we ought to mention that, despite the fact that their classification results for known malware samples are almost 5% higher that ours, we recall that, as in [69] their experiments are performed using 14 malware families, where the impact of philogeny among different malware families is decreased the less different malware families in the training are.

In [42] Tian *et al.* present a scalable method of classifying Trojans based only on the lengths of their functions. The results achieved by the proposed technique indicate that function length may play a significant role in classifying malware, and combined with other features, may result in a fast, inexpensive and scalable method of malware classification. However, while their results are comparable to our model's, the main difference is the fact that in [42] the model has been evaluated using only Trojans.

In [63], Hassen and Chan investigate a linear time function call graph (FCG) vector representation based on function clustering that has significant performance gains in addition to improved classification accuracy. They also show how this representation can enable using graph features together with other non-graph features. Recently, Sikora and Zelinka [64] investigate how behavior of malicious software can be connected with evolution and visualization of its spreading as the network. Their approach is based on hypothetical swarm virus and its dynamics of spread in PC and they show that its dynamics can be then modeled as the network structure and thus likely controlled and stopped, as their experiments suggest. Later, Souri and Hosseini [27] present a systematic and detailed survey of the malware detection mechanisms using data mining techniques. Additionally, it classifies the malware detection approaches in two main categories including signature-based methods and behavior-based detection.

Based on the dependency graphs of malware samples, Ding et al. [65] propose an algorithm to extract the common behavior graph for each known malware, which is used to represent the behavioral features of a malware family. In addition, a graph matching algorithm that is based on the maximum weight subgraph is used to detect malicious code.

Non Graph-based Malware Classification Techniques

In malware classification, there have been proposed other non graph-based malware classification models. Among them, a scalable automated approach for malware classification using pattern recognition algorithms and statistical methods, is presented by Islam *et al.* in [69], utilizing the combination of static features extracted by function length and printable strings. While their evaluation results are very high(i.e., 98.8% classification accuracy), however it is worth mentioning the fact that their experiments include samples from 13 malware families, while the classification accuracy of the model proposed in this paper has been evaluated over 48 malware families. Hence, concerning the impact of philogeny among different malware families the comparative difference between the classification rates achieved by these two models is totally justified, while increasing the number of families in the training set increases the chances of misclassifications.

Recently, Nataraj *et al.* [70] classify malware samples using image processing techniques. Visualizing as gray-scale images the malware binaries, they utilize the fact that,for many malware families, the images belonging to the same family appear very similar in layout and texture. Obviously the results are better than the ones produce by our model however they use at most 25 malware families for their large scale experiments, where the impact of philogeny among different malware families is decreased the less different malware families in the training are. Finally, in [71] Nataraj *et al.* utilize a static analysis technique called binary texture analysis in order to classify malicious binary samples into malware families. They achieve a 72% rate of consistent classification when performing their evaluation on a data set of $60K$ to $685K$ samples comparing their labels with those provided by AV vendors, proving both the accuracy and the scalability of their model.

In the most recent literature, Makandar and Patrot [62] focus on detection and classification of the Trojan viruses using image processing techniques. In their proposed algorithm Gabor wavelet is used for key of feature extraction method and their experimental results are analyzed compared with two classifications such as KNN and SVM.

In [66], Mukesh et al. propose a machine learning based architecture to distinguish existing and recently developing malware by utilizing network and transport layer traffic features.

1.4.3 Related Work on Pandemic Prevention

Epidemic models that are applied to biological threats are also applied to describe the spread of digital security threats like malicious software. The motivation behind the research is triggered by the enormous grow and spread on the number of malicious software and, much more, on mobile devices. The main difference between networks formed by devices connected via Ethernet [72, 73, 74, 75, 76, 77] and networks formed by mobile devices, is that the former are static while the later ones are dynamic networks (i.e., networks that their representing graph changes during time - *ad-hoc* networks). The structure of ad-hoc networks [72, 73, 74, 75, 76, 77] as also the diversity of the nodes concerning so the protection software heterogeneity as the device capabilities [78, 79, 80, 81] are motivating us to investigate the effect of counter-measure's (i.e., security software) response-time on the spread of malware and more precisely on pandemic avoidance. In other words, through this work we focus on investigating how the counter-measure's response time could be crucial on preventing a potential pandemic of a spreading malware, concerning the underlying epidemic model and other factors that affect the spread.

Modern Approaches on Malware Pandemic Prevention

In [82], Chen and Ji focus on modeling the spread of topological malware (spreads based on topology information), as to understanding its potential damages, and developing countermeasures to protect the network infrastructure. Their model is motivated by probabilistic graphs, using a graphical representation to abstract the propagation of malware samples that employ different scanning methods. Utilizing a spatial-temporal random process they describe the statistical dependence of malware propagation in arbitrary topologies. Finally, their results show that the independent model outperforms the previous models, whereas the Markov model achieves a greater accuracy in characterizing both transient and equilibrium behaviors of malware propagation.

In [83], Bose and Shin investigate the propagation of mobile worms and viruses that spread primarily via SMS/MMS messages and short-range radio interfaces such as Bluetooth. In that work, they study the propagation of a mobile virus similar to Commwarrior in a cellular network using data from a real-life SMS customer network, modeling each handheld device as an autonomous mobile agent capable of sending SMS messages to others (via an SMS center) and capable of discovering other Bluetooth equipped devices. Their results show that hybrid worms that use SMS/MMS and proximity scanning (via Bluetooth) can spread rapidly within a cellular network.

Fleizach et al. [84], evaluate the effects of malware propagating using communication services in mobile phone networks. Although self-propagating malware is well understood in the Internet, mobile phone networks have very different characteristics in terms of topologies, services, provisioning and capacity, devices, and communication patterns. To investigate malware in mobile phone networks, they developed an event-driver simulator that captures the characteristics and constraints of mobile phone networks, modeling realistic topologies and provisioned capacities of the network infrastructure, as well as the contact graphs determined by cell phone address books.

An interaction-based simulation framework to study the dynamics of worm propagation over wireless networks developed by Channakeshava et al. [85]. This framework is constructed by their proposed methods for generating synthetic wireless networks using activity-based models of urban population mobility. With this framework they study how Bluetooth worms spread over realistic wireless networks.

Recently, Liu et al. [78] investigate malware's spread taking into account the level of secure protection, in terms of users' security awareness. They develop a new compartmental model concerning heterogeneous immunization, partitioning the traditional susceptible compartment into two sub-compartments, weakly-protected and strongly-protected (weakly-protected susceptible computers have a higher rate of being infected than strongly-protected susceptible computers). The qualitative properties of their model are analyzed through Lyapunov method (taking quadratic functions of independent variables as the candidate Lyapunov functions), and a collection of effective measures for controlling malware spread is proposed, such as keeping as many systems strongly-protected as possible, through numerical simulations.

Latter, Liu et al. [79] investigate the spreading behavior of malware across mobile devices. Modeling mobile networks with complex networks (follow the power-law degree distribution) incorporating the model proposed in [78], and by using the mean-field theory, they propose a novel epidemic model for mobile malware propagation. They calculate the spreading threshold, and analyze the influences of different model parameters as well as the network topology on malware propagation. Through theoretical studies and numerical simulations they show that networks with higher heterogeneity conduce to the diffusion of malware, and complex networks with lower power-law exponents benefit malware spreading. Additionally, malware epidemics over complex networks are greatly different from the previously studied epidemics in fully-connected networks, and the epidemic threshold was found to be density dependent and for all densities considered significantly higher than the value predicted by the previous model.

In [80], Hosseini et al. propose a discrete-time (SEIRS) epidemic model (i.e., Susceptible - Exposed - Infected - Repaired - Susceptible) of malware propagation in scale-free networks (SFNs) considering software diversity. To prevent malware spreading, they use as a parameter the number of diverse software packages installed on nodes, which are calculated using a coloring algorithm. Investigating the existence of equilibria, they compute the basic reproductive ratio and the critical number of software packages for the proposed discrete-time SEIRS model under various conditions, analyzing moreover the local and global stability of the malware-free equilibrium of the model. Through a series of numerical simulations they show that defense mechanisms of software diversity and immunization have important roles in reducing malware's propagation, and that the proposed model is more effective than other existing epidemic models. Finally considering the immunization rate for the cases of uniform immunization and targeted immunization in the proposed epidemic model they show that the targeted immunization is more appropriate than random immunization for controlling malware spreading in SFNs.

Recently, Zema et al. investigate the case of defending against a spreading proximity malware in a network of wireless sensor (WSN) [86]. Using an autonomous flying robot they notify the spread of malicious software over the wireless sensors, by locate, track, access and cure the infected ones. Additionally they propose a mathematical model to decide the optimal path that should be followed by the flying robot as to repair as quick as possible the infected wireless sensors. The authors benchmark their proposed model by the results provided over extended simulations, where their model is compared against classic solutions in different network scenarios.

1.5 Structure of the Thesis

The structure of the thesis is organized as follows. In Section 2 the main principles and properties concerning the scope of the thesis are extensively described. More precisely the fundamental structural components required for the performance of malware detection and classification procedures as also for pandemic prevention are presented, analyzed and discussed over the aspect of their utilization as also how they are combined to consist the basis of this thesis. In Section 3 there are presented the preliminaries regarding the theoretical background (i.e., graph-based representations of malicious software) that has been extensively studied and consists the main corpus over which the proposed models have been developed. Additionally in the same section, it is presented the main contributions of this work, and maybe the one of the major importance, the design and the proposal of the Group Relation Graphs, that actually consist a generalized mutation resilient evolution of the System-call Dependency Graphs, the Coverage Graphs that consist an extension of the Group Relation Graphs as also the Temporal Graphs that depict the temporal structural evolution of the aforementioned graph structures. In Section 4, there are proposed, presented and discussed the similarity metrics utilized by the proposed model for malware detection and classification and are also based upon the commonalities along the features of behavioral graphs. In Section 5 there are presented the proposed procedures for malicious software detection based on the proposed similarity metrics when applied over the proposed graph-based representations of malware's behavior. We discuss the architectural design regarding the setup of an eco-system that incorporates the proposed graph-based representations as also the proposed similarity metrics for distinguishing malicious from benign samples. In Section 6, there are presented the proposed procedures and the wider methodology for malicious software classification based on the proposed similarity metrics when applied over the proposed graph-based representations of malware's behavior regarding their previous indexing into malware families (i.e., sets of malware samples that share common functional characteristics), and following the same structure as in the previous section, we discuss the architectural design regarding the setup of an eco-system that incorporates the proposed graph-based representations as also the proposed similarity metrics for the classification of a test sample that previously has been detected as malicious. Next, in Section 7, it is presented the proposed approach for preventing a pandemic spread of a malicious software that propagates among proximal mobile devices. The main theoretical assumptions regarding the graph-based models for the modeling of the towns-planning, the mobility patterns of the devices as also the underlying compartmental epidemic are all described and extensively discussed, followed by the implementation aspects focusing on the preformed investigation on how the counter-measure's response time affects the malware's spread and leads (or not) to pandemic prevention. In Section8 There are presented the experimental results taken from the evaluation of the proposed models for malware detection and classification, deploying various of the proposed graph-based representations and similarity metrics. Additionally, on the same section are provided experimental results taken from Monte Carlo simulations that achieved through the implementation of the proposed pandemic prevention model regarding various factors that affect the spread across different response-time intervals. In Section 9 there is presented a discussion over the results and a further comparison against other graph-based and non graph-based models for both malware detection and classification procedures, where there have been proven the potentials of the proposed model at its greater extent, as also a few limitations regarding mostly specific implementations aspects. More over, on the same section, beyond the discussion concerning the evaluation of the detection and classification procedures there has been presented a discussion over the simulation results taken from the experiments during the performed study on pandemic prevention. Additionally, another approach has been presented on the same section regarding the deployment of trusted computing (TC) on the side of defense countermeasures. However, in order to integrate the parts of this thesis, the section provides an alignment of the proposed approaches over which a framework that deploys the whole extent of the models presented in this thesis is designed, integrating the proposed graph-based algorithmic techniques into sub-systems in order to incorporate them to construct a robust line of defense to ensure the security against the threat of malicious software. Finally, in Section 10 the thesis concludes discussing the potentials and the limitations exhibited from the proposed models and by presenting specific extensions of the proposed models as also improvements that could be done on the horizon of further research.

Chapter 2

Conceptual Model

The main scope of this thesis is to develop a set of graph-based techniques that consisting the fundamental components of the proposed model will define the principle countermeasures against malware spread. Next there are presented the main principles that construct the basis of the conceptual model over which this thesis is built.

2.1 Representing Malicious Behaviors through Graph Structures

Next there are presented the basic theoretical background regarding the representation of digital object, and especially software, through graph structures.

2.1.1 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 [2].

As referred in [18, 14], most automatic analysis tools utilize an abstract representation of malware's structure such as the Control Flow Graphs (CFGs). According to [1], 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 [87], 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 [18], a *basic block* B is a maximal sequence of instructions $\langle_i, ..., I_m\rangle$ containing at least one control flow instruction at its end. Let V be the set of Bs for a program P and $E \subseteq V \times V \times \{T, F\}$, 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*.

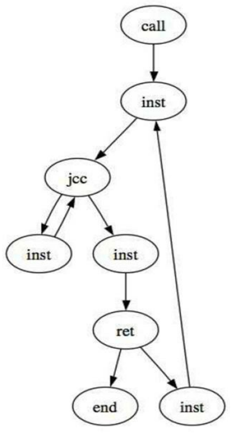
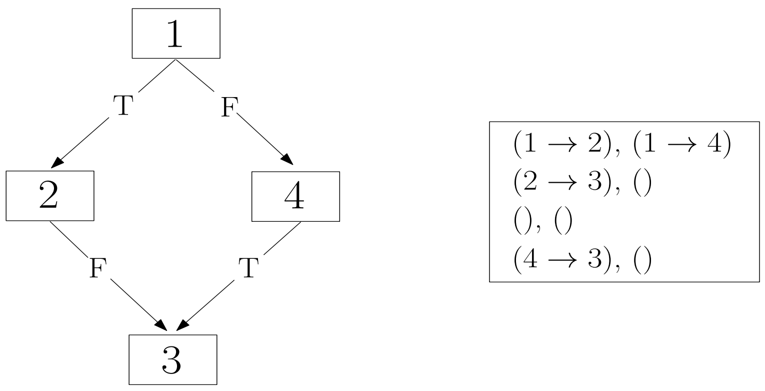


Figure 2.1: Control Flow Graph Representation [1].

The graph-based representations are of major importance since they are able to capture different execution paths of the program under inspection [88]. Additionally, the nodes of the graph can store the instructions and values while they can be interpreted according to more generic semantics [2].

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 [88]. Additionally, isomorphism remains sensitive in morph techniques as code permutation or injection that impact the graph, however these limitations can be addressed by compiler optimizations as referred in [88].

A typical approach for signature creation is presented in [2]. 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 2.2). 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.



$(1 \rightarrow 2), (1 \rightarrow 4)$
$(2 \rightarrow 3), ()$
$(), ()$
$(4 \rightarrow 3), ()$

Figure 2.2: String signature derived by CFG [2].

2.1.2 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 2.3). As referred in [3], the call graph can be gathered from a binary executable through *static analysis*. Actually, disassembly tools like IDA Pro are utilized after the removal of the obfuscation layers (i.e. unpacking).

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.

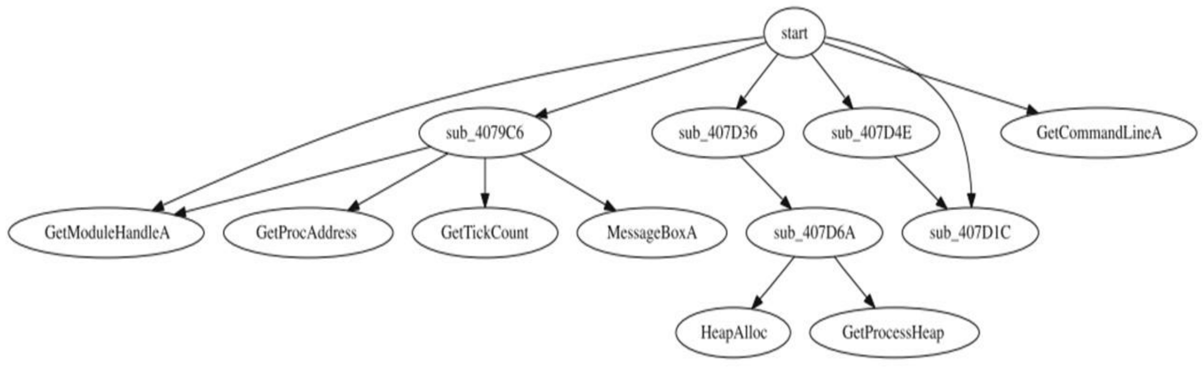


Figure 2.3: Function Call Graph (local and external functions) [3].

2.1.3 Behavioral and 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 [4] 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 [6] 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.

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*. The most usual approach to define an edge in a behavioral graph is the one discussed in [4] 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 dependency between two system calls. Such dependencies can be monitored as we mentioned above through the tainting of data during taint analysis. In Figure 2.4 we cite the behavior graph depicting the dependencies between system calls captured

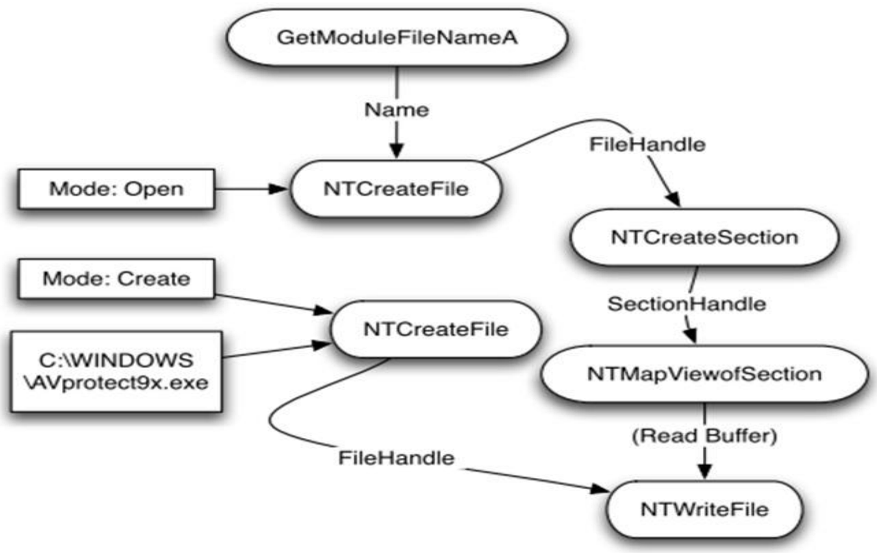


Figure 2.4: Behavior Graph from malware NetSky [4].

through taint analysis during the execution of NetSky malware as presented in [4].

Now, let us proceed with some proper definitions about the behavioral graphs as they are presented in [4, 37, 6, 46]. Compiling the definitions of behavioral graph presented in [4, 37, 6] 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* [4] where the nodes are labeled using system-calls from an alphabet 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 [37] and [6] 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, $\gamma : V \to \Sigma \times 2^{Vars}$ and
- ρ associates constraints with nodes and edges, $\rho : V \cup E \to L_{Dep}$.

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 $\alpha : V \to \Sigma$ associates nodes with the name of their corresponding operations and
- the labeling function $\beta : V \cup E \to L_{Dep}$ 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 [6, 37]:

- *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.

2.2 Knowledge-Base

In order to perform the main procedures of malware detection and classification, regarding its indexing into families of known malicious software samples, actually we need to proceed by implementing particular methods to compare digital object. More precisely, it is needed to be developed a method that would compare a digital object, regardless of its representation, against another one that its intent has to be known. Given that in our case the digital objects refers to an unknown software sample, the target is to compare it against something that is known to be malicious. Hence, in order to define the field of comparison it is selected to compare representations of software's behavior (i.e., graph based representations) against known malicious ones that should be stored into a knowledge database. However, note that such a knowledge data base should include also benign behaviors in order to be utilized as a false positive measurement. Hence the knowledge database consists the fundamental component on the processes of malware detection and classification, as it defines what is known to be malicious and what is not.

2.2.1 Storing Behaviors

As we discussed previously the representation that will be used to compare the software samples (malicious software) is a graph-based structure that depicts the interaction exhibited over the execution of software during its execution time with its host environment (i.e., the Operating System). Particularly, such representations depict the behavioral profile of a software sample, that can be utilized later in order to compare any two given digital objects of this class. Hence, in the proposed approach it is needed to construct a knowledge base of such behavioral profiles, as it is discussed later behavioral graphs, where a set of such graph will represent the behavior of malicious software sample, and another subset of the base will refer to non-malicious, or, in other words benign software samples. The later category of samples are of major importance, since they will be utilized over the evaluation procedure in order to compute the false positive rates of the proposed model.

Several storing approaches could be adopted in order to store, retrieve and process the behavior, however despite their spatial complexity, in the proposed method the adjacency tables have been chosen as the leading technique to store the graph representations of software sample's behavior due to the efficiency exhibited during their process, in terms of accessibility and rational manageability. However, to this point it is worth noting that a $n \times n$ matrix can be deployed to depict the adjacency relations among the nodes of such graphs, where on the same fashion, several matrices, let m can be stored into a uniform structure such as a $n \times n \times m$ matrix, where the relative positions of such matrices should not be left at random as we discuss next.

2.2.2 Organization and Indexing

Malicious software sample tend to exhibit similarities across their structures. More as it is referred through the introduction there exist mutation techniques that given an initial sample of a digital object (in our case a malicious software) they can produce a large set of functionally similar but structurally different copies of this sample (i.e., mutated malicious software samples). However, based on the functionality of these samples a grouping of them can be deployed by merging malicious software sample of similar functionality into groups, the so called malware families.

As we referred previously, several adjacency matrices that represent the behavior graphs of malicious software samples can be stored into a uniform structure, e.g., a three-dimensional matrix in order to create a knowledge database of what is known to be malicious (set of graphs that represent the behavior of a software sample - malware). However, since malicious samples tend to exhibit similar behaviors, in the proposed model it is preferred to group graph representations that depict malicious behaviors of malware samples of similar functionality in to families that in the model correspond to proximal region according to the third dimension (i.e., m) of the $n \times n \times m$ matrix that consists the knowledge database.

2.3 Similarity Metrics

In order to perform the processes of malware detection and classification, beyond the utilization of a knowledge base to compare the unknown sample with something that is known to be malicious, it is needed to establish a comparison method. In the case of comparing digital object, and in this case malicious software sample, the most applicable, efficient and effective method is the utilization of similarity metrics.

2.3.1 Digital Object Comparison

The computation of the similarity between any two given digital objects (in this case malicious software samples) refers to the measurement of structural, functional, or any other kind of closeness of the two specimens. The method used to perform the procedure of computation of the similarity between any given digital objects is called similarity metric.

Most similarity metrics are constructed based on the characteristics of the similarity settings the need to evaluate, and correspondingly the deploy such values over their computation formulas. Similarity metrics have been designed in a way such that the result returned after their computation to be in the range of $[0, 1]$ or $[0, 100]$ through the use of various normalization methods. Computing the similarity metric between digital objects, a return value closer to 0 indicates that the specimens are not similar, while, on the other hand, returned values closer to the upper bound indicate a greater similarity between the samples.

2.3.2 Graph Similarity

In order to perform the computation of similarity between the digital objects (in our case unknown test and known malicious software samples) the proposed model proceed with the computation of graph similarity between a pair of graph representations (i.e., one depicting the behavior of the test sample and one depicting the behavior of the malicious sample).

Several methods could be deployed in order to compute the similarity between a pair of graph representations, however, the vast majority of them is based on the comparison of the edge sets of the two given graphs. Since the vertex set of a behavior graph represents the structural entities that exhibit a behavior pattern, the relations between such entities (i.e., edge set) consists the fundamental element, and hence, the one that should be noticed and examined extensively.

In the proposed model, as we will discuss to a greater extent on th corresponding chapters, there a re proposed various techniques deployed later by the proposed similarity metrics in order to compute the graph similarity. Particularly, specific characteristics of the graph structures, namely relational, quantitative, qualitative, and evolutional characteristics have been leveraged as to measure the similarity by assigning somehow different weights on different types of factors that characterize the nature of an edge of a behavioral graph, taking into account various information resulted over the inspection of the graph's properties.

2.4 Building the Defense Line against Malicious Software

As it has been discussed throughout the previous chapter, the principles of building a defense line against the spread of malicious software are mainly focused in detection and prevention. In other words, an ideal framework proposed for establishing the defense guidelines against malicious software, should include on the one side the definition of a set of techniques for detecting and to a further extent classifying a malicious sample, while on the other, a set of abstract strategies or policies that when deployed over a topology of networked devices would be able to avoid the pandemic spread of a probing malware.

2.4.1 Distinguishing Malicious from Benign Samples utilizing Behavioral Graphs

In the proposed model, the graph similarity procedure is performed over a pair of behavioral graphs, that as will be discussed later, corporate a set of specific characteristics of graph's structure in order to distinguish graphs that represent malicious behaviors from the ones that represent benign behaviors. To this point it is worth

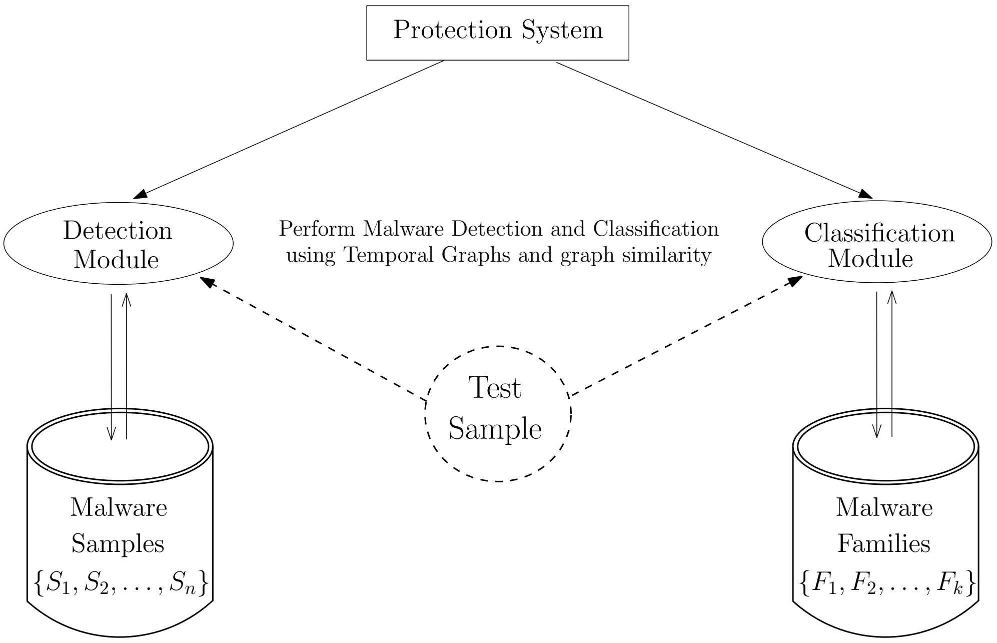

Protection System
Detection Module
Perform Malware Detection and Classification using Temporal Graphs and graph similarity
Classification Module
Test Sample
Malware Samples $\{S_1, S_2, \ldots, S_n\}$
Malware Families $\{F_1, F_2, \ldots, F_k\}$

Figure 2.5: Architecture of the proposed system for detection and classification of malicious software.

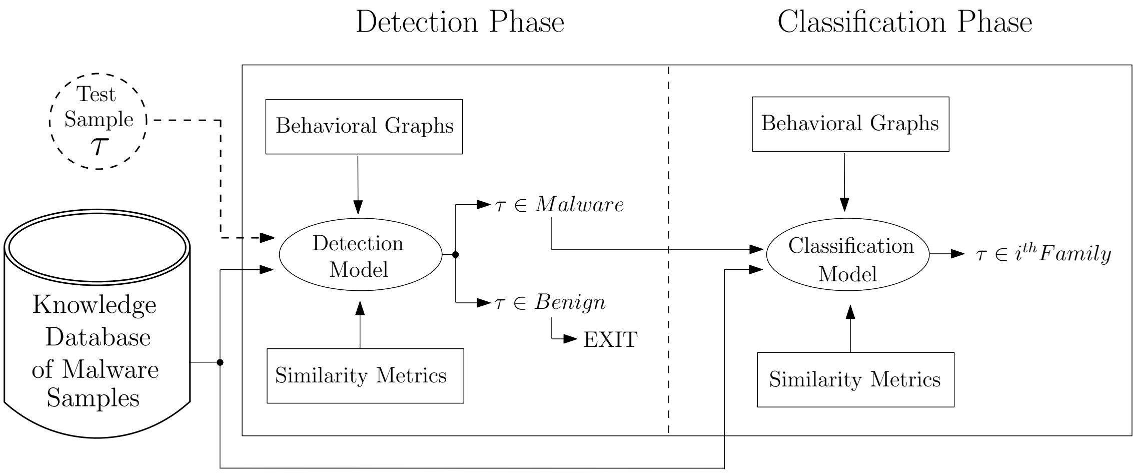


Figure 2.6: The deployment of malware detection and malware classification processes in our model.

nothing that such procedures demand the utilization of a knowledge base of representations of what is "a priori" known to be malicious as also at least one similarity measurement method (i.e., a similarity metric), in order to compute the percentage of "how close" the test sample is, to what is known to be malicious (see, Figure 2.5).

The main scope of this procedure is that, given a graph-based representation of an unknown software sample's behavior (i.e., a behavior graph that depicts its interaction during its execution with its hosting environment - Operating System) and a set of graph-based representations that depict the behaviors exhibited during the execution of known malware samples, to decide based on a similarity metric, if the unknown test sample is actually a malicious or a benign program.

2.4.2 Indexing Malicious Samples into Malware Families utilizing Behavioral Graphs

On the second phase of the development of the proposed countermeasures for defense against malicious software, another procedure, the one of malware classification, or as also referred malware indexing, takes place (see, Figure 2.6). Over this procedure, a software sample under consideration that has been distinguished as malicious according to the preceding detection procedure, has to be classified, or equivalently indexed, to a one of a set of known malware families. Such procedures take into account, beyond the functional similarities exhibited through the execution of the grouped samples, various other characteristics that can be utilized according to a classifier to group a number of samples under the same set.

As it is discussed over the previous chapter, antivirus vendors leverage several heuristic rules in order to group malware samples under the same set. This procedure is performed under the perspective that if a group of samples shares a set of commonalities, with respect to any set of factors, then such commonalities should also be shared over an abstract level of their structures, and hence, at a lower level, more global or, in other words "family level" signature could be also produced in order to cover with one signature a whole malware family, and probably sa few mutations of them. So, to a greater extent, the general scope of indexing malware samples into malware families point to the creating of more generalized signatures that would lead finally to the application of a more effective and efficient detection. Such a consideration leads to the conclusion that the methods of malware detection and classification are actually two procedures that are interacting throughout their execution, where the detection phase should be based upon indexing principles, while the classification phase has as a prerequisite a correct detection decision in order to not distort the quality of the set of known malware samples.

2.4.3 Preventing Malware Pandemics in Mobile Devices

On the second part of the wider framework proposed over this thesis, the main target is to define graph based principles that when applied to the greater extent of the context of mobile devices, is able to avoid or prevent a probable pandemic case of a malware that spreads among mobile devices.

The conceptual model of this approach, contains, among other implementation assumptions, a graph based model for transforming an image taken from Google Maps that represents the town-planning from a region of a city, to an edge-weighted undirected graph, a model for simulation mobile device mobility through the city area (i.e., implementation of shorted path algorithms between points set over the graph that simulates towns-planning), as also e propagation model that simulates the probe of malicious software between proximal mobile devices following the underlying compartmental epidemic model.

Over the above approach, the main concern is to investigate the effect of a factor that represents the time demanded from a countermeasure (antivirus product, removal tool. etc.) to sanitize an infected device. As we will discuss over the corresponding Chapter, in the proposed model, this factor is called “response-time”. In this concept, the effect of response-time is investigate on how the non homogeneity among the mobile devices, leading hence to ranges of response-time, affect the malware’s spread and to a further extent to the prevention of a pandemic spread. In particular, beyond the range of response-time intervals, there are also investigated the effects of the size of the spreading malicious software, the underlying compartmental epidemic model as also the initially infected population (i.e., infected devices at the start of the simulation) and the density of the network (i.e., number of mobile devices coexisting in a specific area).

Chapter 3

Malware Representation through Behavioral Graphs

3.1 Dependency Graphs

Driven by observation, it is noticed that behavioral graphs representing malware samples tend to share some characteristics. With the term *characteristics* we reference the structural commonalities they have, concerning their graph representations, as they are spread among the behavioral graphs of the members in a malware family. So, we distinct three types of characteristics, namely `relational`, `quantitative` and `qualitative` characteristics. Briefly speaking, the term *characteristic* refers to a property that an edge has among a set of graphs, i.e., its existence (or not), its weight, and the significance of that edge. Next we present the aforementioned types of characteristics, describing to a greater extent each one of them.

- **Relational Characteristics**: This type of characteristics refers to the existence or not of an edge. In other words, during the computation of a similarity metric between two GrG graphs we take into account if a specific edge co-exists or not in the two GrG graphs.

- **Quantitative Characteristics**: This type of characteristics concerns the weight of an edge. This means that, measuring the similarity of two GrG graphs, we take into account the difference on the weights of the same edge between the two GrG graphs independently of the co-existence of that edge.

- **Qualitative Characteristics**: This type of characteristics describes the significance of an edge. With the term significance we refer to the percentage of the members in a malware family that have the specific edge. Hence, this type of characteristic is mainly utilized in our model for malware classification, as a set of edges with a greater significance can indicate a whole malware family, constituting thus her identity.

- **Evolutional Characteristics**: The final type of graph characteristics distinguished throughout the investigation of dependency graph properties, is the evolutional characteristics. This type of characteristics represents the structural modifications evolved through the construction in therms of edge modification (i.e., edge creation) of the graph. To this point is critical to refer that the evolution of the structure of a graph takes place over a period of time, which actually refers to the execution time of the represented software sample.

Hence, having described these types of characteristics we can now utilize them through specific similarity metrics so to detect if a given sample is malicious or not, as to decide the malware family of an unclassified malware sample.

3.2 The System-call Dependency Graph (ScDG)

The core works that we base our intuition in the use of system-call dependency graphs are [14, 37, 6] and [5]. 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.

3.2.1 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. Particularly, 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 in the next chapters, the data-set we utilize in order to evaluate the implementation of our proposed model is the same data-set utilized in [5] 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 3.1. 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.

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

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 3.1: System Call Traces.

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.

In Table 3.2 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 Table 3.2 we can proceed by constructing the System-Call Dependency Graph that is a directed acyclic graph (DAG). The vertex set of this graph is consisted from the system-call that took place during the execution of

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 3.2: System-call Dependencies.

the sample and we have captured their trace (Table 3.2) and its edge set is consisted by their in-between dependencies (Table 3.2)

In Figure 3.1 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 3.2. 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

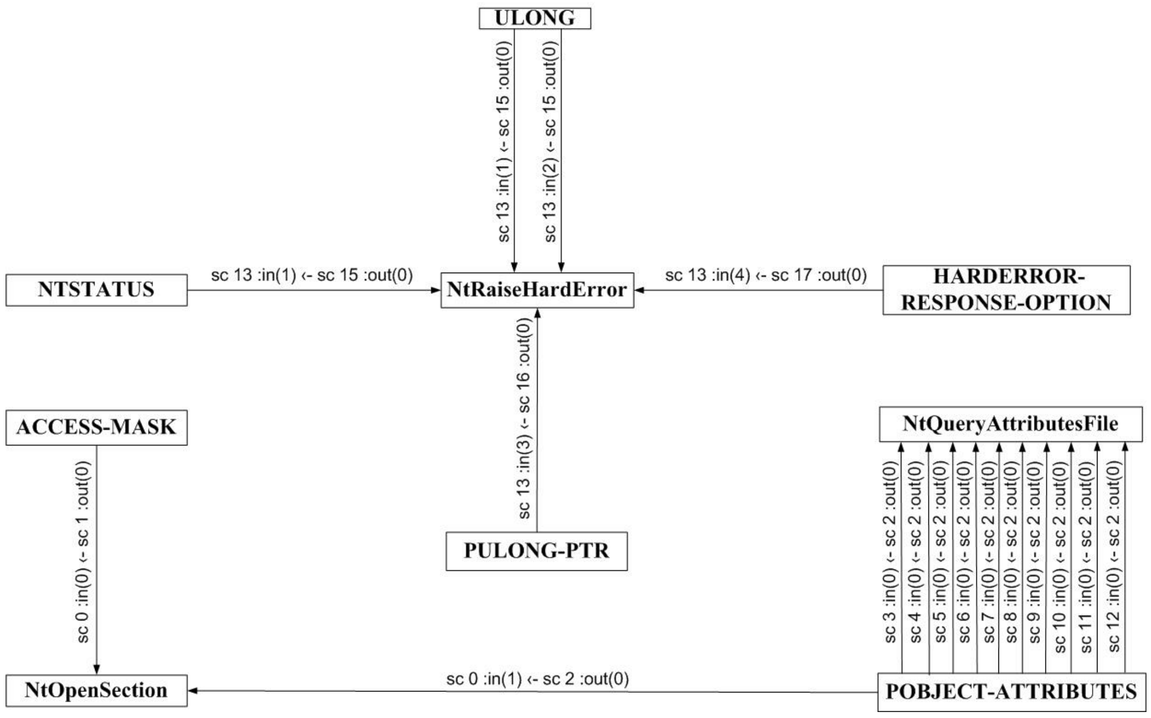

ULONG
sc 13 :in(1) <- sc 15 :out(0)
sc 13 :in(2) <- sc 15 :out(0)
NTSTATUS
sc 13 :in(1) <- sc 15 :out(0)
NtRaiseHardError
sc 13 :in(4) <- sc 17 :out(0)
HARDERROR-RESPONSE-OPTION
sc 13 :in(3) <- sc 16 :out(0)
ACCESS-MASK
sc 0 :in(0) <- sc 1 :out(0)
PULONG-PTR
NtQueryAttributesFile
sc 3 :in(0) <- sc 2 :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

Figure 3.1: System Call Dependency Graph.

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.

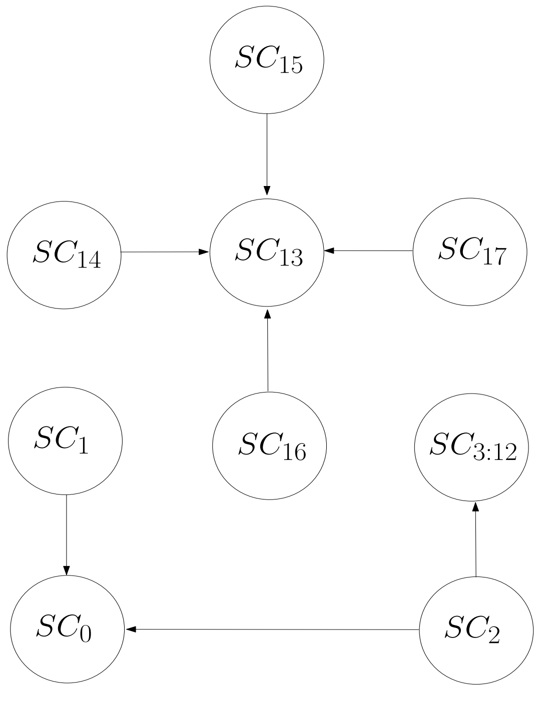


Figure 3.2: Simplified System Call Dependency Graph.

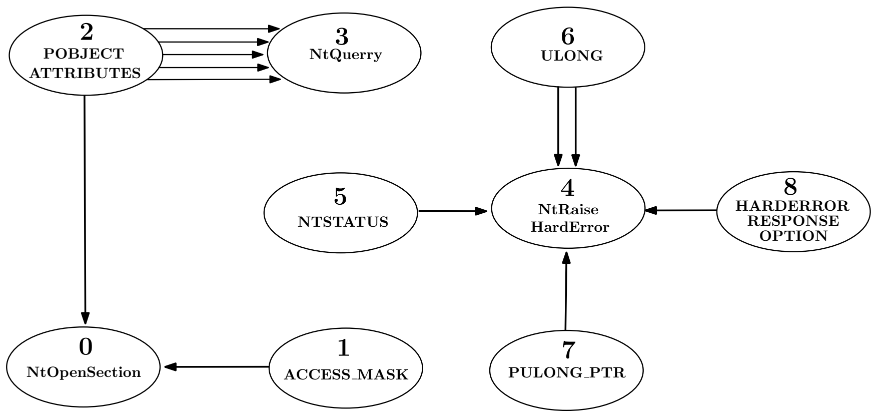


Figure 3.3: A system-call dependency graph $D[P]$ of a program P.

As we discussed previously, the actions performed by a program, depicting its behavior, rely on System-calls. Tracing the System-calls invoked during the execution of a malware program P, we can represent its behavior interpreting this information with a graph, so called *System-call Dependency Graph* (or, ScDG for short); throughout the paper, we shall denote a ScDG by $D[P]$ and the System-calls invoked by P by S_i, $1 \leq i \leq n$. The vertex set of a ScD-graph $D[P]$ is consisted by all the system-calls invoked during the execution of a program P, i.e., S_1, S_2, …, S_n, while its edge set contains the pairs of System-calls that exchanged arguments during the execution representing data-flow dependencies between System-calls. Thus, an edge of ScDG $D[P]$ is a tuple of type $(S_i{:}k,\ S_j{:}\ell)$ indicating that the system-call S_i invokes S_j and the kth output argument of S_i is passed as the ℓ^{th} input argument in S_j.

Recalling that the suspicious sample needs to be executed in a contained environment (i.e., a virtual machine), where during its execution time, dynamic taint analysis is performed in order to capture system-call traces, next we illustrate a simple example that includes the system-call traces obtained, constructing the ScDG of a program. In Figure 3.3, it is easy to see that the vertex set of this graph is consisted from the system-calls invoked during the execution of the sample and its edge set is consisted by their in between data-flow dependencies, constructing a directed acyclic graph (dag). To this point we ought to notice that a dependency graph is by its definition acyclic, since each node (System-call in our case) uses as its input the output from another system call (so the first system call depends on the second) and so on, constructing hence an acyclic graph.

3.3 The Group Relation Graph (GrG)

The key idea of our detection and classification model is based on the fact that sSystem-calls of similar functionality can be gathered into the same group, as firstly presented in [89]. For a proper System-call grouping, we utilize the 30 System-call groups provided by Microsoft's documentation for MS-Windows, where each System-call has a detailed description indicating the group it belongs to; hereafter, we denote by $\mathcal{C}^*$ the set of system-call groups for a given operating system and by $\mathcal{C}_1$, $\mathcal{C}_2$, ..., $\mathcal{C}_{n^*}$ the groups of $\mathcal{C}^*$.

Group Name	Size	Group Name	Size
ACCESS_MASK	1	PHANDLE	1
Atom	5	PLARGE_INTEGER	1
BOOLEAN	1	Process	49
Debug	17	PULARGE_INTEGER	1
Device	31	PULONG	1
Environment	12	PUNICODE_STRING	1
File	44	PVOID_SIZEAFTER	1
HANDLE	1	PWSTR	1
Job	9	Registry	40
LONG	1	Security	36
LPC	47	Synchronization	38
Memory	25	Time	5
NTSTATUS	1	Transaction	49
Object	19	ULONG	1
Other	36	WOW64	19

Table 3.3: The 30 System-call groups - Total Groups.

Thus, if a System-call Dependency Graph $D[P]$ of a given program P is composed by n system-calls S_1, S_2, ..., S_n, then each system-call S_i, $1 \leq i \leq n$, belongs to exactly one group $\mathcal{C}_j$, $1 \leq j \leq n^*$. In Table 3.3, we present the groups of system-calls $\mathcal{C}_1$, $\mathcal{C}_2$, ..., $\mathcal{C}_{n^*}$ and the number of System-calls inside each group.

Having the grouping $\mathcal{C}^*$ and a ScDG $D[P]$, we next construct the key component of our model that is the *Group Relation Graph* (or, for short, GrG). The graph GrG, which we denote by $D^*[P]$, is a directed weighted graph on n^* nodes u_1, u_2, …, u_{n^*}; it is constructed as follows:

(i) we define a bijective function $f : V(D^*[P]) \longrightarrow \mathcal{C}_i$ from set $V(D^*[P]) = \{u_1, u_2, \ldots, u_{n^*}\}$ to the set of groups $\mathcal{C}_i = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{n^*}\}$;

(ii) for every pair of nodes $\{u_i, u_j\} \in V(D^*[P])$, we add the directed edge (u_i, u_j) in $E(D^*[P])$ if (S_p, S_q) is an edge in $E(D[P])$ and, $S_p \in \mathcal{C}_i$ and $S_q \in \mathcal{C}_j$, $1 \leq i, j \leq n^*$;

(iii) for each directed edge $(u_i, u_j) \in E(D^*[P])$, we assign the weight w if there are w invocations from a System-call in group $f(u_i) = \mathcal{C}_i$ to a System-call in group $f(u_j) = \mathcal{C}_j$, $1 \leq i, j \leq n^*$.

ID	System-call	Group
0	NtOpenSection	Memory
1	ACCESS_MASK	ACCESS_MASK
2	POBJECT_ATTRIBUTES	Object
3	NtQueryAttributesFile	File
4	NtRaiseHardError	Process
5	NTSTATUS	NTSTATUS
6	ULONG	ULONG
7	PULONG_PTR	Process
8	HARDERROR_RESPONSE_OPTION	Process

Table 3.4: The 9 system-calls of Figure 3.3 and their corresponding groups - Active Groups.

We point out that the number of non-isolated nodes of graph $D^*[P]$ equals the number of groups formed by the system-call of graph $D[P]$; note that, the total number of nodes of $D^*[P]$ is always n^*. For example, the 9 System-calls of ScDG graph belong to 7 groups (see, Table 3.4), the ScDG graph $D[P]$ of Figure 3.3 contains 9 nodes, while its corresponding GrG graph $D^*[P]$ contains 7 non-isolated nodes and thus 23 isolated nodes in $Iset$; see, Figure 3.4.

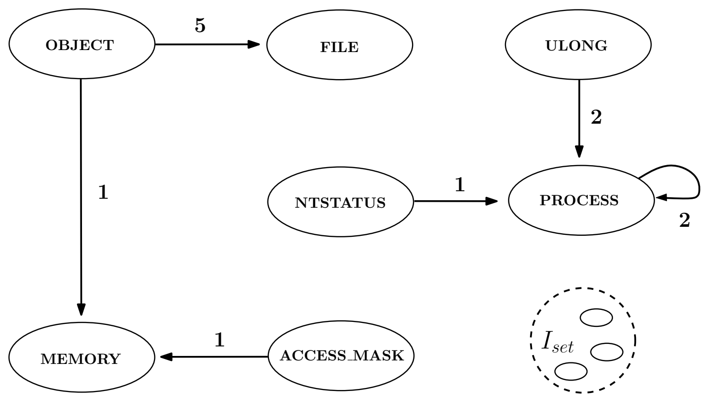


Figure 3.4: The GrG graph $D^*[P]$ of the graph of Figure 3.3.

Figure 3.4 depicts the GrG graph $D^*[P]$ of the ScDG graph $D[P]$ of Figure 3.3; the set I_{set} contains all the isolated nodes of $D^*[P]$. We point out that while the ScDG $D[P]$ is by definition an acyclic directed graph, the produced GrG $D^*[P]$ is not in general acyclic, since by grouping nodes in $D[P]$ it is very likely to create directed circles and/or self-loops; see, Figure 3.4.

Observing the Figure 3.4, we ought to refer that the number of non-isolated nodes of graph $D^*[P]$ equals the number of groups formed by the System-calls of graph $D[P]$; note that, the total number of nodes of $D^*[P]$ is always n^*, namely `Total Groups`. For example, the 9 system-calls of ScDG belong to 7 groups (see, Table 3.4), the ScDG $D[P]$ of Figure 3.3 contains 9 nodes, while its corresponding GrG $D^*[P]$ contains 7 non-isolated nodes, namely `Active Groups` and thus 23 isolated nodes, namely `Inactive Groups` in I_{set}; see, Figure 3.4, obviously, `Active Groups` $+$ `Inactive Groups` $=$ `Total Groups`. We mention that, we name a System-call group as `Active Group` if it contains a System-call that is invoked in a data-flow dependency with another System-call (the node corresponding to this group has non-zero in- or out-degree), or `Inactive Group` otherwise.

Finally, notice that while the ScDG graph $D[P]$ is by definition an acyclic directed graph, the produced GrG graph $D^*[P]$ is not, in general, acyclic. As easily one can see that by grouping nodes in $D[P]$ it is very likely to create directed circles and/or self-loops; an indicative example appears in graph $D^*[P]$ of Figure 3.4.

Properties of Group Relation Graphs. Driven by observation, we noticed that GG graphs representing malware samples tend to share some characteristics. With the term *characteristics* we reference the structural commonalities they have, concerning their graph representations, as they are spread among the GrG graphs of the members in a malware family. So, we distinct three types of characteristics, namely `relational`, `quantitative` and `qualitative` characteristics. Briefly speaking, the term *characteristic* refers to a property that an edge has among GrG graphs, i.e., its existence (or not), its weight, and the significance of that edge. Next we present the aforementioned types of characteristics, describing to a greater extent each one of them.

3.4 The Coverage Graph (CvG)

As we described previously, the System-calls invoked during the execution of a program can be traced through taint analysis, and hereafter the behavior of a program can be represented with a directed acyclic graph (dag), the so called System-call Dependency Graph see, Figure 3.5(a). The vertex set of a ScDG is consisted by all the System-calls invoked during the execution of a program and its edge set represents the communication between System-calls as described in [89, 5, 6].

Then, given a graph representation of malware's behavior such a ScDG, a more abstract graph representation of a program's behavior can be constructed based on the fact that System-calls of similar functionality can be classified into the same group. The produced graph representation is a directed weighted graph called Group Relation Graph; see, Figure 3.5(b).

As described previously, having the grouping of system-calls and a system-call dependency graph ScDG, the GrG graph $D^*[P]$ is a directed edge-weighted graph on n vertices v_1, v_2, …, v_n constructed as follows:

(ii) for every pair of vertices $\{v_i, v_j\} \in V(D^*[P])$, a directed edge (v_i, v_j) is added in $E(D^*[P])$ if the two system-calls communicating with each other, let (S_p, S_q), is an edge in $E(D[P])$ and, S_p belongs to the i-th system-call group and S_q belongs to the j-th system-call group;

(iii) for each directed edge $(v_i, v_j) \in E(D^*[P])$, a weight $w(v_i, v_j) \in \Re$ is assigned on it if there are $w(v_i, v_j)$ invocations from a system-call in the i-th group to a system-call in the j-th group.

Having defined the GrG graph $D^*[P]$, we also define the underlying vertex-weighted graph $D^+[P]$ of the graph $D^*[P]$ having vertex-weights $w(v_i) = \sum_{v_j \in Adj(v_i)} w(v_i, v_j)$,

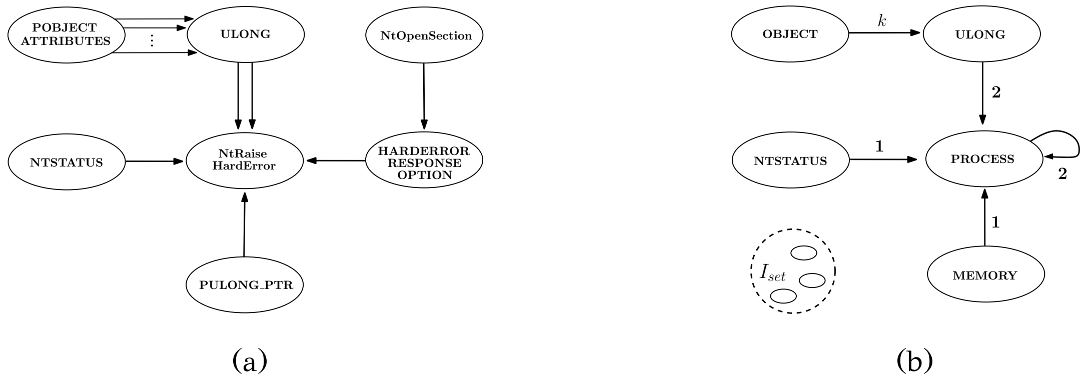


Figure 3.5: (a) The System-call Dependency Graph of a program; (b) The corresponding Group Relation Graph of a program.

for every $v_i \in V(D^+[P])$.

As mentioned previously, a GrG graph $D^*[P]$ is a edge-weighted directed graph which, in our approach, we transform it to its underlying vertex-weighted undirected graph $D^+[P]$. We first define domination relations on the vertices of the graph $D^+[P]$ and then utilizing these relations we construct the Coverage Graph of the GrG graph $D^*[P]$, denoted by $C[p]$.

Next, it is also presented a 2D-representation of the underlying vertex-weighted graph $D^+[P]$ of the graph $D^*[P]$ utilizing the degrees and the vertex-weights of its vertices and show a different way to compute the domination relations on the graph $D^+[P]$.

Definition 1 Let $D^+[P]$ be the underlying vertex-weighted graph of a GrG graph $D^*[P]$ and let v_i, $v_j \in V(D^+[P])$. We say that v_i *dominates* v_j, denoted by $v_i \xrightarrow{dom} v_j$, if $deg(v_i) \geq deg(v_j)$ and $w(v_i) \geq w(v_j)$, where $deg(v)$ and $w(v)$ denote the degree and the weight of the vertex $v \in V(D^+[P])$, respectively.
The domination set D_i of a vertex $v_i \in V(D^+[P])$ is the set of all the vertices v_j : $v_i \xrightarrow{dom} v_j$. If $v_i \xrightarrow{dom} v_j$ we say that vertices v_i and v_j are in a domination relation.

Definition 2 Let $D^+[P]$ be the underlying vertex-weighted graph of a GrG graph $D^*[P]$ with vertices $V(D^+[P]) = \{v_1, v_2, \cdots, v_n\}$. The Coverage Graph (CvG) of the GrG graph $D^*[P]$, denoted also $C[P]$, is a directed graph defined as follows:

(i) $V(C[P]) = \{v_1^*, v_2^*, \cdots, v_n^*\}$ and $\{v_1^*, v_2^*, \cdots, v_n^*\} \leftrightarrow \{v_1, v_2, \cdots, v_n\}$;

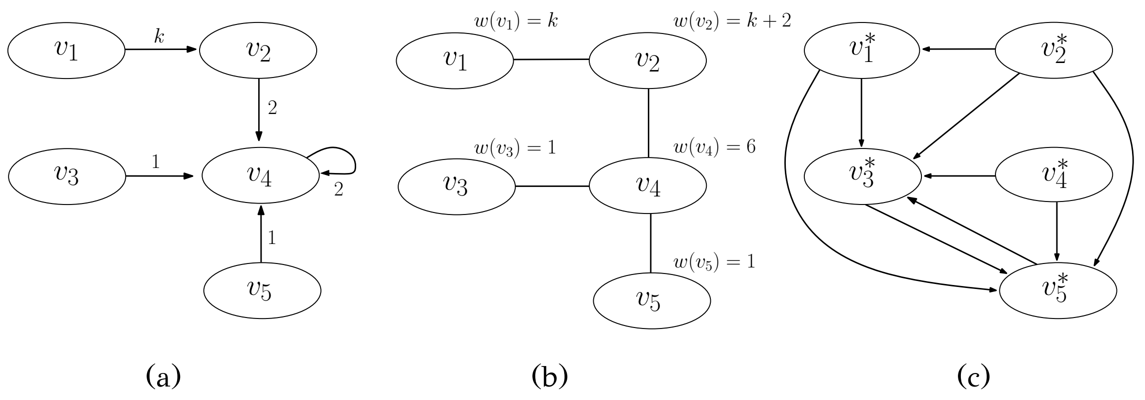


Figure 3.6: (a) A GrG graph $D^*[P]$; (b) Its underlying vertex-weighted graph $D^+[P]$; (c) The CvG graph $C[P]$ produced by the graph $D^+[P]$ through its vertex domination relations.

(ii) $v_i^* v_j^* \in E(C[P])$ if $v_i \xrightarrow{dom} v_j$, where v_i and v_j correspond to v_i^* and v_j^*, respectively.

In Figure 3.6(a) we show a GrG graph $D^*[P]$ which is isomorphic to the GrG graph $D^*[P]\backslash I_{set}$ of Figure 3.5(b), in Figure 3.6(b) we depict its underlying vertex-weighted graph $D^+[P]$ with $w(v_i) = \sum_{v_j \in Adj(v_i)} w(v_i, v_j)$, $\forall v_i \in V(D^+[P])$, where $w(v_i, v_j)$ is the weight of the edge $(v_i, v_j) \in E(D^*[P])$, while in Figure 3.6(c) we show the Coverage Graph $C[P]$ constructed from the graph $D^+[P]$ by utilizing its vertex domination relations. Note that the vertices of each graph in Figure 3.6 correspond the System-call groups of Figure 3.5(b).

3.5 Temporal Graphs

Throughout the development of our research, we have noticed that, to the best of our knowledge, there does not exist any approach on the literature that references or leverages the factor of the temporal evolution of a graph. Similarly to *philogeny* that examines the temporal evolution of malware families, the key component of our proposed detection and classification model, leverage the temporal evolution of graphs (i.e., GrG and CvG graphs) in order to depict the structural modifications performed on the graph and that could distinguish either a malware sample or to a further extent a malware family.

In our model, we define two types of graphs that depict the temporal evolution of our initial graph structures (i.e., Group Relation Graphs and Coverage Graphs), namely Group Relation Temporal Graphs or, for short, GrTG and Coverage Temporal Graphs or, for short, CvTG, respectively. In order to implement such graph structures we approach this modeling by creating instances of the initial GrG and CvG graphs during their construction. As we mentioned above, GrG graphs are constructed by the sum of the system-calls invoked interconnecting pairs of system call groups, and correspondingly CvG are constructed by they respective dominating relation (i.e., by their supremacy regarding degree and weight) between the system-call groups. Hence, since we are given the system-call dependencies in a series that depicts the time correlation among (i.e., an edge sequence of the System-call Dependency Graph that shows the system-call invocations during execution time), such constructions can be obtained by creating an instance of the produced graphs (i.e., GrG, CvG) at specific steps.

Formalizing our previous claim, we can define that for a set of time-slots, let $t_1, t_2, \ldots, t_n$ we can construct n instances of graphs GrG and CvG and denote them as $T_1(D^*[P]), T_2(D^*[P]), \ldots, T_n(D^*[P])$ and $T_1(C[P]), T_2(C[P]), \ldots, T_n(C[P])$, respectively, that depict the structure in terms of edges, vertex-degrees and vertex-weights of the corresponding graphs at specific time slots. Through this approach we can maintain information about the temporal evolution of the graph thorough its construction procedure,and further leverage such information in order to perform more elaborated graph similarity techniques.

3.5.1 Partitioning Time

The factor of time actually does not represent the actual quantum of run-time, but each time-quantum corresponds to one system-call dependency or, equivalently, relation between two System-call Groups (i.e., edge of the Group Relation Graph). Hence, the total time-line depicts the slots or time-partitions from the appearance of the first to the last group relation.

Additionally, in our model, we define as epochs the set of time-partitions, i.e., $t_1, t_2, \ldots, t_n$, and an epoch, let t_i, contains the structural modifications (i.e., edges added on the corresponding GrG or CvG graph) from the begin to the end of the i^{th} epoch, where $1 < i < n, \forall n \in K$, and $K = \{n : |E(G)| \mod n = 0, \forall n \in N\}$.

As we described throughout the paper, the conceptual substance of Temporal

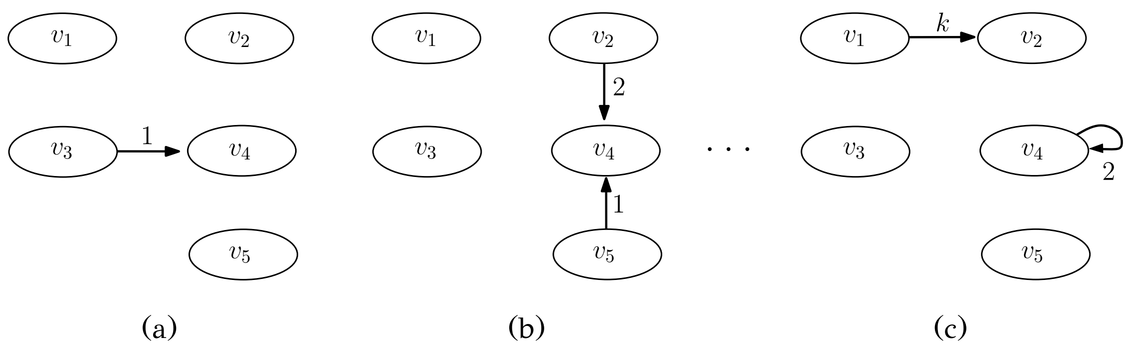


Figure 3.7: The temporal evolution of a GrG graph $D^*[P]$ represented by its Discrete Modification Temporal Graph $T^f(D^*[P])$ over n epochs: (a) $T^f_1(D^*[P])$, (b) $T^f_2(D^*[P])$ and (c) $T^f_n(D^*[P])$.

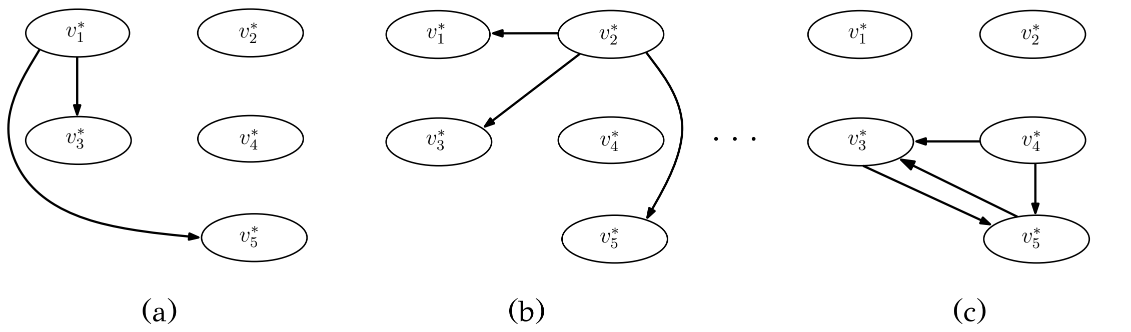


Figure 3.8: The temporal evolution of a CvG graph $C[P]$ represented by its Discrete Modification Temporal Graph $T^f(C[P])$ over n epochs: (a) $T^f_1(C[P])$, (b) $T^f_2(C[P])$ and (c) $T^f_n(C[P])$.

Graphs is to depict the structural evolution of the GrG and CvG graphs through the time. However, the structural modification on the instances of the graph over the time can be described either discretely as addition of edges over the exact previous graph instance, or cumulatively as successive additions of edges performed on all the previous graph instances. Next, we discuss the construction of the corresponding Temporal Graphs according to the two approaches.

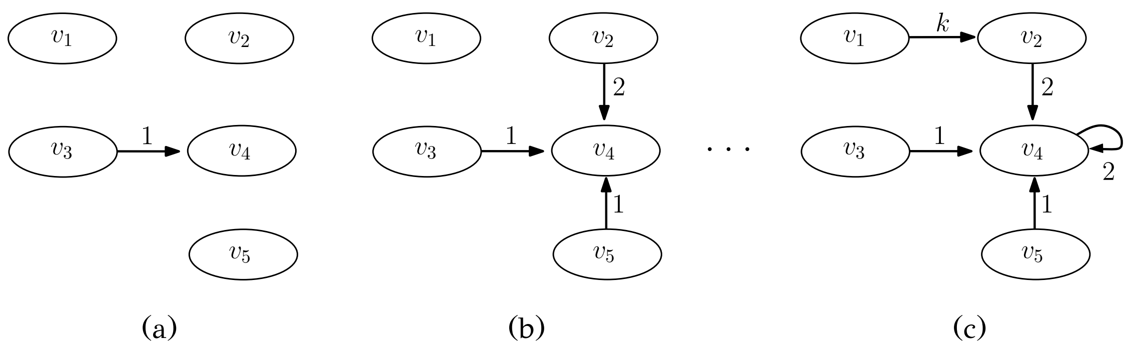


Figure 3.9: The temporal evolution of a GrG graph $D^*[P]$ represented by its Cumulative Modification Temporal Graph $T^F(D^*[P])$ over n epochs: (a) $T_1^F(D^*[P])$, (b) $T_2^F(D^*[P])$ and (c) $T_n^F(D^*[P])$.

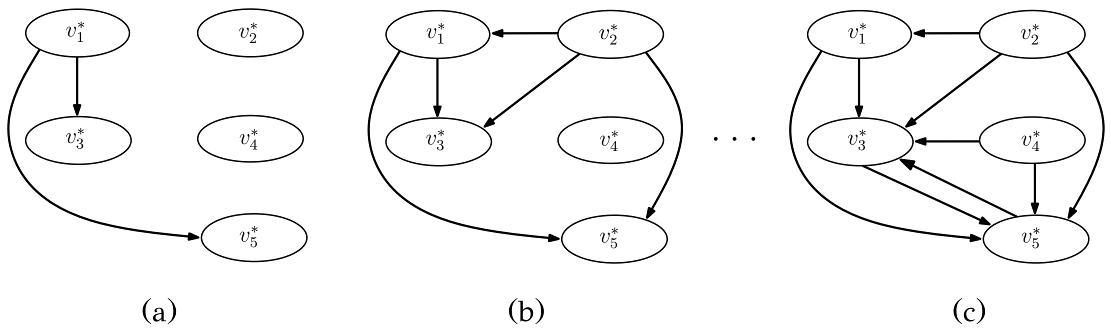


Figure 3.10: The temporal evolution of a CvG graph $C[P]$ represented by its Cumulative Modification Temporal Graph $T^F(C[P])$ over n epochs: (a) $T_1^F(C[P])$, (b) $T_2^F(C[P])$ and (c) $T_n^F(C[P])$.

Discrete Modification Temporal Graphs.

In the first approach of our proposed scheme, the construction of the Temporal Graph, that represents the evolution of GrG or CvG graphs during time, constructs the induced subgraph of GrG and CvG, respectively, including only the edges that where added on a specific epoch. So, let the epoch t_i we construct the Temporal Graphs $GrTG_i$, $CvTG_i$ of the graphs GrG and CvG, denoting them with $T_i^f(D^*[P])$, $T_i^f(C[P])$, respectively, where f denotes the cardinality of edges added on this epoch. In Figure 3.7 and Figure 3.8, we depict the discrete structural modification (i.e., temporal evolution) of graphs GrG and CvG over the construction of their corresponding Temporal Graphs $T^f(D^*[P])$ and $T^f(C[P])$ during n epochs.

Cumulative Modification Temporal Graphs.

In this type of Temporal Graphs, the evolution of the graph is represented as an additive procedure, since once an edge has been created at a given time, let i, on the graph between two system-call groups on the GrG graph, or a domination relation has been resulted on the CvG graph, it will remain permanent on the ancestor Temporal Graphs (i.e., if $\{u, v\} \in E(T_i(D^*[P])) \rightarrow \{u, v\} \in E(T_j(D^*[P])$ and if $\{u, v\} \in E(T_i(C[P])) \rightarrow \{u, v\} \in E(T_j(C[P]))$, $\forall i < j < n$), since it consists a predecessor of the following temporal graphs. In the second approach of our proposed scheme, the construction of the Temporal Graph, that represents the evolution of GrG or CvG graphs during time, actually extends the graphs GrG and CvG, respectively, during time, by adding on them the edges that where added on a specific epoch. So, let the epoch t_i we construct the Temporal Graphs $T(D^*[P])$, $T(C[P])$ of the graphs GrG and CvG, denoting them with $T_i^F(D^*[P])$, $T_i^F(C[P])$, respectively, where F denotes the cardinality of edges added from epoch t_1 until epoch t_i. In Figure 3.9 and Figure 3.10, we depict the cumulative structural modification (i.e., temporal evolution) of graphs GrG and CvG over the construction of their corresponding Temporal Graphs $T^F(D^*[P])$ and $T^F(C[P])$ during n epochs.

Chapter 4

Comparing Digital Objects through Graph-based Similarity Metrics

Next ewe present the proposed similarity-metrics that will be utilized further for malware detection and classification.

4.1 Δ-Similarity Metric

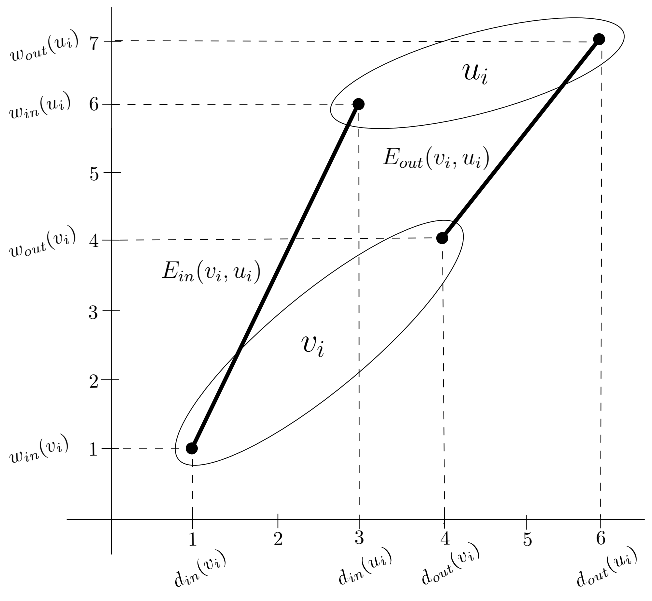
Next we present the δ-distance, that utilizing the Euclidean distance measures the structural likeness between two given GrG graphs.

It is well known that the Euclidean Distance between two points, say, $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q_2, ..., q_n)$ in R^n space, is defined as follows:

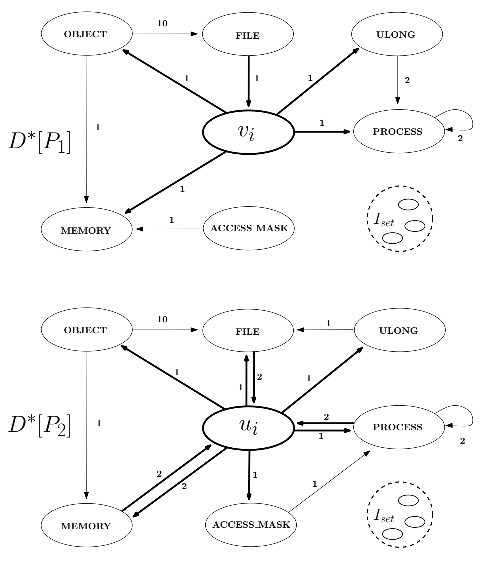
$$E(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}. \tag{4.1}$$

Having defined the Euclidean distance in R^n space, then we proceed by defining our Δ-Similarity metric and presenting the process of computing it.

As we mentioned above, we focus on the computation of similarity between two GrG graphs based on characteristics that reveal their structural likeness. Specifically, for each node x of the GrG graph, we utilize the in-degree $d_{in}(x)$ and the out-degree $d_{out}(x)$, along with the corresponding averaged weights $w_{in}(x)$ and $w_{out}(x)$ respectively, in order to represent it on the 2D plane by points with coordinates the aforementioned characteristics.



(a)



(b)

Figure 4.1: (a) The Cartesian representation of the corresponding nodes v_i and u_i of the GrG graphs $D^*[P_1]$ and $D^*[P_2]$, respectively; (b) Two GrG graphs $D^*[P_1]$ and $D^*[P_2]$.

We first compute the δ-distance between two GrG graphs $D^*[P_1]$ and $D^*[P_2]$ by summing the Euclidean distances of each pair of corresponding nodes; note that, GrG graphs are labeled graphs as their nodes correspond to System-call groups (see, Table 3.3) and hence we can select all the pairs of corresponding nodes of two such graphs.

More precisely, for a pair of programs, say, P_1 and P_2, let $D^*[P_1] = (V_1, E_1)$ and $D^*[P_2] = (V_2, E_2)$ be their corresponding GrG graphs and let V_1=$\{v_1, v_2, \ldots, v_k\}$ and V_2=$\{u_1, u_2, \ldots, u_k\}$.

Then, we define the δ-distance on $D^*[P_1]$ and $D^*[P_2]$, as follows:

$$\delta(D^*[P_1], D^*[P_2]) = \sum_{i=1}^{k}[\alpha \cdot E_{in}(v_i, u_i) + \beta \cdot E_{out}(v_i, u_i)] \tag{4.2}$$

where,

$\alpha + \beta = 1$ and

$E_z(v_i, u_i) = \sqrt{(d_z(v_i) - d_z(u_i))^2 + (w_z(v_i) - w_z(u_i))^2}\,,\ \ z \in \{in, out\}.$

note that, x is the i-th node in both GrG graphs $D^*[P_1]$ and $D^*[P_2]$.

We point out that $\alpha, \beta \in \Re$ are two factors experimentally determined and $E_z(v_i, u_i)$ is the Euclidean distance between the two points (nodes) v_i and u_i. Having described the δ-distance, let us next define the Δ-similarity metric which is the decision component of our detection model. Since Δ-similarity utilizes the δ-distance metric, we compute it as follows:

$$\Delta(D^*[P_1], D^*[P_2]) = \frac{\Gamma}{\Gamma + \delta(D^*[P_1], D^*[P_2])} \tag{4.3}$$

where, $\Gamma \in N^+$ and $0 \leq \Delta() \leq 1$.

Finally, we ought to refer that our choice of using Γ is driven by the fact that Δ-similarity metric should return values in the range $[0, 1]$. So, reversing its distance properties, for two GrG graphs (let $D^*[P_1], D^*[P_2]$) a Δ-similarity value between them that is close to 0 indicates that the graphs are dissimilar, or identical when this value is close to 1. Hence, as the usage of Γ is exclusively for reversing the distance properties of Δ-similarity metric, the value of Γ can be set experimentally as any positive integer.

4.2 $\overline{\Delta}$-Similarity Metric

The $\overline{\Delta}$-Similarity Metric follows the perspective of Δ-similarity metric, since using the δ-distance previously defined, computes the distance of each vertex of the graph of the test sample from the corresponding "median-vertex" of the graphs of the known malware samples belonging to the same malware family. Next, we present the procedure of constructing the "median-vertex", for each node of a given graph and how it is represented on te Cartesian plane by its degree and averaged weight.

In order to construct for each vertex let v_i of a set of graphs, let $G_1, G_2, \dots G_m$

that correspond to the GrG graphs of the m members of a malware family, the "median-vertex", that we shall denote by $\overline{v_i}$, it should be computed the mean in-degree (resp. out-degree) as also the corresponding mean of the averaged in-weights (resp. mean averaged out-weights) by each of the corresponding vertices v_i of the graphs $G_1, G_2, \ldots G_m$.

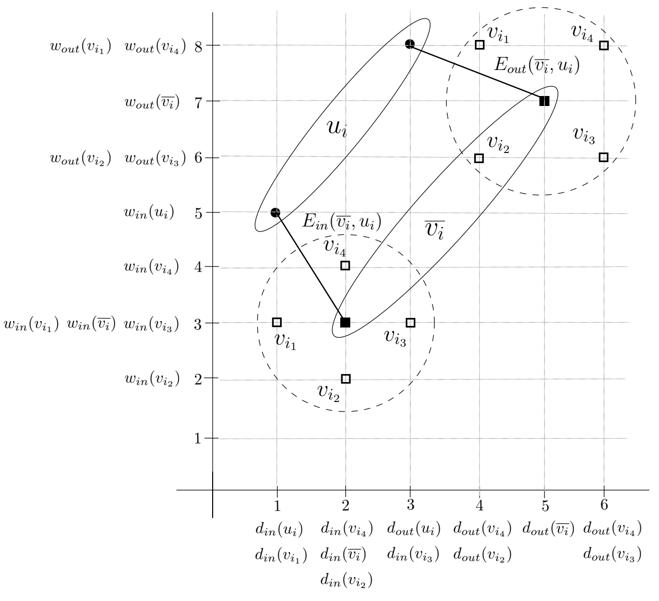
To this point we ought to notice that since the procedure of averaging a value is weak against outlier values, it is preferred to utilize the median value computed by the $\lfloor (n+1)/2 \rfloor$ ordinate statistic.

In particular, the i-th ordinate statistic of a set S of a plurality of n is the i-th smaller component of the S-set. For example, the minimum in a set of elements is the 1st ordinate statistic ($i = 1$), and the maximum is n ordinate statistics ($i = n$). The median of a set S is the $\lfloor (n+1)/2 \rfloor$ ordinate statistic, i.e., the element $e \in S$ that is greater than just $\lfloor (n+1)/2 \rfloor - 1$ elements of S or, equivalently the element $e \in S$ has exactly $\lfloor (n+1)/2 \rfloor - 1$ smaller elements than this in S.

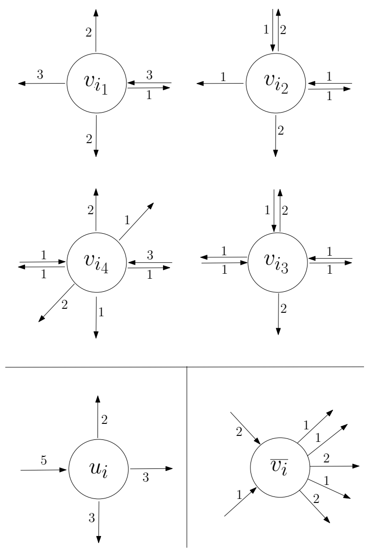
Hence, using the method described above we compute on this way the median for the degrees and the averaged weights of each corresponding vertex v_i from each of the graphs $G_1, G_2, \ldots G_m$ utilizing respectively both the in- and out-degrees and averaged weights. In Figure 4.2, we represent the construction of the "median-vertex" for a set of 4 graphs (i.e., four vertices $v_{i_1}, v_{i_2}, v_{i_3}$, and v_{i_4} are utilized for the computations of the medians of the in/out degrees and averaged weights) and how we measure the δ-distance between the vertex u_i, which belongs to the vertex set of the GrG of a test sample and the constructed $\overline{v_i}$ produced from the corresponding vertices $v_{i_1}, v_{i_2}, v_{i_3}$, and v_{i_4} that belong to the GrG graphs of known malware samples belonging to the same malware family.

Focusing on the computation of similarity between two GrG graphs based on characteristics that reveal their structural likeness the similarity between each vertex u_i of the test sample's graph and the "median-vertex" produced as the vertex with the mean in/out degree and averaged in/out weights of each corresponding vertices $v_{i_1}, v_{i_2}, v_{i_3}$, and v_{i_4} that belong to the GrG graphs of known malware samples of a malware family. These two factors, i.e., in/out degree and averaged in/out weight of each vertex are plotted on the the Cartesian plane and a modification of δ-distances between the vertices of the two GrG graphs are computed by summing the Euclidean distances of each pair of corresponding nodes.

More precisely, for a set of known malware programs, let, $P_1, P_2, \ldots P_m$ of a mal-



(a)



(b)

Figure 4.2: (a) The construction of the “median-vertex” $\overline{v_i}$ from a set of corresponding vertices and the computation of δ-distances between the vertex u_i and $\overline{v_i}$; (b) A node u_i from the vertex set of test sample’s GrG and the corresponding $v_{i_1}, v_{i_2}, v_{i_3}$, and v_{i_4} that belong to the GrG graphs of known malware samples belonging to the same malware family.

ware family F, let $D^*[P_1] = (V_1, E_1), D^*[P_2] = (V_2, E_2), \dots D^*[P_m] = (V_m, E_m)$ be their corresponding GrG graphs and let $V_1 = \{v_{1_1}, v_{1_2}, \dots, v_{1_k}\}, V_1 = \{v_{2_1}, v_{2_2}, \dots, v_{2_k}\} \dots V_1 = \{v_{m_1}, v_{m_2}, \dots, v_{m_k}\}$ be the corresponding vertex set of these graphs and, respectively, for a unknown program let, P_t, let $D^*[P_t] = (V_t, E_t)$ be its GrG graph and let $V_t = \{u_1, u_2, \dots, u_k\}$ its corresponding vertex set..

Then, we define the $\overline{\delta}$-distance on $\{D^*[P_1], D^*[P_2], \dots D^*[P_m]$ belonging on a malware family F and $D^*[P_t]$, as follows:

$$\overline{\delta}(F, D^*[P_t]) = \sum_{i=1}^{k} [\alpha \cdot E_{in}(\overline{v_i}, u_i) + \beta \cdot E_{out}(\overline{v_i}, u_i)] \tag{4.4}$$

where,

$\alpha + \beta = 1$ and

$E_z(v_i, u_i) = \sqrt{(d_z(\overline{v_i}) - d_z(u_i))^2 + (w_z(\overline{v_i}) - w_z(u_i))^2}\,, \ \ z \in \{in, out\}.$

note that, $\overline{v_i}$ and u_i correspond to the i-th nodes in both GrG graphs.

Similar to Δ-similarity metric next is defined the $\overline{\Delta}$-similarity metric, which utilizes the $\overline{\delta}$-distance, as follows:

$$\overline{\Delta}(F, D^*[P_t]) = \frac{\Gamma}{\Gamma + \delta(F, D^*[P_t])} \tag{4.5}$$

where, $\Gamma \in N^+$ and $0 \leq \Delta() \leq 1$.

4.3 Cover Similarity Metric

In order to perform the malware detection process, we actually compute graph similarity between pairs of CvG graphs: the CvG graph of an unknown sample and a graph of a set of CvG graphs from known malware specimens. The graph similarity is computed over the edge set of each pair of CvG graphs using the Jaccard similarity metric.

Let $C[P_1]$ be the CvG graph of an unknown sample and $C[P_2]$ be a CvG graph of the set of known malware samples. The Cover-Similarity $CS(\cdot)$ of the graphs $C[P_1]$ and $C[P_2]$ is computed as follows:

$$CS(C[P_1], C[P_2]) = \frac{E(C[P_1]) \cap E(C[P_2])}{E(C[P_1]) \cup E(C[P_2])}, \tag{4.6}$$

where $E(C[P_1])$ and $E(C[P_2])$ are the edge sets of the two tested CvG graphs; note that, the vertices $v_1^*, v_2^*, \ldots, v_n^*$ of the graph $C[P_1]$ correspond to the vertices $u_1^*, u_2^*, \ldots, u_n^*$ of the graph $C[P_2]$.

4.4 SaMe Similarity Metric

In order to measure the similarity between the test sample T and any member M of a malware family $F \in \mathcal{Q}$ we compute the satisfiability of relational characteristics, in terms of edge co-existence in both GrG graphs, utilizing the Jaccard similarity J and the satisfiability of quantitative characteristics, in terms of edge weight, utilizing both the Bray-Curtis similarity BC and the Cosine similarity CS. Next, describe the computation of SaMe similarity between a test sample T and each member M of a family F_k.

Jaccard similarity. Describing the Jaccard similarity we ought to refer that since it operates on binary values we perform a casting in the adjacency matrices representing the corresponding GrG graphs $D^*[T]$ and $D^*[M]$ by turning any non-zero values to 1s. Hence, in order to measure the similarity concerning the edge co-existence, we compute the Jaccard similarity $J(T^C, M^C)$ between the test sample's casted adjacency matrix T^C and the member's casted adjacency matrix M^C, as follows:

$$J(T^c, M^c) = \frac{|T^c \cap_{11} M^c|}{|T^c \cap_{11} M^c| + |T^c \cap_{10} M^c| + |T^c \cap_{01} M^c|}. \tag{4.7}$$

Bray-Curtis similarity. Bray-Curtis similarity is mostly applied for the computation of diversity between two objects represented by vectors of continuous values. More precisely, this metric measures the similarity of quantitative characteristics in terms of edge weights, between the test sample's adjacency matrix T and a member's adjacency matrix M. However, since Bray-Curtis is known as a dissimilarity metric referring to the distance between two objects, we reverse this property performing a subtraction from 1. Hence, utilizing their initial adjacency matrices, we compute the Bray-Curtis similarity between T and M denoting it with $BC(T, M)$ as follows:

$$BC(T, M) = 1 - \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(T(i,j) - M(i,j))}{\sum_{i=1}^{n}\sum_{j=1}^{n}(T(i,j) + M(i,j))}. \tag{4.8}$$

Cosine similarity. The Cosine similarity between the test sample and the current member of a family, is computed taking into account the quantitative characteristics, utilizing their initial adjacency matrices as presented below:

$$CS(T, M) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(T(i,j) \times M(i,j))}{\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} T(i,j)^2} \times \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} M(i,j)^2}}. \tag{4.9}$$

SaMe-similarity Computation. Having already computed the aforementioned similarity metrics between T and a member M_ℓ ($1 \leq \ell \leq m_k$) of a malware family F_k ($1 \leq k \leq N$), next we define the corresponding SaMe similarity between the unknown test sample T and a known malware family F_k by computing the maximum value that appears among all the members of the family as follows:

$$SaMe(T, F_k) = \max_{1 \leq \ell \leq m_k} [J(T^C, M_\ell^C) \times BC(T, M_\ell) \times CS(T, M_\ell)] \tag{4.10}$$

where, m_k is the size of the kth malware family F_k in set $\mathcal{Q}$, $1 \leq k \leq N = |Q|$.

4.5 NP Similarity Metric

The NP-similarity metric focuses on the computation of similarity between the test sample T and any malware family F_k using the family's ID-Matrix that we describe next, as also the initial GrG graphs, operating on them using similarity metrics that take into account different types of characteristics (i.e., qualitative, relational and quantitative).

Next, we provide a description of the family's ID-Matrix construction and what we call qualitative characteristics proceeding by the presentation of the components of NP-similarity metric. Briefly speaking, the similarity metrics Family.to.Test Cover similarity metric FT, and its complement Test.to.Family Cover similarity metric TF, compute the rate of satisfiability of the strong qualitative characteristics of a family F_k by the casted adjacency matric T^c of test sample T and vice versa using the family's ID-Matrix. On the other hand, mean Jaccard similarity metric $\overline{J}$ and mean Bray-Curtis similarity metric $\overline{BC}$, measure the mean values of similarity in terms of relational and quantitative characteristics respectively, between T and the members of a known malware family F_k.

Family ID-Matrix. We next define the family ID-matrix which focuses on storing valuable information about the significance of an edge across the members of a malware family. Thus, we focus on edges that exist in most of GrG graphs $D^*[P]$ of the members in a family, constituting hence a qualitative characteristic of their family. More precisely, we are based on the notion that, if a specific edge appears in the ma-

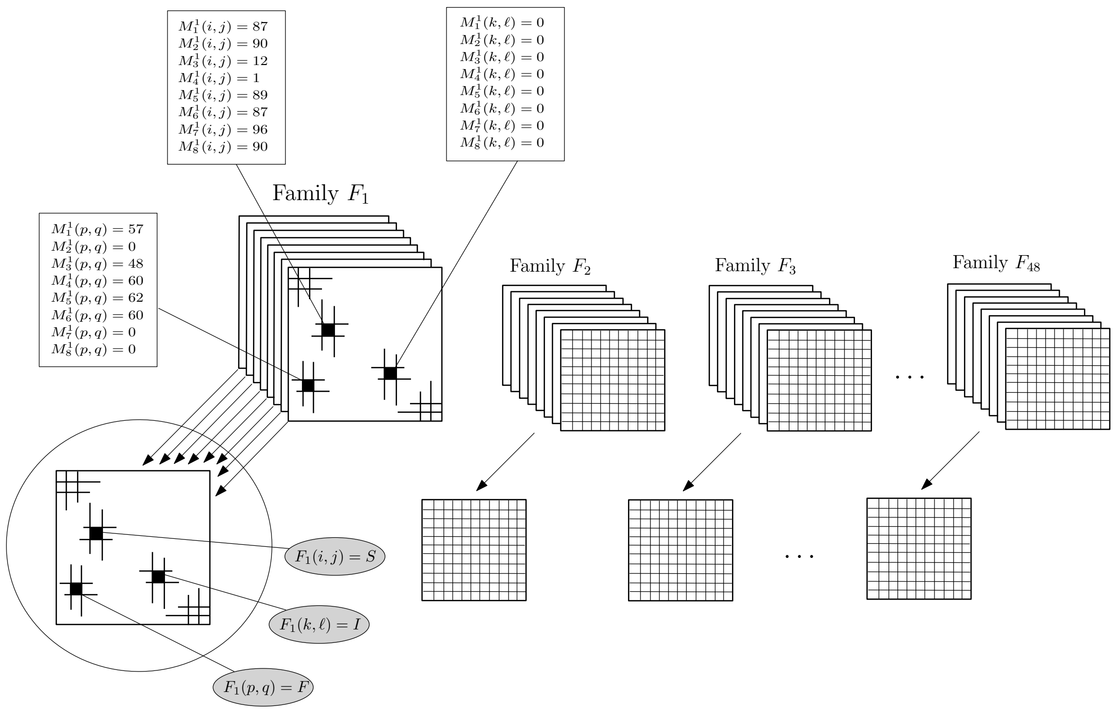


Figure 4.3: The ID-Matrix.

jority of the members' GrG graphs inside the malware family, then this edge exposes a greater significance, in contrast with another one that exists in the minority of them.

Hence, in order to represent the significance of an edge, we take into account the percentage of the members in a family in which this edge has a non-zero value, and assign a significance tag to the corresponding cell of family's ID-Matrix as shown in Figure 4.3. The significance tags are assigned, partitioning the values of edge appearance percentage (ranging from 0 to 100) to three categories, namely `SIGNIFICANT`, `FUZZY`, and `INSIGNIFICANT`, or, for short, `S`, `F`, and `I`, respectively.

In our model, the `SIGNIFICANT` tags cover cells containing values in the range $[0.95 - 1]$ (i.e. $F_k(i,j) = \texttt{S}$), the `FUZZY` tags cover cells containing values in the range $(0.05 - 0.95)$ (i.e. $F_k(i,j) = \texttt{F}$), while the `INSIGNIFICANT` tags cover the ones containing values in the range $[0 - 0.05]$ (i.e. $F_k(i,j) = \texttt{I}$). Hence, the Family ID-Matrix is a 30×30 (i.e., cardinality of system-call groups) matrix, one per malware family, where its cells have tags representing the significance of each edge in this family.

Below, we show an accumulative view of the aforementioned partitioning on the edge-appearance percentage ranges.

$$F_k(i,j) = \begin{cases} \mathtt{S}, & if\ \frac{|M_\ell(i,j)>0|}{m_k} \in [0.95, 1] \\ \mathtt{I}, & if\ \frac{|M_\ell(i,j)=0|}{m_k} \in [0.95, 1] \\ \mathtt{F}, & otherwise \end{cases} \tag{4.11}$$

where, M_ℓ $(1 \leq \ell \leq m_k)$ is the ℓ-th member of malware family F_k, $|M_\ell(i,j) > 0|$ is the number of members of family F_k in which the edge (i,j) (i.e. relation from System-call group i to System-call group j) has a non-zero value $(0 \leq |M_\ell(i,j) > 0| \leq m_k)$, and m_k is the size of malware family F_k.

It is worth noting that, in order to compare the test sample's adjacency matrix T with the ID-Matrix F_k of the kth family of the set $\mathcal{Q}$, we first need to make a cast on test sample's adjacency matrix T. Recall that, the cells of the test sample's adjacency matrix T have either zero or non-zero values. Thus, we cast any non-zero values existed in test sample's adjacency matrix T into 1s resulting the casted matrix T^c.

Finally, we ought to notice that the approach presented above, as also the selection of the corresponding percentage ranges, are based on the intuition that we mostly take into account the edges of GrGs that exist in the majority of the members in a malware family, as it depicts a strong qualitative characteristic for that family. So, since a value close to 1 would allow only a few edges to be tagged as significant, reducing so our field of appliance where only the most frequent edges would be tagged as significant, we experimentally tuned optimally this range to $[0.95, 1]$. Hence, the most significant edges are those who exist in at least 95% of the members of a malware family, while, on the other hand, the least significant ones (insignificant) are those that do not exist in at least 95% of the members of a malware family.

Family.to.Test Cover. The main process of this similarity metric is to compute the satisfiability of the strong qualitative characteristics of a malware family F_k (i.e., $F_k(i,j) = \mathtt{S}$) by the edges of test sample T's GrG graph (i.e., $F_k(i,j) = \mathtt{S} \wedge T^c(i,j) = 1$). In order to compute such a quantity by a percentage we divide by the total number of cells in family's ID-Matrix F_k that have a `SIGNIFICANT` tag (i.e., $F_k(i,j) = \mathtt{S}$). Thus, the formula that gives the Family.to.Test cover similarity metric is the following:

$$FT(T^c, F_k) = \frac{|F_k \cap_{\mathtt{S}\to 1} T^c|}{|F_k = \mathtt{S}|} \tag{4.12}$$

where, F_k is the ID-Matrix of the kth family of the set $\mathcal{Q}$ and T^c is the test sample's casted adjacency matrix.

Test.to.Family Cover. In contrast with Family.to.Test Cover similarity metric, Test.to.Family Cover similarity metric aims on computing the satisfiability of the edge existence of T's GrG graph by the strong qualitative characteristics in the ID-Matrix of malware family F_k (i.e., $T^c(i,j) = 1 \wedge F_k(i,j) = \mathsf{S}$). In order to compute such a quantity by a percentage we divide by the total number of cells edges in T's GrG graph. Thus, the formula that gives the Test.to.Family Cover similarity metric is the following:

$$TF(T^c, F_k) = \frac{|T^c \cap_{1 \to \mathsf{S}} F_k|}{|T^c = 1|} \tag{4.13}$$

where, again F_k is the ID-Matrix of the kth family of the set $\mathcal{Q}$ and T^c is the test sample's casted adjacency matrix.

Mean Jaccard similarity. This metric measures the mean value of similarity in terms of relational characteristics, regarding the edge co-existence, between the test sample's casted matrix T^c and the casted matrices of the members of a malware family F_k. So, for a malware family F_k, containing m_k members, let $M_1^c, M_2^c, \ldots, M_{m_k}^c$ be the casted matrices of the members of F_k, we then compute the mean Jaccard similarity as follows:

$$\overline{J}(T^c, F_k) = \frac{\sum_{\ell=1}^{m_k} J(T^c, M_\ell^c)}{m_k} \tag{4.14}$$

where, F_k is the kth malware family of a set $\mathcal{Q}$, T^c is the test sample's casted adjacency matrix, and m_k is the size of the family F_k.

NP-similarity Computation. Having already computed the Mean Jaccard $\overline{J}(\cdot)$, Family.to.Test Cover $FT(\cdot)$ and Test.To.Family Cover $TF(\cdot)$ similarity metrics, we next define their corresponding NP-similarity as follows:

$$NP(T, F_k) = \overline{J}(T^c, F_k) \times TF(T^c, F_k) \times FT(T^c, F_k). \tag{4.15}$$

where, T is the test sample and F_k is a known malware family.

Chapter 5

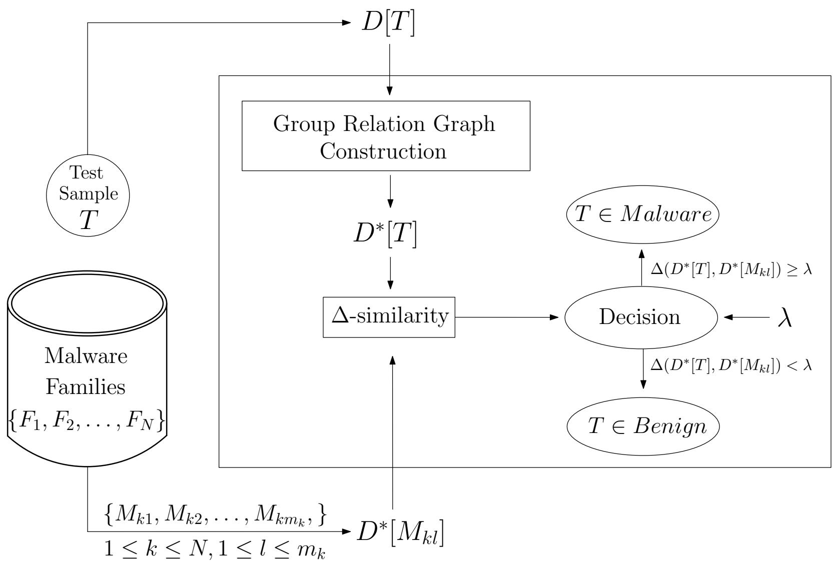
Detecting Malicious Behaviors

Next there is presented the utilization of the proposed similarity techniques for distinguishing malicious from benign samples.

5.1 Computing Δ-Similarity between Group Relation Graphs

We implement our malware detection model by first performing a transformation to the initial ScD-graphs, converting them to GrG graphs as we described in the previous section, and then computing the Δ-similarity metric in order to measure the structural similarities between two given GrG graphs. Next, we describe the main process of determining if an unknown sample is malicious or benign based on the result of Δ-similarity metric when applied on its GrG graph and a set of GrG graphs that represent known malicious software samples organized into malware families. In Figure 5.1 we depict the total architecture of our proposed model for detecting malicious software samples.

In the example of Figure 5.1 we suppose we are given an unknown test sample T



$D[T]$
Group Relation Graph Construction
Test Sample T
$D^*[T]$
$T \in Malware$
$\Delta(D^*[T], D^*[M_{kl}]) \geq \lambda$
Δ-similarity
Decision
λ
Malware Families $\{F_1, F_2, \ldots, F_N\}$
$\Delta(D^*[T], D^*[M_{kl}]) < \lambda$
$T \in Benign$
$\{M_{k1}, M_{k2}, \ldots, M_{km_k}\}$
$1 \leq k \leq N, 1 \leq l \leq m_k$
$D^*[M_{kl}]$

Figure 5.1: Architecture of the detection model.

that we do not know if it is malicious, and we are asked to decide if T is malicious or benign. Having a data-base with the GrGs of known malware samples organized into families, firstly we proceed by constructing the GrG representing T using its ScDG. Once $D^*[T]$ is been constructed, we compute the Δ-Similarity metric between $D^*[T]$ and any $D^*[M_{ij}]$ (i.e., the j-th member of the i-th malware family), or, in other words, between the GrG representing T and any GrG representing a known malware into our data-base. So, let S the total number of malware samples in our date-base, we result to S values of Δ-Similarity, one per pair (T, Malware Sample), where if the maximum value appeared is greater that a threshold λ indicates that T is malicious.

Hence, formulating the whole process, in order to determine if the GrG graph $D^*[T]$, which represents an unknown software sample T, is malicious or benign we compute the Δ-Similarity between $D^*[T]$ and every graph representing a known malicious software from any malware family. More precisely, for each malware family we store, as a representative, the maximum result of Δ-Similarity exhibited by a member of her (i.e., the minimum distance). Finally, we compare the maximum result of Δ-Similarity produced among all the representatives with a predefined experimentally tuned threshold λ, and if this value is above that threshold we claim that the GrG graph $D^*[T]$ represents a malicious software.

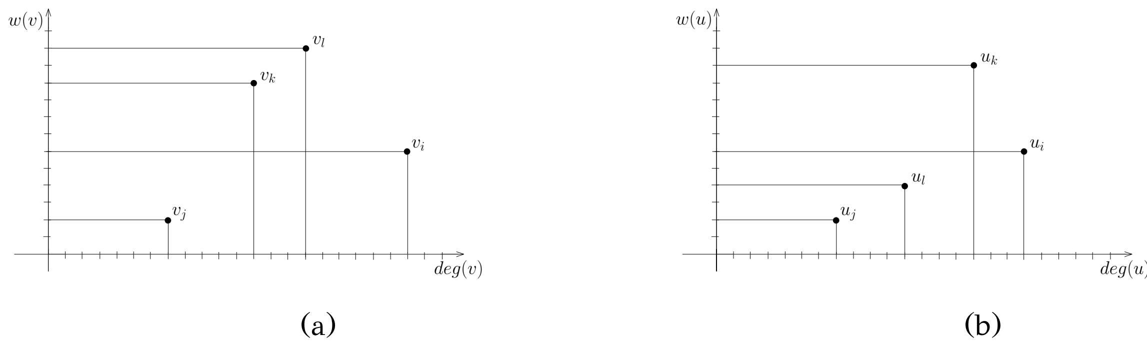


Figure 5.2: (a) The 2D-representation of the graph $D^+[P]$ of a GrG graph $D^+[P]$ produced by the ScDG of an unknown software; (b) The 2D-representation for the case where the ScDG graph is a known malicious software.

5.2 Computing Cover Similarity between Coverage Graphs

In order for a program P to construct its corresponding CvG graph $C[P]$ from the underlying vertex-weighted graph $D^+[P]$ of a GrG graph $D^*[P]$, we present a different way to compute the domination relations defined on the vertices of $D^+[P]$. In fact, we utilize the degrees and the vertex-weights of the graph $D^+[P]$ and we map the vertices $v_1, v_2, \ldots, v_n$ of $D^+[P]$ to points $p_1, p_2, \ldots, p_n$ on the Cartesian plane; for a point p_i, we denote by $x(p_i)$ and $y(p_i)$ the $x-$ and $y-$coordinate of p_i, respectively. More precisely, given the graph $D^+[P]$ of a GrG graph $D^*[P]$, we represent each vertex $v_i \in V(D^+[P])$ as a point on the Cartesian plane based on the mapping:

$$\text{vertex } v_i \rightarrow \text{point } p_i = (deg(v_i), w(v_i)).$$

In Figure 5.2 we show a 2D-representation of the underlying vertex-weighted graph $D^+[P]$ of a GrG graph $D^*[P]$. The 2D-representation in Figure 5.2(a) depicts the domination relations on the vertices of the test sample's GrG, while the one in Figure 5.2(b) depicts the domination relations on the vertices of the malware sample's GrG. As we can observe, according to the definition of the domination relation, in Figure 5.2(a) the domination sets of the vertices v_i, v_j, v_k, and v_l are $D_i = \{v_j\}$, $D_j = \emptyset$, $D_k = \{v_j\}$, and $D_l = \{v_k, v_j\}$, respectively, while in Figure 5.2(b) the vertex domination sets of the corresponding vertices u_i, u_j, u_k, and u_l are $D'_i = \{u_l, u_j\}$, $D'_j = \emptyset$, $D'_k = \{u_l, u_j\}$, and $D'_l = \{u_j\}$, respectively.

As we described previously, we represent a GrG graph $D^*[P]$ on the Cartesian plane through the use of its underlying vertex-weighted graph $D^+[P]$ of $D^*[P]$. However, despite the fact that the degree of a vertex is uniquely defined, its corresponding weight may be computed in various ways. Hence, in our model we follow two approaches namely the *sum* and the *mean* vertex weights. For a vertex $v_i \in V(D^*[P])$, the former is defined as $w_s(v_i) = \sum_{v_j \in adj(v_i)} w(v_i, v_j)$, while the later is defined as $w_m(v_i) = w_s(v_i)/deg(v_i)$.

Through the implementation of our graph-based malware detection model, we first convert the initial ScDG graphs to GrG graphs, and then compute the domination relations, utilizing the weight and the degree of each vertex of the underlying vertex-weighted graph $D^+[P]$ of GrG graph $D^*[P]$, to construct the corresponding CvG graph $C[P]$ as described previously.

In order to perform the malware detection process, we actually compute graph similarity between pairs of CvG graphs: the CvG graph of an unknown sample and a graph of a set of CvG graphs from known malware specimens. The graph similarity is computed over the edge set of each pair of CvG graphs using the Jaccard similarity metric.

Let $C[P_1]$ be the CvG graph of an unknown sample and $C[P_2]$ be a CvG graph of the set of known malware samples. The Cover-Similarity $CS(\cdot)$ of the graphs $C[P_1]$ and $C[P_2]$ is computed as follows:

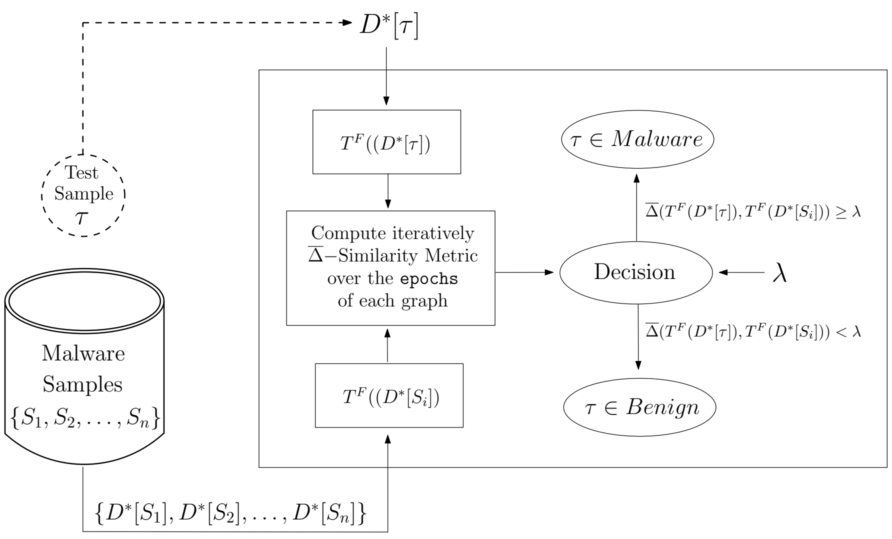
$$CS(C[P_1], C[P_2]) = \frac{E(C[P_1]) \cap E(C[P_2])}{E(C[P_1]) \cup E(C[P_2])}, \tag{5.1}$$

where $E(C[P_1])$ and $E(C[P_2])$ are the edge sets of the two tested CvG graphs; note that, the vertices $v_1^*, v_2^*, \ldots, v_n^*$ of the graph $C[P_1]$ correspond to the vertices $u_1^*, u_2^*, \ldots, u_n^*$ of the graph $C[P_2]$.

5.3 Computing $\overline{\Delta}$-Similarity between Temporal Graphs

Next we discuss the operation of our proposed graph based malware detection model, and present an overview on its constructional principles alongside with a brief discussions over its implementation aspects.

Model Overview. We implement our malware detection model by first performing a transformation to the initial ScDG graphs, converting them to GrG graphs, computing



$D^*[\tau]$
Test Sample τ
$T^F((D^*[\tau])$
Compute iteratively $\overline{\Delta}$–Similarity Metric over the epochs of each graph
$T^F((D^*[S_i])$
Malware Samples $\{S_1, S_2, \ldots, S_n\}$
$\{D^*[S_1], D^*[S_2], \ldots, D^*[S_n]\}$
$\tau \in Malware$
$\overline{\Delta}(T^F(D^*[\tau]), T^F(D^*[S_i])) \geq \lambda$
Decision
λ
$\overline{\Delta}(T^F(D^*[\tau]), T^F(D^*[S_i])) < \lambda$
$\tau \in Benign$

Figure 5.3: Architecture of the detection model.

then their corresponding Temporal Graphs (i.e., $T^f(D^*[P]) or T^F(D^*[P])$), while, for any given test sample we follow the same procedure as to conclude with the computation of $\overline{\Delta}$-Similarity metric in order to measure the structural similarities between the graphs of the two objects.

Next, we describe the main process of determining if an unknown sample is malicious or benign based on the results of our similarity metrics when applied on the corresponding Temporal Graph of a test sample and a set of Temporal Graphs that represent known malicious software samples. In Figure 5.3 we depict the total architecture of our proposed model for detecting malicious software samples.

Implementation Aspects. In the example of Figure 5.3 we suppose we are given an unknown test sample τ that we do not know if it is malicious, and we are asked to decide if τ is malicious or benign. Having a database with the Temporal Graphs of known malware samples. Once the corresponding Temporal Graphs have been constructed, we compute the our similarity metrics between the Temporal Graph of τ and each Temporal Graph that represents a malware sample in our database. So, let S the total number of malware samples in our database, we result to S values in our measurements on our similarity metrics, one per pair (τ, S_i), where if the maximum value exhibited is above a predefined threshold λ it indicates that τ is malicious.

So, formalizing the above implementation of the proposed approach, firstly for each member of each malware family, it is computed the "median-vertices" represented on the Cartesian plane by the medians of in/out degrees and averaged in/out weights for each node. Next, we compute the $\overline{\delta}$-distance between these vertices and each of the corresponding vertices of the test sample's behavioral graph. Note that, in order to compute an similarity metric utilizing the Temporal Graphs, it is required to iterate over all the `epochs` of the segmented graph in order to compute the average similarity exhibited throughout the `epochs`. Finally, the distances are summed up and the $\overline{\Delta}$-Similarity is computed, where compared to a previously tuned threshold λ the test sample is distinguished as malicious or not.

Chapter 6

Classifying Malicious Samples

Next there is presented the utilization of the proposed similarity techniques for indexing malicious samples into known malware families.

6.1 Malware Classification computing SaMe and NP Similarity Metrics between Group Relation Graphs

Our classification method is based on the application of the proposed similarity metrics described above. More precisely, our method selects those families of $\mathcal{Q}$ that are most similar to sample T according to SaMe-similarity metric, calling them `SaMe-dominant` families and denoting this subset of $\mathcal{Q}$ as $\mathcal{D}_{SaMe}(\mathcal{Q})$. Next, we proceed by measuring the similarity between the malware sample T and all the families in the set $\mathcal{D}_{SaMe}(\mathcal{Q})$ produced in the previous step, computing the NP-similarity as to select, respectively, the `NP-dominant` family denoting this subset of $\mathcal{D}_{SaMe}(\mathcal{Q})$ as $\mathcal{D}_{NP}(\mathcal{D}_{SaMe}(\mathcal{Q}))$. This subset ($\mathcal{D}_{NP}(\mathcal{D}_{SaMe}(\mathcal{Q})) \subseteq \mathcal{D}_{SaMe}(\mathcal{Q}) \subseteq \mathcal{Q}$) is consisted by solely one malware family, let F_k, into which the malware sample T will be classified.

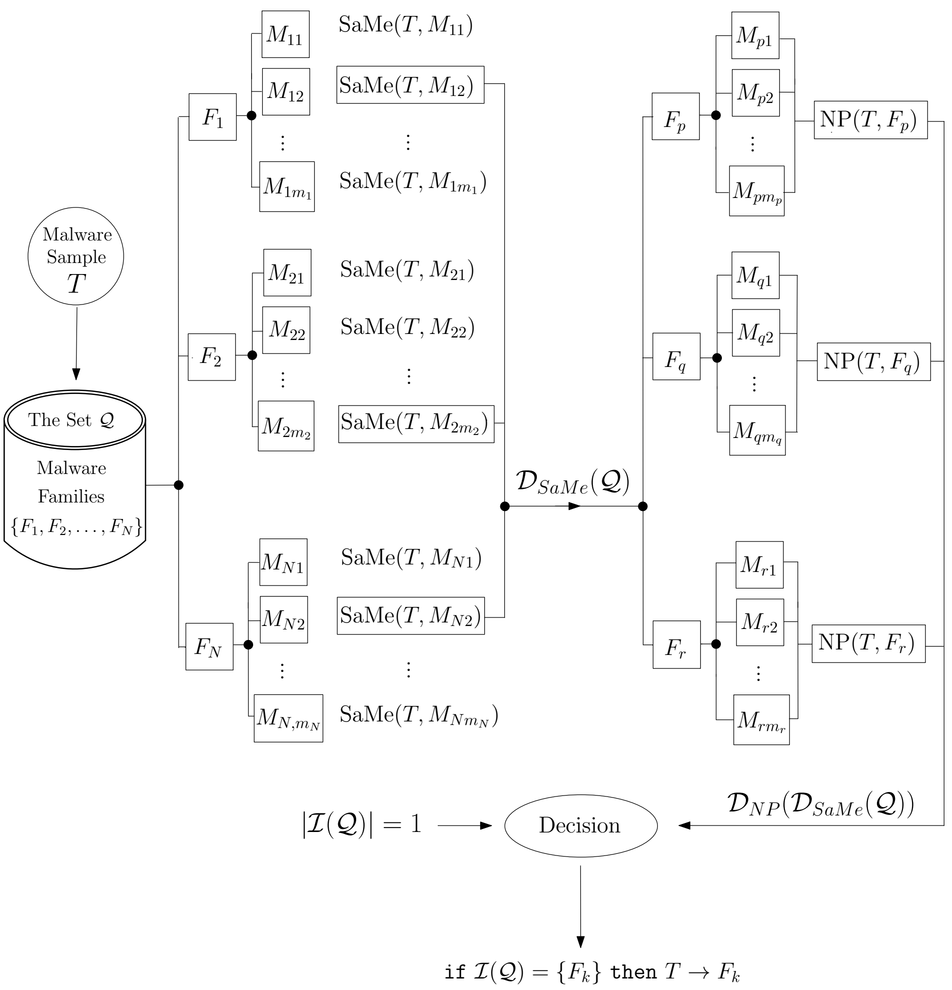


Figure 6.1: Architecture of the classification model computing the SaMe and NP similarity metrics between GrG graphs.

In Figure 6.1 we show a representation of the procedure for classifying an unknown test sample T to a known malware family utilizing the proposed methods (i.e., SaMe-similarity and NP-similarity). Our classification technique proceeds as follows: given a set $\mathcal{Q} = \{F_1, F_2, \ldots, F_N\}$ of known malware families and an unclassified malware sample T, we compute the SaMe-similarity between T and each family, keeping the maximum result (`representative`) for each family resulting to $\mathcal{Q}$ results, one per family. In this step, we select those families that their `representatives` are more similar to test sample T according to SaMe-similarity (`SaMe-dominant` families), consisting a subset of $\mathcal{Q}$ of families denoting it with $\mathcal{D}_{SaMe}(\mathcal{Q})$. Next,we compute the NP-similarity between T and each family of the set $\mathcal{D}_{SaMe}(\mathcal{Q})$. In this step, we select the family that is more similar to test sample T according to NP-similarity (`NP-dominant` family), consisting a subset of $\mathcal{D}_{SaMe}(\mathcal{Q})$ of families denoting it with $\mathcal{D}_{NP}(\mathcal{D}_{SaMe}(\mathcal{Q})$. The set that contains the family into which the test sample T will be classified is denoted by $\mathcal{I}(\mathcal{Q})$ containing this family. Finally, as we mentioned above, the set $\mathcal{D}_{NP}(\mathcal{D}_{SaMe}(\mathcal{Q})$ contains only one family let F_k, so, if $\mathcal{I}(\mathcal{Q}) = \{F_k\}$ then we classify test sample T to malware family F_k.

6.2 Malware Classification Computing $\overline{\Delta}$-Similarity between Temporal Graphs

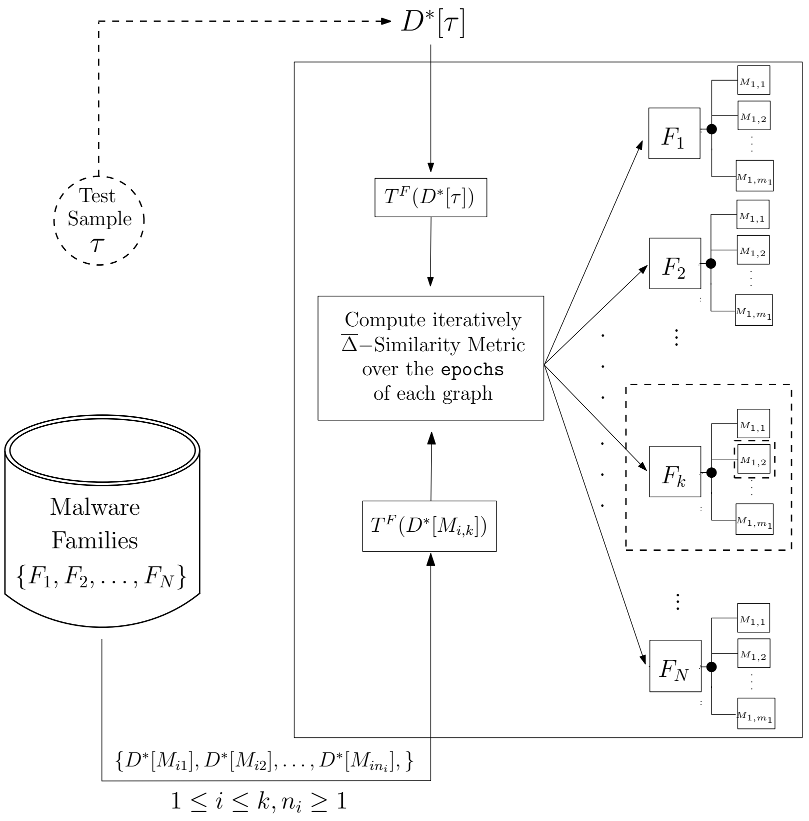
Next we discuss the operation of our proposed graph based malware classification model, and present an overview on its constructional principles alongside with a brief discussions over its implementation aspects.

6.2.1 Model Overview.

Our proposed method is based on application our proposed similarity metrics over the set of known malware families in order to classify on them an unclassified malware sample, let τ. More precisely, our method selects the family that is most similar to τ according to the similarity results exhibited by the measurement of $\overline{\Delta}$-Similarity metric, calling that family `dominant` family. More precisely, using our proposed similarity metrics, we iterate over all the members of all the known malware families measuring the similarity between each pair of τ, M_{ik}, where M_{ik} is the i^{th} member of the k^{th} malware family. Then, for each family we select a member that is the most similar to τ, according to $\overline{\Delta}$-Similarity metric, and denote this member as `representative sample` for this specific family. Finally, among all the `representative samples` for all the known malware families, we select to classify the unclassified test sample τ to the malware family that its `representative sample` exhibits the maximum similarity with τ according to $\overline{\Delta}$-Similarity metric, denoting this family as `dominant family`.

6.2.2 Implementation Aspects.

In the example of Figure 6.2 we show a representation of the procedure for classifying an unknown test sample τ to a known malware family utilizing the aforementioned methods (i.e., Δ-Similarity and Cover-Similarity metrics). More formally, our



$D^*[\tau]$
Test Sample τ
$T^F(D^*[\tau])$
Compute iteratively $\overline{\Delta}$−Similarity Metric over the epochs of each graph
Malware Families $\{F_1, F_2, \ldots, F_N\}$
$T^F(D^*[M_{i,k}])$
F_1
F_2
F_k
F_N
$M_{1,1}$
$M_{1,2}$
M_{1,m_1}
$\{D^*[M_{i1}], D^*[M_{i2}], \ldots, D^*[M_{in_i}],\}$
$1 \leq i \leq k, n_i \geq 1$

Figure 6.2: Architecture of the classification model computing the $\overline{\Delta}$ similarity metric between temporal instances of GrG Graphs.

classification technique proceeds as follows: given a set of known malware families $F_1, F_2, \ldots, F_N\}$ and an unclassified malware sample τ, we measure the Δ-Similarity and Cover-Similarity metrics over all the members of each family, keeping the maximum result (i.e., `representative sample`) for each family resulting to N results (i.e., N `representative samples`), one per family. Then, we classify the test sample to the family that exhibited the maximum value among all results. In other words, we compute the aforementioned similarity metrics between τ and all the malware families of the data-set, selecting as the `dominant family`, the one that has the `representative sample` that exhibits the maximum value in our similarity measurements.

So, formalizing the above implementation of the proposed approach, firstly for each member of each malware family, it is computed the "median-vertices" represented on the Cartesian plane by the medians of in/out degrees and averaged in/out weights for each node. Next, we compute the $\overline{\delta}$-distance between these vertices and each of the corresponding vertices of the test sample's behavioral graph. Note that, in order to compute an similarity metric utilizing the Temporal Graphs, it is required to iterate over all the `epochs` of the segmented graph in order to compute the average similarity exhibited throughout the `epochs`. Finally, the distances are summed up and the $\overline{\Delta}$-Similarity is computed, where the test sample is set to be closer to a member of a malicious family (i.e., `representative sample` and `dominant family`, respectively), where the `representative sample` that exhibits the major similarity according to $\overline{\Delta}$-Similarity indexes the test sample to its malware family.

Chapter 7

Spread Prevention

7.1 Background and Assumptions

Our proposed system, for investigating the effect of counter-measure's response time on malware's spread, is consisted by our propagation model that simulates the spread of a malware to proximal mobile devices, as well as the mobility model and its main principles concerning the motion of the devices in a city. Additionally, we incorporate a city-representation model to draw the town-planning through an $n \times m$ matrix consisted by 0s and 255s, where 0 denotes a point on a road of the city and 255 denotes any obstacle (e.g., buildings) produced by an image taken from Google Maps.

Our malware propagation model incorporates two of the most applied extensions of the SIR epidemic model, namely SIRpI and SIRpS epidemic models where each device can be either in `Susceptible`, `Infected` or `Repaired` (`Immunized`) state. On the other hand, our device mobility model generates traces, utilizing shortest path algorithms, for the mobile devices which are moving inside a city. Note that the city is represented by its image taken from Google Maps and its town-planning is modeled by an $n \times m$ matrix of 0s and 255s, where 0 denotes a point on a road while 255 denotes any obstacle. We utilize our malware propagation and device mobility models to develop a simulator that we use to study the spread of malware in mobile devices with respect to response-time of a counter-measure.

As far as the architecture of the proposed system, our main consideration is to model the user awareness and the capability of a security software to provide up to date and reasonably fast protection through response-time gradation. We implement and simulate the selected epidemic models SIRpI and SIRpS and, through a series of experiments, we investigate the behavior of the spread in each case, taking into account the presence of a counter-measure activated on its corresponding response time. Given an initially infected population, we perform a series of simulations for various response-time intervals, concerning also other factors which affect the malware's propagation, i.e., initially infected population, network density and malware's size, establishing upper bounds on the response-time needed by a counter-measure, such as a malware detector [7], in order to prevent pandemic. In other words, through our model we determine the maximum permitted time for a counter-measure to be activated when a specific percentage of the population is infected in order to guarantee that not all the susceptible devices in the city get infected and some (or, all) infected ones get sanitized. We finally present experimental results for the pandemic prevention provided by our simulations for various response-time intervals, where the contribution of this work extends to the interpretation of the provided results.

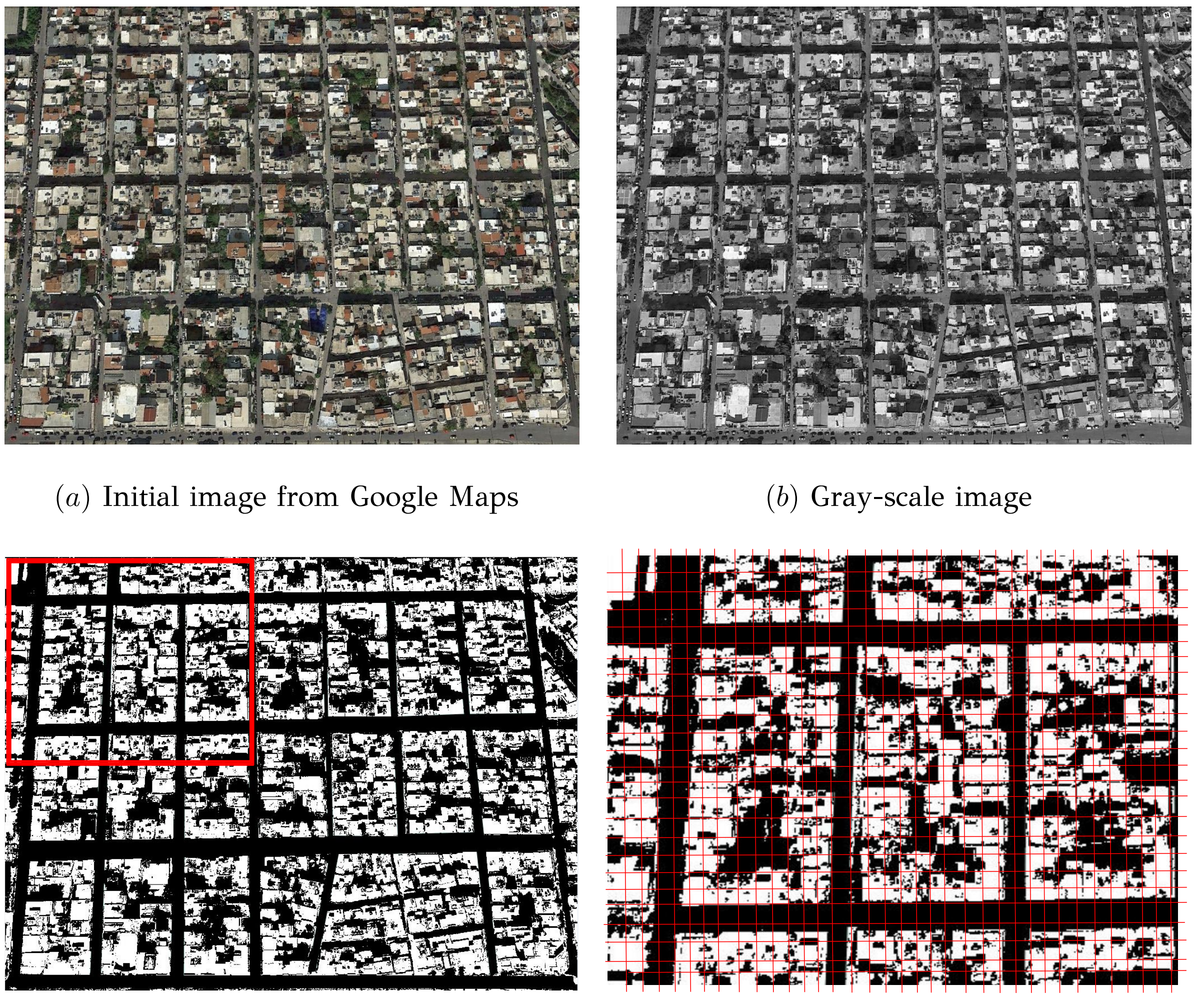
It is worth noting that, given any town's planning and an epidemic model that describes the spread, our model can efficiently simulate the procedure and consecutively establish for that case an upper bound on the maximum response-time permitted for a counter-measure to be activated in order to prevent the pandemic.

7.2 Simulating Town's Planning through Weighted Graphs

Our model simulates malware propagation to mobile devices that are changing their positions, or geological coordinates, according to a town-planing. In order to make our simulation closer to reality we used images of real towns-planning from Google Maps. We transform these images from RGB to gray-scale color system and then to black and white using an appropriate threshold. Hence, having an image of dimensions

$n \times m$ representing the town-planing of the city, we transform it to an $n \times m$ matrix M_{map} with values 0s and 255s, where 0 represent a free space (i.e., road) and 255 represents any obstacle (i.e., building). So, in our simulation, we permit a mobile device to move into a position with coordinates (x, y) if the corresponding cell (i, j) of matrix M_{map} has value 0. In Figure 7.1 we illustrate the construction of the town-planning representation.

In our model,we compute the attraction level of each cell based on its distance from a set of points that we call `Attraction` and we denote them with A (see, Figure7.2(c)). Hence the higher the distance of a cell from an attraction point A the lower its attraction level. For the non-zero cells of M_{map} it holds that its attraction level is computed by its closest attraction point A. More precisely, for a given city town-planning we define a set of points A_1, A_2, $\ldots$, A_k, called *attractions*, from which the attraction level ℓ_i of each map-cell u_i is computed with respect to its distance from the closest attraction A_j to this cell, $1 \leq i \leq |V(G_{map})|$ and $1 \leq j \leq k$; the less the distance the higher the attraction level of u_i map-cell. To represent the attraction level of each cell with value 0 in M_{map} matrix, we define intervals of values $[1 - 3], [4 - 7]$ and $[8 - 10]$ to depict cold-, warm-, or hot- spots reflecting the attraction level of its point according to the distance of each cell from its closest attraction point A. In our model, the range $[1 - 3]$ represents in descending order cold-spots, the range $[4 - 7]$ represents nearly warm-spots, while $[8 - 10]$ represents in ascending order hot-spots (i.e., the coldest spot has value 1 while the hottest one has value 10). So, if a cell has attraction level $w \in [1 - 3]$ (i.e., cold-spot) it is a distant cell according to its closest attraction point. Respectively, if $w \in [4 - 7]$ then the corresponding cell is in a medium distance from its closest attraction point nearly (i.e., warm-spots), while if $w \in [8 - 10]$ then the cell has a very close attraction point. In other words with this 10-value scale we represent the cold-spots with values near to 1, the warm-spots with values around 5, and the hot-spots with values near 10. Finally, having the attraction levels of each cell, we compute the weight w of an edge (u_i, u_j) in G_{map} is computed as $10 - \ell_{u_i} + 10 - \ell_{u_j} + 1$, where ℓ_{u_i} and ℓ_{u_j} are the attraction levels of nodes u_i and u_j, respectively.



(a) Initial image from Google Maps

(b) Gray-scale image

(c) Black and White image

(d) Grid-view of the Black and White image

Figure 7.1: (a) – (c) The transformation of a Google Maps image to a Black and White matrix; (d) Zoom-in to a part of a Black and White image.

7.3 Modeling the Transporting of Mobile Nodes Through Shortest Paths Algorithms

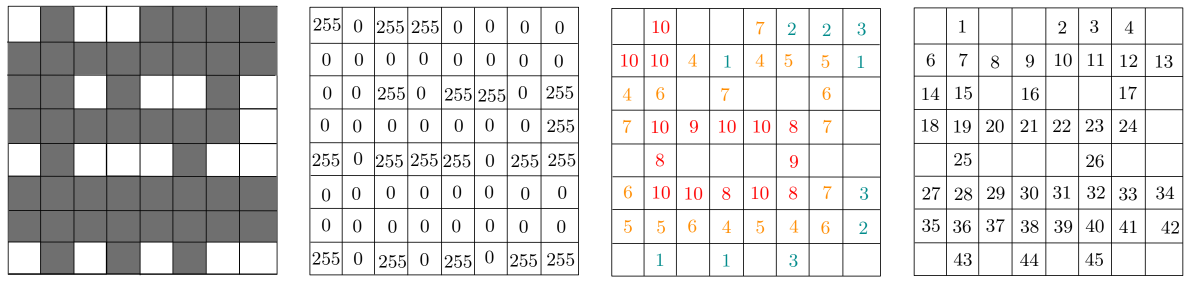
In our model, we simulate the movements of a mobile device by changing the coordinates of a node taking into account the corresponding cells (i, j) in the M_{map} matrix with values 0 and 255 that represents the town-planning. More precisely, we permit a device to move on a point in the map if the corresponding cell (i, j) in M_{map} matrix has value 0 since such a cell represents a road or, equivalently, we do not allow a device to move on a cell with value 255 since it represents any obstacle such as building. To make our simulation more realistic, we propose and implement a trace generator for device mobility, that is, for each node we generate a trace between an initial position and a target position. In particular, we set each node at a pre-defined point with coordinates (i,j) on the grid and then a destination point (i',j') is assigned on that as to be reached through a path computed by a shortest path algorithm computed on the weighted directed graph we define next, and which we shall call it G_{map}.

Definition 7.1. The weighted directed graph G_{map} represents the town's planning using its M_{map} representation; it is constructed as follows:

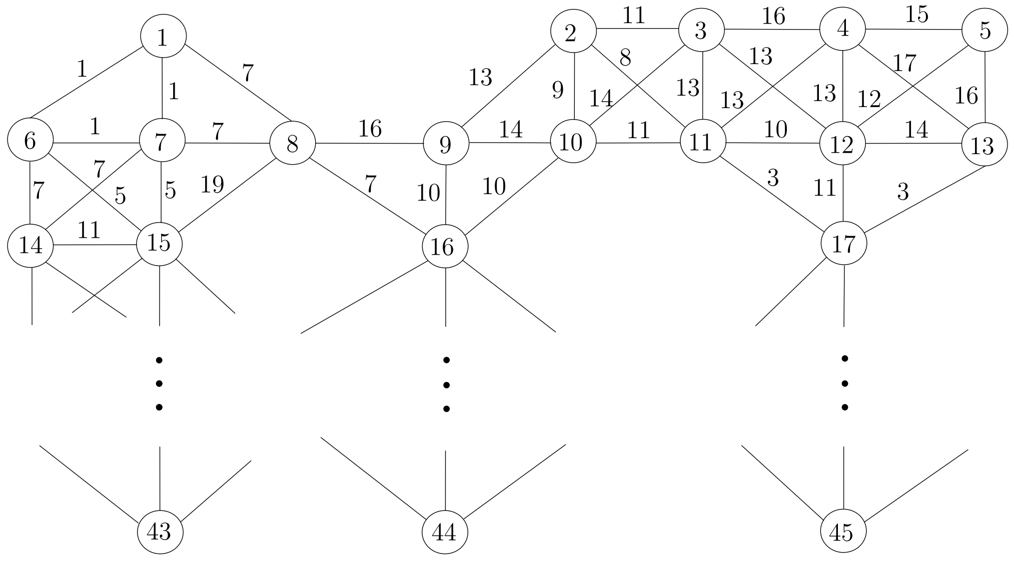
- its vertices $V(G_{map})$ correspond to the cells of matrix M_{map} with value 0, and
- two vertices are joined by an edge if their corresponding cells with value 0 are adjacent in the M_{map} matrix; note that, a cell (i,j), with value 0 in the M_{map} matrix, is adjacent to every cell with value 0 in its 8-neighborhood.

In Figure 7.2, we show in detail the construction of the graph G_{map} from the M_{map} matrix. In particular, in Figures 7.2(a) and 7.2(b) we show the black/white representation of an example map representing each point of a road with value 0 (black) and any obstacle (building) with value 255 (white), in Figure 7.2(c) we show the attraction level matrix which is constructed by assigning to its cells values in the range $[1,10]$ depicting the `cold-`, `warm-` and `hot- spot` of the city, in Figure 7.2(d) we assign an ID on each point, indicating a node on the G_{map} graph, while in Figure 7.2(e) we present the resulting graph G_{map}.

The proposed device mobility model is developed in such a way to reflect in some fashion the behavioral motion of pedestrians. More precisely, by the start of each experiment we define for each mobile device a start-destination set of points. Posed initially at random they try to reach the destination following a shortest path between them preserving the pass of specific points of the city with different attraction level (in our terminology: `cold-`, `warm-` and `hot-spots`). Once a mobile device reaches the assigned destination a new one is randomly assigned, and so on, until the end of the experiment. The reasons we chosen to approximate the selection process of destination points, are indicated in the fact that while mobile devices have a destination assigned to be in the boundary areas of the city (not the one they are located at that time) the probability to cross the center of the city area is increased. Scaling up the view, we observe that the orbits of all devices is more probable to be crossed around the city area's center. A congestion occurs around the city center increasing the probability of any pair of nodes to appear close enough to probe the malware.



(a) The matrix M_{map} (b) M_{map} in $[0, 255]$ (c) Attraction levels (d) ID assignment



(e) The graph G_{map}

Figure 7.2: Map representation and undirected weighted graph construction.

To this point, it is worth noting to refer that the destination points are not randomly assigned as for each device we select a destination that is located to the `NW`, `N`, `NE`, `W`, `E`, `SW`, `S`, `SE` boundaries of the map, where once a mobile device reaches the assigned destination, then a new destination point is assigned and thus we guarantee that always the devices change their positions.

Finally, deepen into the relation between the graphs G_{map} and G_{dev} we can claim that the structure of G_{dev} strongly dependents on the structure of G_{map}. The relation between these two graphs relies on the property that the density of G_{dev} (by means of sparse or dense graphs) is affected by the cardinalities of vertex sets $V(G_{dev})$ and $V(G_{map})$, as the cardinality of edge set $E(G_{dev})$, which determines the density of G_{dev}, is inversely analogous to cardinality of $V(G_{map})$.

Definition 7.2. For a given set of mobile devices, say, dev, that are moving inside a city represented by its corresponding G_{map} we define the density of this network (i.e., G_{dev}), denoting it with D as follows:

$$D(G_{dev}, G_{map}) = \frac{|V(G_{dev})|}{|V(G_{map})|}, \tag{7.1}$$

where $|V(G_{dev})| = |I| + |S|$ correspond to the number of infected and susceptible devices respectively, and from which it follows that for a given number of devices, say, n, the higher the cardinality of $|V(G_{map})|$, the less the density of G_{dev}.

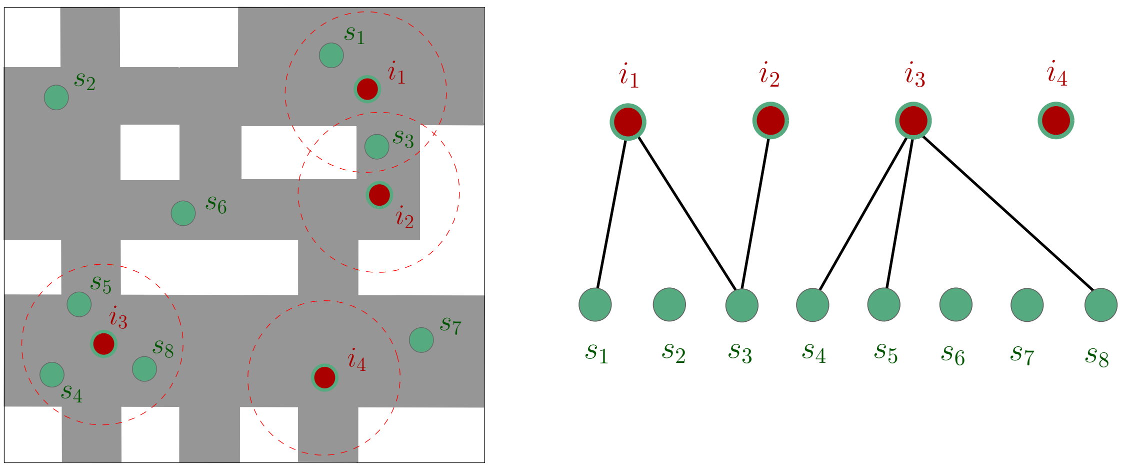
7.4 Implementing the Epidemic Models

Focusing on investigating the effect of counter-measure's response time on proximity malware's spread between mobile devices, we decompose the requirements of the modeled settings concerning mainly the malware propagation procedure as also the behavioral characteristics of the network formed among the mobile devices. Additionally, the proposed malware propagation model takes into account factors that affect the spread such as the range of mobile devices, the size of the spreading malware, and the velocity of mobile devices.

In order to simulate malware's propagation to proximal mobile devices, we take into account that the underlying network is formed on-the-fly between the devices while they are moving inside the city. This network can be represented by an undirected graph (we shall denote it G_{dev} throughout the paper) that is modifying its structure (i.e., topology by means of edge creations and deletion among its nodes - devices). So, we could claim that during a specific period of time, let $[t_1, t_n]$ this graph can be referenced by its structurally different instances as $G^1_{dev}, G^2_{dev}, ..., G^n_{dev}$.

Definition 7.3. We define $G_{dev} = (I, S, E)$ to be a bipartite graph whose vertices correspond to the devices of the network and an edge between two vertices occur if their corresponding devices have distance less than r at time t.

In Figure 7.3 we illustrate the process of constructing the bipartite graph G_{dev}. In Figure 7.3(a) we depict how the mobile devices are moving inside a city represented



(a) Probing devices (b) The graph G_{dev}

Figure 7.3: Mobile devices moving inside a city represented by its M_{map} matrix, constructing the bipartite graph G_{dev} by the links formed among them.

by its M_{map} matrix; recall that cells with value 0 correspond to roads while cells with value 255 correspond to obstacles. The circles around the mobile devices show the range of them while their colors, blue or red, correspond to transmissions by susceptible or infected devices respectively. Then, in Figure 7.3(b), we illustrate how we create the bipartite graph G_{dev}: for any pair of mobile devices (i.e., vertices in the G_{dev}), we add an edge between them in G_{dev} if their distance is less than r. Concerning the above network represented by G_{dev}, next we provide some definitions about its characteristics.

Definition 7.4. Let $\tau_1, \tau_2, ..., \tau_k$ be the k states of a given epidemic model $\mathcal{M}$. We define the *state-cover* $C(\tau^*)$ of state $\tau^* \in \{\tau_1, \tau_2, ..., \tau_k\}$ as the rate of the objects (i.e., mobile devices) that are in state τ^* by the sum of the objects that are in any state of the epidemic model $\mathcal{M}$, that is

$$C(\tau^*) = \frac{|\tau^*|}{\sum_{i=1}^{k} |\tau_i|}. \tag{7.2}$$

In our model a state $\tau_i \in \{S, I, Rp, Im\}$. Additionally, for the `infected` state I and the `susceptible` state S, the *Infected-cover* and the *Susceptible-cover* are defined as follows:

$$C(I) = \frac{|I|}{|S| + |I| + |Rp| + |Im|} \quad \text{and} \quad C(S) = \frac{|S|}{|S| + |I| + |Rp| + |Im|}, \tag{7.3}$$

respectively, where $|Rp|$ is the number of devices in the `repaired` state Rp of our model.

Definition 7.5. Let $\tau_1^*, \tau_2^* \in \{\tau_1, \tau_2, ..., \tau_k\}$ be two states of a given epidemic model E. We define the *state-rate* $R(\tau_1^* \tau_2^*)$ of states τ_1^* and τ_2^* as the rate of the number of the objects belonging to τ_1^* state over the number of objects that belong to τ_2^* state, that is:

$$R(\tau_1^*, \tau_2^*) = \frac{C(\tau_1^*)}{C(\tau_2^*) + 1}, \tag{7.4}$$

where the addition of 1 in the denominator avoids a division with 0. By definition it follows that $R(\tau_1^*, \tau_2^*) = \dfrac{|\tau_1^*|}{|\tau_2^*| + 1}$.

The state-rates *IS-rate* and *SI-rate* referring the `infected` state I and the `susceptible` state S, respectively, are defined as follows:

$$R(I, S) = \frac{|I|}{|S| + 1} \quad \text{and} \quad R(S, I) = \frac{|S|}{|I| + 1}. \tag{7.5}$$

In our model, in order to simulate the propagation of malware we allow the activation of a counter-measure to remove the malicious software from the infected devices. This feature is adapted to our model in order to remove, or clean in some fashion, the malicious software from the device. Specifically, in our model, if a device remains for a period of time (i.e., simulation steps demanded for the transmission of all packets of the malware) within a specific radius from an infected device then it gets infected too. However, if this device moves out of range then it would need more time, in terms of simulation steps, in order to get infected. Upon the activation of a counter-measure, it sanitizes the device by removing the malware. However, different epidemic settings provide the counter-measure with the ability to immunize or not the infected mobile device.

Concerning the practical infection, a susceptible mobile device gets infected, when the entire malicious software is located in it. On the implementation of our model we abstract the constructive representation of malicious software that is partitioned to packets in order to be transmitted. By design, we refer to a packet as the unit that could represent any given number of PDU packets. So, in our model a susceptible mobile device is defined as infected when has caught by its nearby mobile devices all the packet units of the spreading malware.

Based on the property that if the pandemic is prevented for a given number n of packets, then it is also prevented for any number $m > n$; this is not always true for $m < n$. Following an opposite argument, it follows that if the pandemic is not prevented for a given number of packets n, then it is also not prevented for any number $l < n$; similarly, this is not always true for $l > n$. Since we are interested in investigating the spread of a proximity malware in its worst case scenarios we start by simulation an as small as possible one, and then to explore its propagation behavior by increasing its size.

On the other hand, regarding the repair process that depends on the implementation of the epidemic models utilized (i.e., SIRpI or SIRpS epidemic models), the counter-measure's reaction in our proposed model removes at once all the packet units consisting the malware from the infected mobile devices. Particularly for the SIRpI epidemic model, an after-repair immunization process guarantees that the devices is no longer susceptible to the spreading malware. To this end we ought to make clear that the counter-measure's reaction refers to the process of its activation after a period from the time where the susceptible device gets infected, while the period of time need by a counter-measure from the time a susceptible device is considered as infected until the start of the repair process is called Response-Time (throughout the paper we shall denote it as R_t).

Chapter 8

Evaluation

8.1 Experimental Setup for Malware Detection and Classification

In order to evaluate our proposed malware detection and classification techniques we use a data-set of 2631 malware samples from 48 malware families, each containing from 3 to 317 malware samples, and also a set of 35 benign samples that cover a wide variety of commodity software types including editors, office suites, media players, etc. To this point, it is of major importance to mention that, for evaluation purposes, we use the same data-set of malicious and benign ScDG graph samples as in [5] transforming each sample's ScDG graph $D[P]$ into GrG graph $D^*[P]$, based on the grouping of system-calls as described in the corresponding chapter. The set $\mathcal{Q}$ of the 48 malware families along with their sizes (i.e., number of members) are listed in Table 8.1, note that the abbreviations DNSCGR stands for *DNSChanger* and OLG stands for *OnLineGames*. On the other hand, in Table 8.2 we present the set of benign software samples used for the evaluation of the detection phase, concerning the false positive rates. Additionally, we ought to notice that even the number of the benign samples is not comparable to the number of malware samples, the diversity of them covers the full range of the commodity software products can be deployed in any computer. So, in other words, we can say that we use 48 malware families

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

Table 8.1: The set $\mathcal{Q}$ of the 48 malware families $F_1, F_2, \ldots, F_{48}$ provided by Babic, along with their sizes (i.e., number of members), as used in [5]

with various number of members in each one, against 35 benign families with one member per family.

Benign Software Products			
Adobe_Reader	Freecell	MSN_Messenger	Skype
Apple_Software_Update	Freeciv-(server)	Netcat_port_listen-(scan)	Solitaire
Autoruns	GIMP	NetHack	Sys_information
Battle_for_Wesnoth	Google_Earth	Notepad-WordPad	Task_Manager
Chrome	Hello_world	OpenOffice_Writer	Tux_Racer
Chrome_Setup	Internet_Explorer	Outlook_Express	uTorrent
Copy_to_sys_folder	iTunes	ping	VLC
Firefox	Minesweeper	Self_extracting_archive	Media_Player

Table 8.2: The set of benign software samples used for the evaluation of false positive of the detection phase, as used in [5]

Finally, before we proceed with the presentation of the detection and classification results achieved by our model, we ought to clarify some issues concerning the data-set used for evaluation and the presentation of results achieved by other detection or classification models. As we will discuss later in this section, we present various results concerning the detection rates and the classification accuracy from various graph-based and non graph-based models that implement different algorithms and use different data-set for their evaluation. Hence, a straight comparison between our model and other detection or classification models (noting that they implement different algorithms and evaluate with different data-sets) is not applicable, we just present the results achieved by these models and provide a discussion on them. However, we ought to notice that in the cases of [6] and [5] where the data-set used for the evaluation of the detection rates is the same with the one used in this work we can proceed with a straight comparison. So, summarizing, next in 8.1 we present the detection results of our model, and compare them against [6] and [5] which use the same data-set (see, Table8.3).

8.1.1 Detection Results over the Proposed Techniques

Next there are presented the experimental results exhibited when evaluating the detection accuracy of the proposed model. The detection results achieved by the proposed model, utilizing the proposed similarity metrics (i.e., Delta-Similarity and Cover Similarity metrics) and the graph-based representations (i.e., Group Relation Graph and Coverage Graph) of the software samples.

Detection Ratio over GrG and Δ-Similarity

In order to evaluate the potentials of our model concerning its ability to distinguish a GrG graph that represents a malware sample from a GrG graph that represents a benign one, we perform 5-fold cross validation utilizing the data set we described above. Additionally, we set up the detection threshold $\lambda = 0.97$, the parameters $\alpha = 1$ and $\beta = 0$, while $\Gamma = 10^3$, in order to maximize the ratio of true-positives by the false-positives. We next present our results after performing a set of 5-fold cross validation experiments partitioning the data set described above into 5 buckets, using in each experiment one bucket as test-set and the other four as train-set. Our results provide us with the performance of the detection rates or, equivalently, true-positives (TP) and

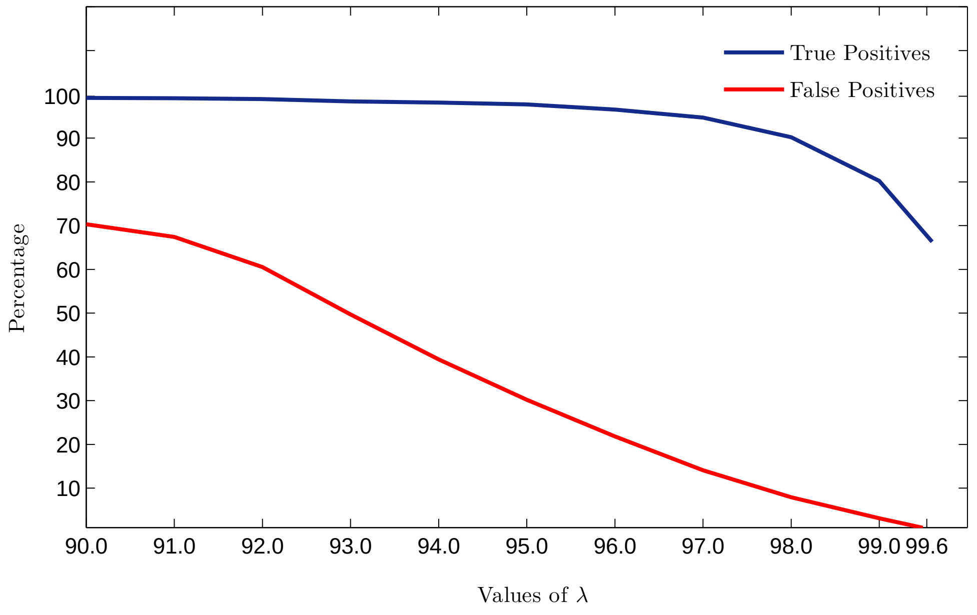


Figure 8.1: Detection rate and false-positives for multiple values of λ archived by applying Δ-similarity.

the corresponding false-positives (FP) of our proposed model according to various values of threshold λ as we described previously; see, Figure 8.1 and Figure 8.2. Note that due to the 5-fold cross validation process the percentage values below are averaged over the 5-folds.

In our evaluation experiments, in order to maximize the ratio of true-positives by the false-positives, we tune the detection threshold $\lambda = 0.97$ and the parameters $\alpha = 1$ and $\beta = 0$, while $\Gamma = 10^3$. Hence, our model, among the various values depending on the aforementioned tuning (see, Figure 8.1 and Figure 8.2), achieved a 94.70% detection rate with 13.10% false positives; see, Figure 8.1 and Figure 8.2.
In Table 8.3, we present the results from other works presenting similar graph-based models for malware detection and proceed with a discussion about the true-positives and false-positives achieved by these models. More precisely, the first column refers to the result's host, the second one refers to the utilized technique, while the third and fourth columns refer to the true-positive and false-positive rates, respectively.

Fredrikson *et al.* [6] 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, it can efficiently distinguish malicious

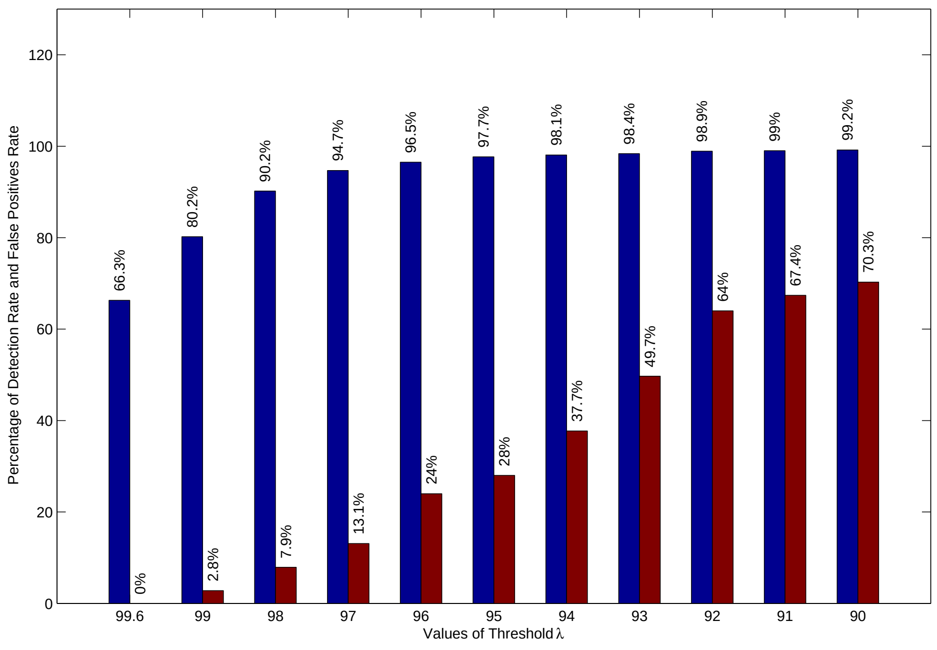


Figure 8.2: The true-positive and false-positive rates achieved for multiple values of λ when applying Δ-similarity.

In:	Technique	True Positives	False Positives
[6]	Graph Mining (ScD)	92.40 %	06.10%
[5]	Tree Automata Inference (ScD)	80.00%	05.00%
[this paper]	Δ-similarity (GrG graphs)	94.70%	13.10%

Table 8.3: Detection results; note that [this paper] uses the same dataset as[6] and [5].

from benign programs. The proposed technique's results range from an 86.5% detection rate with 0 false-positives to a detection rate of 99.4% which however exhibits higher false-positives (57.14%). However,someone can observe that our model reaches the detection rates of the proposed model presented in [6], proving its potentials for further improvements.

Babic *et al.* [5] achieved the malware detection by k-testable tree automata inference from system-call data flow dependence graphs. To this point we ought to underline that in this work the authors use the same data-set that we used in our work, provided by Babic. Thus, this work provides a fair instance to compare our model's results. However, while Babic *et al.* perform 2-fold cross validation using the first half of data-set as train-set and the second one as test-set, we perform 5-fold cross validation. Comparing the results exhibited in [5] with ours, easily we can claim that our proposed model is quite competitive especially for specific values of λ (i.e., 0.99) achieving less false positives for the same detection rate.

Detection Ratio over CvG and Cover-Similarity

In this section we present our experimental setup and discuss the performance of our proposed graph-based model for malware detection while altering the approach of measuring the weight on each vertex (i.e., by the sum or by the mean value of the weights). Then, we describe how we divide our data set into train-set and test-set and how we tune our threshold parameters according to feedback taken by a series of experiments to evaluate our model's potentials on detecting malicious software samples using the Cover-Similarity.

Through the evaluation of our proposed graph-based model for malware detection we use a data-set of 2631 malware samples from 48 distinct malware families, and also a set of 35 benign samples that cover a wide range of commodity software types. Additionally, it is of major importance to mention that the data-set is the one provided by D. Babic and also used in [5], transforming each sample's ScDG into GrG and then to CvG, based on the methodology described in the previous section. We implemented our proposed model in Java and performed the experiments on a commodity computer with 4-th generation Intel i3 with 4Gb of RAM detecting almost 600 test samples in less than 10 minutes.

We next present our results after performing a set of 5-fold cross validation experiments partitioning the data-set into 5 compartments, using in each experiment 80% of samples as training set and 20% as test set. Our results provide us with the performance of the detection accuracy or, equivalently, true-positives rates (TP) and the corresponding false-positives rates (FP).

In Figure 8.3 we show our detection results, when measuring the weight of each vertex by summing all the weights on it, for various threshold values. Note that due to the 5-fold cross validation process the percentage values below are averaged over the k-folds, (i.e., 5 folds in our experiments). Our experimental results show that, despite that the FP rates where not behaved as expected, achieving less than 50% detection accuracy for 0% FP, our proposed graph-based model for malware detection exhibited

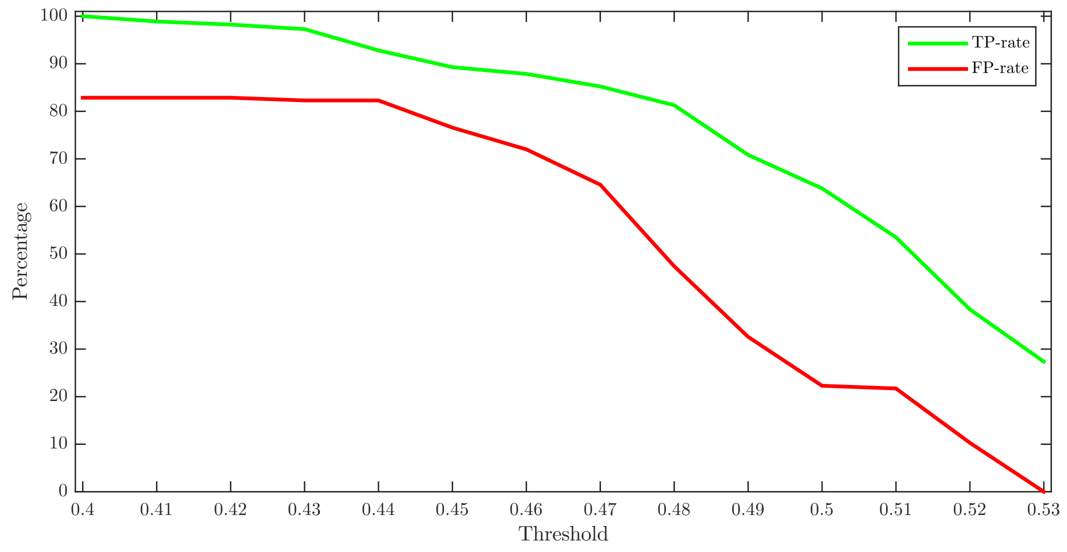


Figure 8.3: True-positive rate and false-positive rates for multiple values of threshold archived by applying the sum on domination measuring.

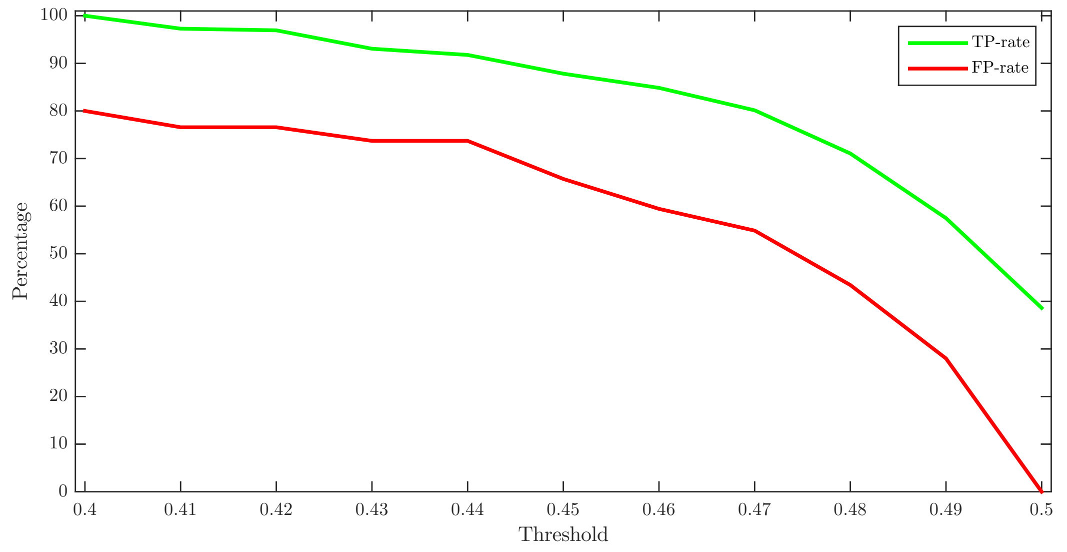


Figure 8.4: True-positive rate and false-positive rates for multiple values of threshold archived by applying the mean on domination measuring.

high rates of detection ability, with TP rates ranging from 27.3% for threshold value 0.53 to 100% for threshold value 0.4.

In Figure 8.4 we show our detection results, when measuring the weight of each vertex by computing the mean value of all the weights on it over its degree, for various threshold values. Note that due to the 5-fold cross validation process the percentage values below are averaged over the k-folds, (i.e., 5 folds in our experiments). The measuring of weight by its mean exhibited a slightly different behavior on the detection ability of our proposed graph-based model for malware detection. Our experimental results show that, that the FP rates have been reduced where the detection accuracy for 0% FP has been increased. Our model exhibited high rates of detection ability, with TP rates ranging from almost 40% for threshold value 0.5 to 100% for threshold value 0.4.

8.1.2 Classification Results over the Proposed Techniques

Next there are presented the experimental results exhibited when evaluating the classification accuracy of the proposed model. The classification results achieved by the proposed model, utilizing the proposed similarity metrics (i.e., SaMe and NP) and the graph-based representation (i.e., Group Relation Graph) of the software samples.

Classification Accuracy over GrG and SaMe and NP Similarity Metrics

In order to evaluate our proposed malware classification model we use the same data-set of 2631 malware samples pre-classified into 48 malware families F_1, F_2, ..., F_{48} of the set $\mathcal{Q}$. We evaluate the classification accuracy of our model performing a set of 5-fold cross validation experiments partitioning the data set described above into 5 buckets (i.e., 20% test-set and 80% train-set). Evaluating our model's classification rates we defined three types of correct classification, the so called `partial matching`, `directed matching` and `exact matching`. So, in case we are given test of malware family (x, y) and our model classifies it to the family (a, b), then if $(a = x) \vee (a = y) \vee (b = x) \vee (b = y)$ the `partial matching` returns a successful classification, if $(a = x) \vee (b = y)$ the `directed matching` returns a successful classification, and if $(a = x) \wedge (b = y)$ the `exact matching` return a successful classification.

Evaluating our model's classification accuracy, our model achieved a mean of 82.39% correct classifications with directed matching, a mean of 69.28% correct classifications with exact matching, and a mean of 83.42% correct classifications with partial matching, classifying each bucket of 526 malware samples in 500 seconds, performing the experiments on a commodity desktop equipped with an Intel core i3 processor at 3.2 GHz and 4 GB of RAM. In Table 8.4 we provide a straight comparison of the results achieved by our model for classifying malware samples to malware families using the

Similarity Metric:	Exact	Direct	Partial
SaMe and NP	69.28%	82.39%	83.42%
Bray-Curtis	65.83%	80.42%	82.15%
Cossine	65.26%	79.08%	80.81%
Jaccard	52.90%	71.40%	74.09%
Test.to.Family Cover	20.15%	22.60%	27.26%
Family.to.Test Cover	02.05%	06.72%	11.71%

Table 8.4: Classification results from SaMe-NP similarity compared to those achieved by each single similarity metric.

SaMe- and NP-similarity metrics, to the results achieved by our model using single similarity metrics without combining them as to test the similarity concerning different types of characteristics (i.e., relational, quantitative and qualitative). Observing Table 8.4, it is obvious that the combination of various similarity metrics that take into account different types of characteristics, when utilized by our model achieves higher classification results.

8.2 Preventing Malware Pandemics

In the framework of our work, we are interested in investigating the effect of the time a counter-measure needs to be activated after the infection (i.e., response-time) on the malware's propagation and also, in a second level, how other factors such as the size of a malware, the density of the network and the initial infected population affect the spread of malware. The experiments are organized into two categories developed over two axes, concerning the SIRpI and SIRpS epidemic models. The experiments are categorized into two categories, where the first contains experiments oriented to the effect caused by the response-time intervals, note that throughout the paper we shall denote them as R_t-intervals, and the second contains a sub-set of the previous experiments where we have modified other factors as the size of the initial infected population and the density of the formed network. Regarding the simulation environment, we implemented the whole system, including the city-representation model and the malware-propagation and device mobility models, in Matlab.

8.2.1 Experimental Design over the Series of Simulations

In our experiments, we utilize the image of a city taken from Google Maps, transforming it to a black and white matrix, as we described in the previous section. Within this approach, we assign weights (attraction levels) to each cell (i, j) with value 0; recall that, cells with such values represent points on a road. By the start of the simulation the mobile devices are set randomly to any point in the city area, with each point having equal probability to host a mobile device. Then, a destination point is assigned to each device as to be reached following the proposed mobility model computing shortest paths for a set of start-destination points utilizing a shortest path algorithm.

Due to the probabilistic aspect of our device mobility model concerning the target-point assignment, we performed a series of experiments for each case of settings and plot the mean values for each simulation step. More precisely, due to the stochastic behavior of mobile device paths, in each experiment we lock in some fashion the attraction points A and hence the attraction level of each cell and perform each experiment several times with the same setting in order to present statistically our results.

Next we present results for a series of experiments for malware consisted by 3 and 6 packets for various response time intervals and *IS-rate* $R(I, S) = 0.25$ having an initially infected population consisted by 20 devices and initial susceptible population consisted by 80 devices, i.e., $I = 20$ and $S = 80$. So, next, we present the results achieved by our simulator implementing the proposed models for malware propagation and device mobility, where in sub-sections $3.2 - 3.3$ and $3.4 - 3.5$ we discuss the experimental results for pandemic prevention on different response time intervals and other factors that affect the spread for the SIRpI and SIRpS epidemic models respectively.

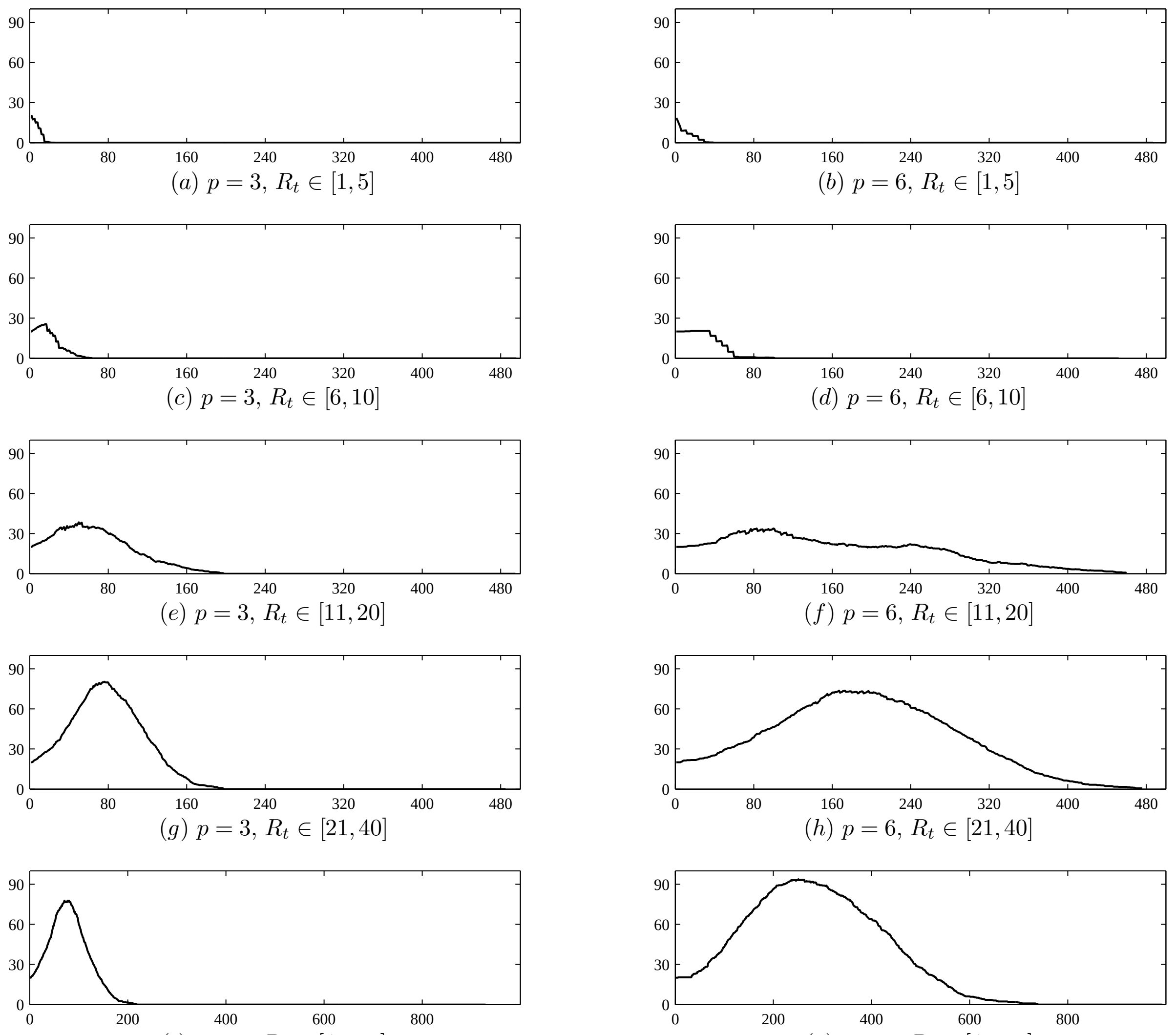
In the two categories of experiments presented, we study different intervals of values concerning the response-time of a counter-measure; note that, not all the devices have the same response-time but on each one is assigned a response-time of the interval under consideration. Specifically, we are interested in exploring the the effect of R_t in pandemic prevention and the behavior of malware's spread concerning both the early and late activated counter-measures and the heterogeneity among a wide range of different types of divides. To this end, we selected five response-time intervals, i.e., $[1, 5]$, $[6, 10]$, $[11, 20]$, $[21, 40]$ and $[41, 80]$, covering cases from early activated counter-measures applied on devices with low heterogeneity to late ones applied on devices with high heterogeneity. So, in each experiment a value R_t is uniformly selected from a response-time interval $[a, b]$ and assigned on each device. In our experiments where, e.g., $R_t \in [1, 5]$, each device has a counter-measure which is activated after a period (i.e. simulation steps passed after its infection) that is between 1 and 5 simulation steps.

Additionally, using the same categories of experiments we investigate the effect of the malware size, expressed in packets, and show how it can affect the result on both categories. In both categories of experiments, we perform a set of simulations on malware spread between moving devices in a city region. In the first category we keep the same ratio of the initial infected population and susceptible devices varying the counter-measure response time, while in the second one we increase the initial infected population and the network's density.

Throughout the paper, we shall denote the initial infected population by I, the susceptible devices by S, the counter-measure response time by R_t and the malware's size by p. To this point we ought to notice that the $R_t = t$ does not actually corresponds to t simulation steps but in the case where the size p of the malwere (in terms of packs) is greater than 1 it holds $R_t = t \times p$, that means, the infected device will propagate t times the full malware, and thus $t \times p$ simulation steps are required before its counter-measure has been activated. In Figures 8.5, 8.6, 8.7, 8.8, 8.9 and 8.10 the $x - axis$ refers to the simulation steps taken up to the end of simulation, while the $y - axis$ refers to the number of infected devices.

Preventing Pandemics that follow the SIRpI Epidemic Model

Pandemic Prevention for Various R_t-intervals As we can observe in Figures 8.5(a) and 8.5(b), the size of the spreading malware does not affect the spread at all since the response time is low. That means, the propagation of malware to neighboring susceptible devices fails, despite its size, due to the early activation of the counter-measure. However, the duplication of the size of the malware leads to the duplication on the time required for all the susceptible devices in the city to avoid the infection and all infected ones get sanitized. Moreover, increasing the counter-measure response time inside the same order of magnitude, we observe that still there is no significant increase on the number of the infected population, (see, Figures 8.5(c) – 8.5(d) and 8.5(e) – 8.5(f)) where the pandemic has been prevented due to the early activation



(a) $p = 3$, $R_t \in [1,5]$ (b) $p = 6$, $R_t \in [1,5]$

(c) $p = 3$, $R_t \in [6,10]$ (d) $p = 6$, $R_t \in [6,10]$

(e) $p = 3$, $R_t \in [11,20]$ (f) $p = 6$, $R_t \in [11,20]$

(g) $p = 3$, $R_t \in [21,40]$ (h) $p = 6$, $R_t \in [21,40]$

(i) $p = 3$, $R_t \in [41,80]$ (j) $p = 6$, $R_t \in [41,80]$

Figure 8.5: Simulation experiments implementing the properties of SIRpI epidemic model, for different values of malware packets (p), on a network with $R(I,S) = 0.25$ for various counter-measure response-time intervals (R_t).

of counter-measure on rational response-time intervals.

Leaving the rest parameters unchanged and increasing only the response-time of the counter-measure, we observe a global maximum of the spread in both experiments (see, Figures 8.5(g) – 8.5(h) and Figures 8.5(i) – 8.5(j)) where on all cases the counter-measure's activation in such larger response time intervals nearly avoids pandemic. In these cases the maxima are due to the multiplication of the response-time R_t by a factor f that causes the infected devices to propagate the full malware f times more than in the previous experiments. However in all cases the malware failed to spread to all the population since for such settings the counter-measure is activated adequately early preventing successfully the pandemic. Finally, in Figure 8.5(j) we can observe that further increase on the response-time interval, could lead to a pandemic, as nearly 90% of all susceptible devices have been infected.

Other Results on SIRpI Next we present results for a series of experiments for malware consisted by 3 and 6 packets for various response time intervals for pandemic prevention of an epidemic that follows the properties of the SIRpI epidemic model. We change some factors concerning the characteristics of the network (i.e., G_{dev}) and firstly perform a series of experiments for a different *IS-rate* $R(I,S) = 0.66$ having an initially infected population consisted by 40 devices and initial susceptible population consisted by 60 devices, i.e., $I = 40$ and $S = 60$, while then we increase the density D of our network by duplicating the number of devices keeping the *IS-rate* equal to that of Figure 8.5; recall that, $R(I,S) = 0.25$, where $I = 40$ and $S = 160$.

The results of the second category of our experiments are depicted in Figures 8.6 and 8.7. In this category, we modify the experiments of Figures 8.5(a) – 8.5(b), 8.5(e) – 8.5(f) and 8.5(i) – 8.5(j), changing the ratio I/S from 0.25 to 0.66 (see, Figure 8.6) and the density D of the network by duplicating the number of devices (see, Figure 8.7); recall that, I and S denote the numbers of the initial infected and susceptible population, respectively.

It is rational to expect that in presence of an early activated counter-measure (i.e., response-time intervals close to 0) a quick response is crucial for the immediate repression of malware's propagation. So, response-time intervals close to 0, act the same for pandemic prevention despite the size of the initial infected population, as shown in the results contrasting Figures 8.6(a) and 8.6(b) with Figures 8.5(a) and 8.5(b), where the prevention of a pandemic needs almost the same number of simulation steps for this set of experiments (see, Figures 8.6(a) – 8.5(a) and 8.6(b) – 8.5(b)).

Comparing Figures 8.6(c) and 8.6(d) with Figures 8.5(e) and 8.5(f), respectively, we observe that in case of $R(I,S) = 0.25$ the flow of the propagation follows a decrease, with less or none grows (see, Figure 8.5(e) and Figure 8.5(f)) that is attributed to the number of low initially infected population. However, observing Figures 8.6(c)

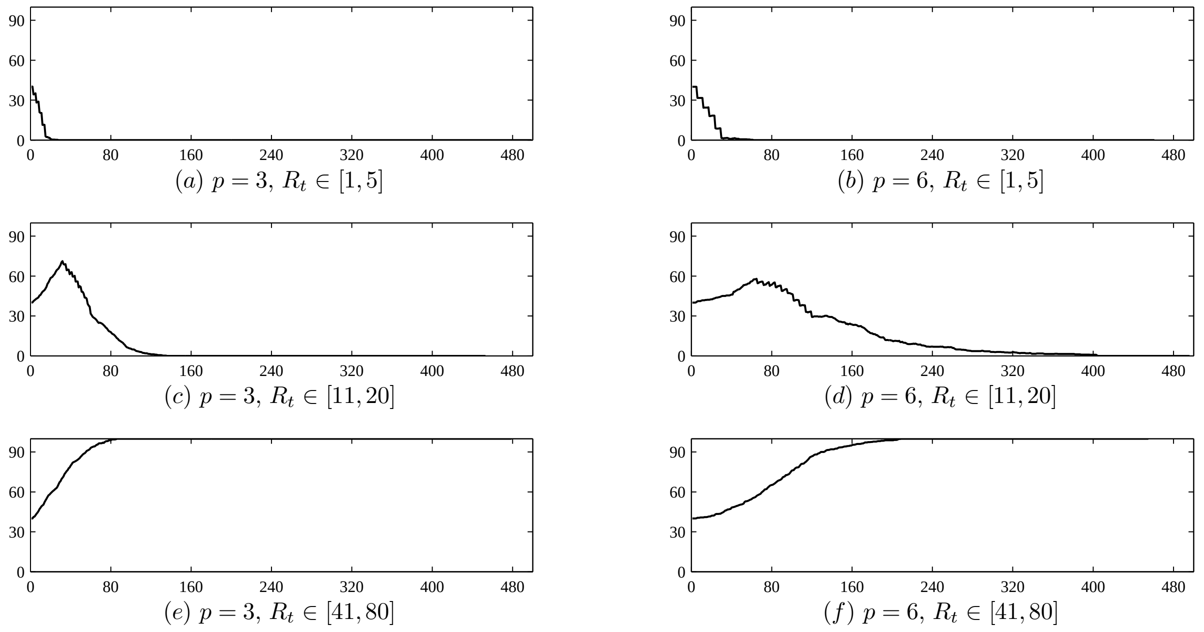


Figure 8.6: Simulation experiments implementing the properties of SIRpI epidemic model, for the cases of a malware with $p = 3$ and $p = 6$, on a network with $R(I,S) = 0.66$ and double initially infected devices.

and 8.6(d) we can see that in both cases a grow-level is expressed through relatively early steps, which is a fact that respectively is attributed to the higher initially infected population, i.e. $R(I,S) = 0.66$. Moreover comparing Figures 8.6(c) and 8.5(e) we observe that due to the application of immunization procedure (applying SIRpI epidemic model) the whole pandemic prevention lasts less as more susceptible devices, including the initially infected population, have been early enough sanitized and hence immunized, leaving no space for farther infection.

Contrasting Figures 8.6(e) – 8.6(f) with Figures 8.5(i) – 8.5(j), respectively, we observe interesting evidences about our intuition that the number of initially infected devices could significantly speed up the propagation of malware in such a network. So, in the experiments depicted in Figure 8.6(e) and 8.6(f) we observe that in both cases the counter-measure's activation, due to its larger response time against a larger initially infected population, failed to prevent pandemic, in contrast to the case of experiments presented in Figure 8.5(i) and 8.5(j).

In the second set of experiments for malware propagation in a network of mobile

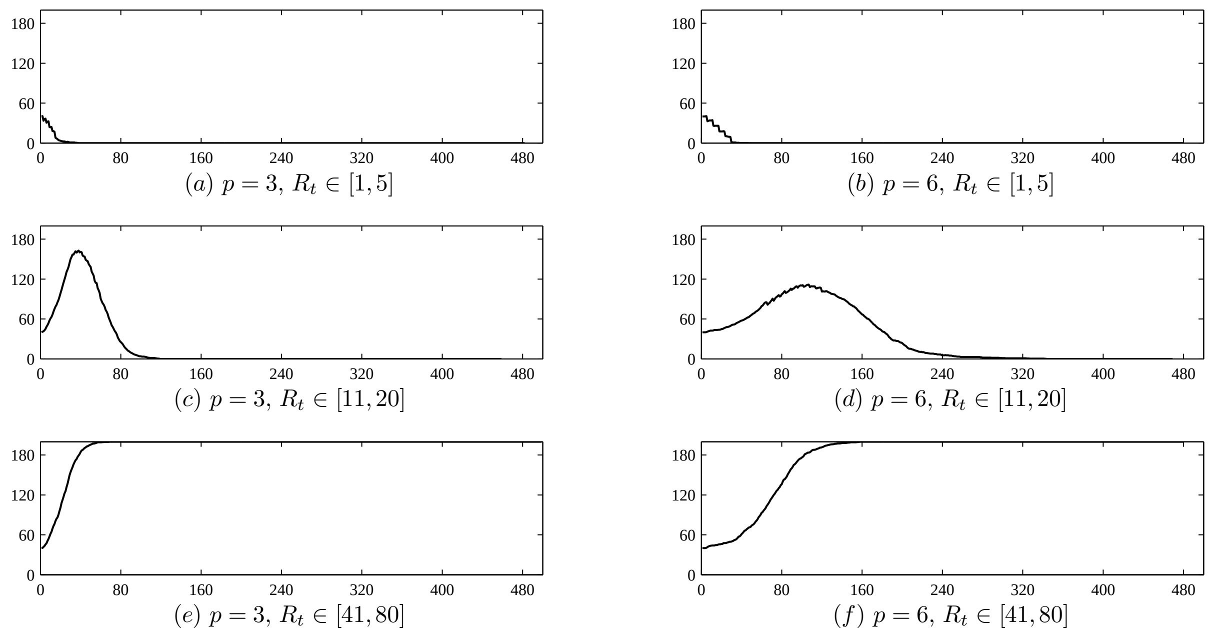


Figure 8.7: Simulation experiments implementing the properties of SIRpS epidemic model, for the cases of a malware with $p = 3$ and $p = 6$, on a network with $R(I, S) = 0.25$ and double density.

devices with double density of the network on which the experiments of Figure 8.5 performed we observe that for an early activated counter-measure the pandemic is quickly prevented (see, Figures 8.5(a) – 8.7(a) and Figures 8.5(b) – 8.7(b)).

On the other hand observing Figures 8.5(e) – 8.7(c) and Figures 8.5(f) – 8.7(d), similarly to the experiments presented on Figures 8.6(c) – 8.6(d), we can see that in both cases the grow-level of the epidemic is expressed through relatively early steps, which is attributed to the higher initially infected population and once again that the application of immunization procedure causes the whole pandemic to be prevented in less time by a counter-measure activated on the same response-time interval, as more infected devices get immunized. Hence, comparing Figure 8.5(e) and 8.5(f) with Figure 8.7(c) and 8.7(d), we observe that the network's density (as also the initial infected population, see Figures 8.6(c) and 8.6(d)) do not affect the prevention of a pandemic when a properly activated counter-measure exists.

Finally, comparing Figures 8.7(e) and 8.7(f) with Figures 8.5(i) and 8.5(j), respectively, also interesting evidences arise about our intuition that an increase on the density of the network could significantly speed up the propagation of malware. More precisely, in Figures 8.7(e) and 8.7(f) we observe that in both cases the counter-measure's activation, due to its larger response time, failed to prevent pandemic, in contrast to the case of experiments presented in Figures 8.5(i) and 8.5(j).

Preventing Pandemics that follow the SIRpS Epidemic Model

Next, we present the two categories of experiments concerning the SIRpS epidemic model, investigating the effect of the counter-measure's response-time on the spread of the malware concerning the activation of the counter-measure on each device by setting the response-time to various intervals, as also the effect of other factors, such as the initial size of the infected population and the density of the network. Finally, we present some comparative results taken by our simulator implementing our device mobility and malware propagation models, related to the properties of SIRpI and SIRpS epidemic models.

Pandemic Prevention for Various R_t-intervals

As we can observe in Figures 8.8(a) and 8.8(b), the size of the spreading malware does not affect the spread at all since the response time is low. That means, the propagation of malware to neighboring susceptible devices fails, despite its size, due to the early activation of the counter-measure. However, as in the corresponding experiments of the SIRpI epidemic models (see, Figures 8.5(a) and 8.8(b)), duplicating the size of the malware the time required for all the susceptible devices in the city to avoid the infection and all infected ones get sanitized is also duplicated. Moreover, increasing the counter-measure response time inside the same order of magnitude, we observe that still there is no significant increase on the number of the infected population, where in both cases (see, Figures 8.8(c) and 8.8(d)) the number of infected population, from the start of simulation, is decreasing since the cure has started once the counter measure has been activated, while the effect made by the duplication of spreading malware's size is the duplication on the time needed by the counter-measure to prevent the pandemic successfully.

Similarly to the experiments on the SIRpI epidemic model (see, Figures 8.5(e) through 8.5(h)), leaving the rest parameters unchanged and increasing only the response-time of the counter-measure, we observe that even if there do not yield any maxima of the spread that could lead to pandemic, still all of them (see, Fig-

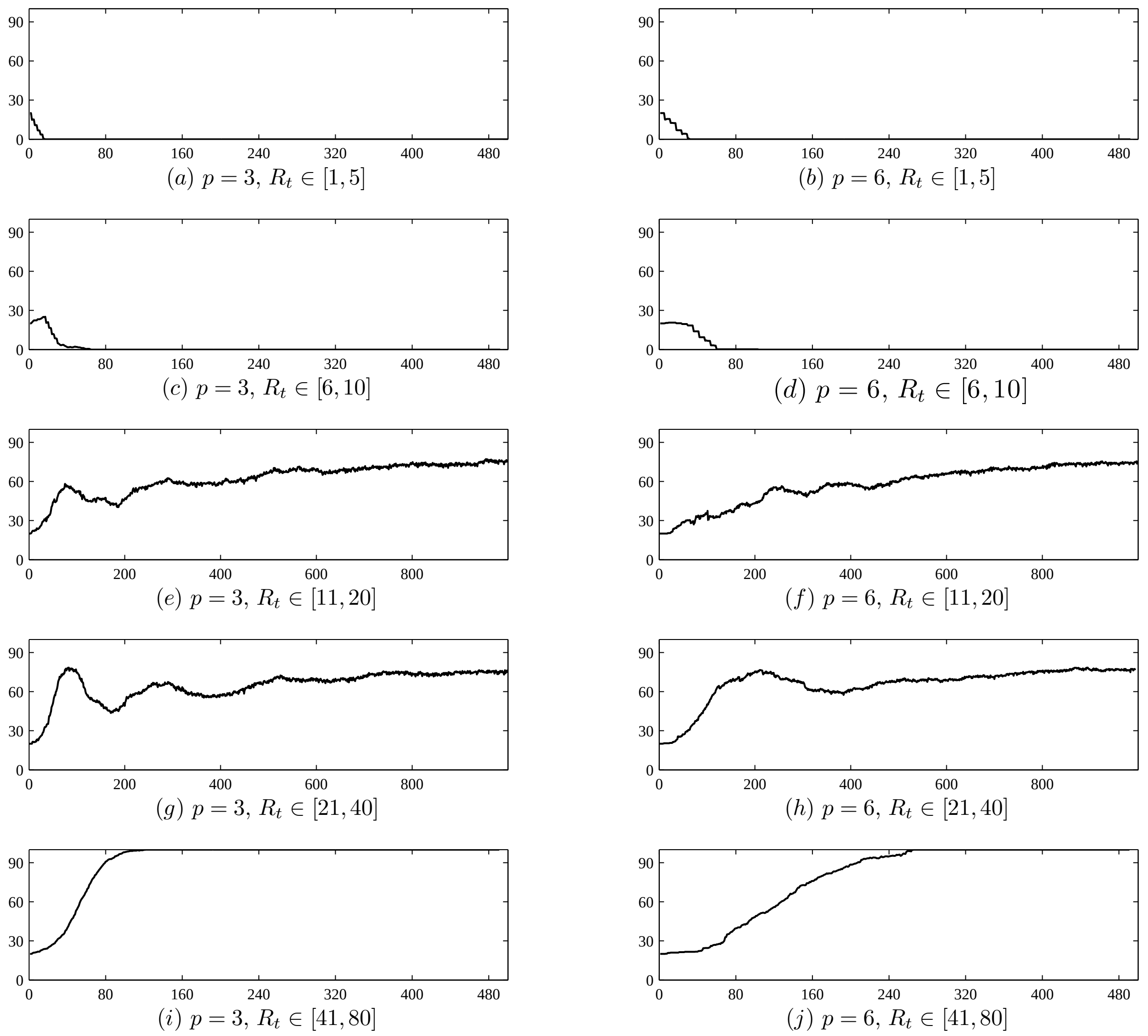


Figure 8.8: Simulation experiments implementing the properties of SIRpS epidemic model, for different values of malware packets (p), on a network with $R(I,S) = 0.25$ for various counter-measure response-time intervals (R_t).

ures 8.8(e) through 8.8(h)) follow a similar propagation behavior. As we can observe, contrary to the corresponding experiments of SIRpI epidemic model, in the experiments presented on figures Figures 8.8(e) through 8.8(h), the properties of the SIRpS epidemic model lead the whole malware's propagation to a stabilized state concerning the *IS-rate* through time. This means that, despite the adequately early counter-measure activation time (concerning the properties of SIRpI epidemic model), a counter-measure response-time bound established to work properly over an epidemic that follows the properties of SIRpI model would not prevent the pandemic when applied on epidemics that follow the properties of SIRpS epidemic model, as the portion of infected population seems to converge to a stabilized value if we extent the observation time.

Finally, observing the behavior of the propagation, a pandemic prevention failure results in a further increase on the counter-measure's response time. We can clearly see that, in contrast to the experiments presented in Figures 8.8(e) through 8.8(h), no fluctuations on the spread are exhibited as a non-adequately early activated counter-measure is applied. The behavior of the propagation is straightly differentiated from the previous ones where the spread follows the well-known in epidemics sigmoid curve (see, Figures 8.8(i) and 8.8(j) respectively), where, as previously, the duplication of spreading malware's size leads to a duplication on the time needed by the spreading malware to infect all the susceptible devices.

Other Results on SIRpS

Next we present results for a series of experiments for malware consisted by 3 and 6 packets for various response time intervals for pandemic prevention of an epidemic that follows the properties of the SIRpS epidemic model. The design of the experiments discussed in this section follows the design of those presented in Section 3.2. We change some factors concerning the characteristics of the network (i.e., G_{dev}) and firstly perform a series of experiments for a different *IS-rate* $R(I, S) = 0.66$ having an initially infected population consisted by 40 devices and initial susceptible population consisted by 60 devices, i.e., $I = 40$ and $S = 60$, while then we increase the density D of our network by duplicating the number of devices keeping the *IS-rate* equal to that of Figure 8.5; recall that, $R(I, S) = 0.25$, where $I = 40$ and $S = 160$.

Following the experimental design presented in Section 3.2, the results of the second category of our experiments are depicted in Figures 8.9 and 8.10. In this category, we modify the experiments of Figures 8.8(a) – 8.8(b), 8.8(e) – 8.5(f) and 8.8(i) – 8.5(j), changing the ratio I/S from 0.25 to 0.66 (see, Figure 8.6) and the density D of the network by duplicating the number of devices (see, Figure 8.10); recall that, I and S denote the numbers of the initial infected and susceptible population, respectively.

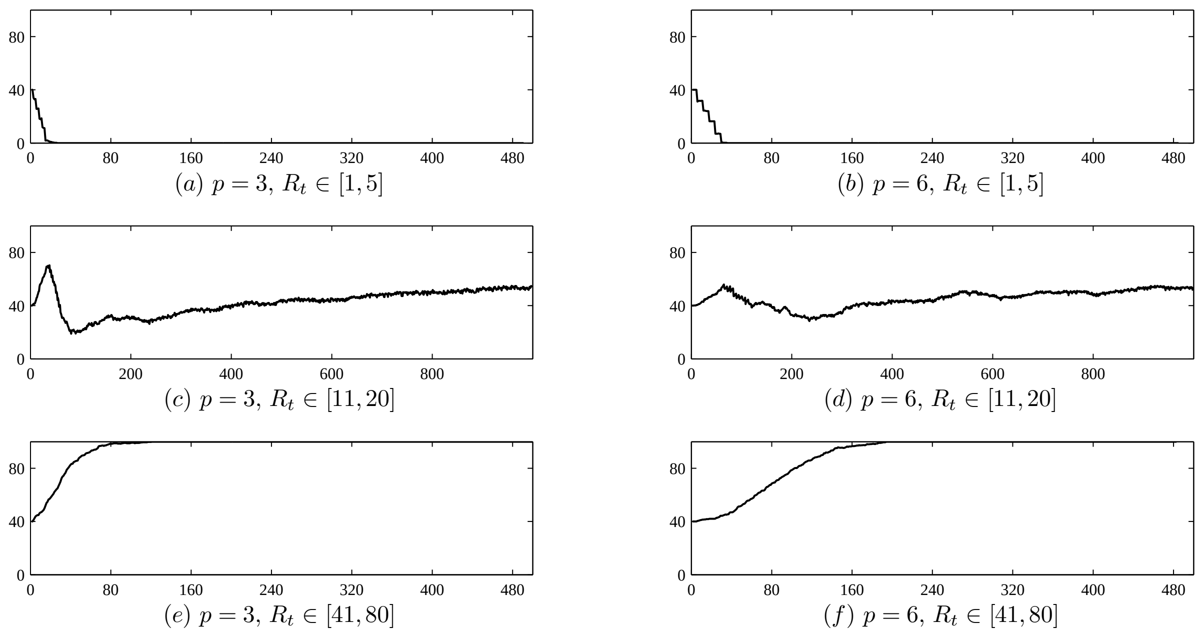


Figure 8.9: Simulation experiments implementing the properties of SIRpS epidemic model, for the cases of a malware with $p = 3$ and $p = 6$ on a network with $R(I, S) = 0.66$ and double initially infected devices.

It is rational to expect that in presence of an early activated counter-measure (i.e., response-time intervals close to 0), even in case of an epidemic that follows the properties of the SIRpI epidemic model, the response-time of an adequately early activated counter-measure is crucial to achieve a successful pandemic prevention. So, response-time intervals close to 0, act the same for pandemic prevention despite the size of the initial infected population, as shown in the results contrasting Figures 8.9(a) and 8.9(b) with Figures 8.8(a) and 8.8(b), where the prevention of a pandemic needs almost the same number of simulation steps for these couples of experiments (see, Figures 8.9(a) – 8.8(a) and 8.9(b) – 8.8(b)).

Comparing Figures 8.9(c) and 8.9(d) with Figures 8.8(e) and 8.8(f), respectively, we observe that they express almost the same behavior. However, for the case where the initial *IS-rate* is duplicated, we observe (see Figures 8.9(c) and 8.9(d)) that the grow level is more higher and also it has been expressed over earlier steps when compared to the one of Figures 8.8(e) and 8.8(f), an observation that is definitely attributed to the increased initial infected population. Additionally, similarly to Fig-

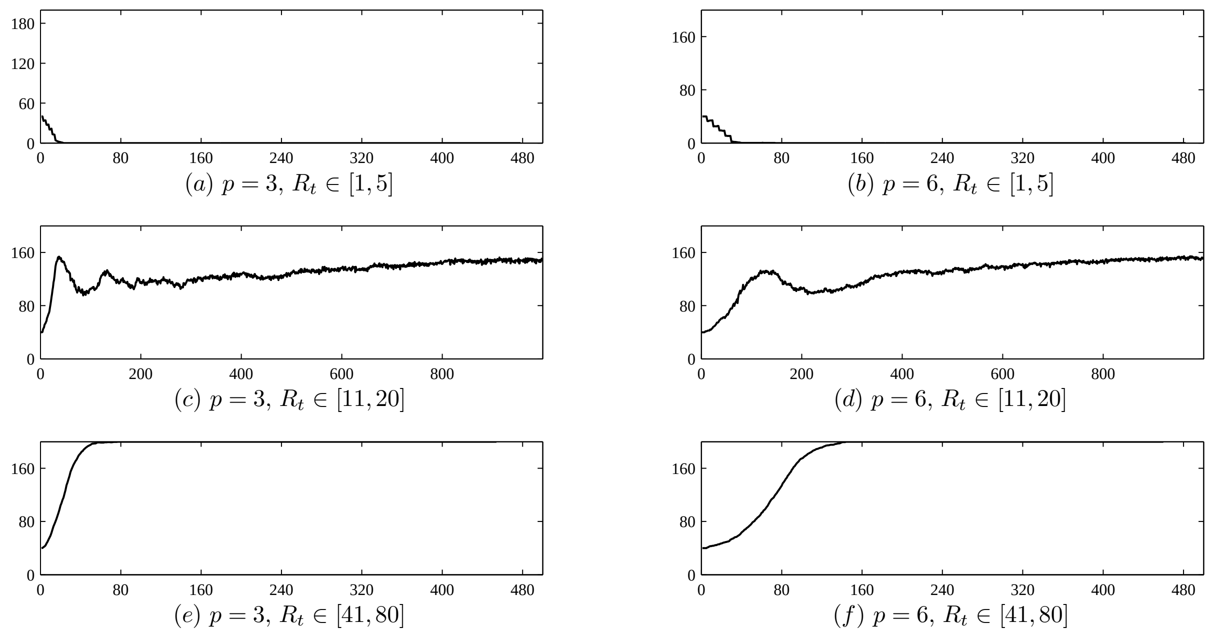


Figure 8.10: Simulation experiments implementing the properties of SIRpS epidemic model, for the cases of a malware with $p = 3$ and $p = 6$, on a network with $R(I, S) = 0.25$ and double density.

ures 8.5(e) through 8.5(h)) in these classes of response-time intervals (i.e., $[11, 20]$ and $[21, 40]$) the properties of SIRpS epidemic model lead the whole malware's propagation to a stabilized state concerning the *IS-rate*, as the portion of infected population seems to converge to a stabilized value if we extent the observation time.

However, comparing Figures 8.9(e) and 8.9(f) with Figures 8.8(i) and 8.8(j), respectively, we observe a rational speed-up on the propagation rate w.r.t the simulation steps demanded by the spreading malware to infect all the Susceptible devices in the area, where the duplication of the initial Infected population leads to a faster spread, with a pandemic to be expressed, demanding half time needed by the one with the half *IS-rate*.

In the second set of experiments for malware propagation in a network of mobile devices with double density of the network on which the experiments of Figure 8.8 performed, similarly to the case of Figures 8.5(a)–8.7(a) and 8.5(b)–8.7(b) we observe that the behavior of the propagation on presence of an early activated counter-measure is almost the same.

However, similarly to previous cases, comparing Figures 8.8(e) and 8.8(f) with Figures 8.10(c) and 8.10(d), we observe that in the second case there is a greater grow level on the number of Infected devices in the early steps. Comparing the corresponding results of previous cases we conclude that, regardless the underlying epidemic model, the success or not of a pandemic (or, respectively, pandemic prevention) in presence of a counter-measure of moderate response-time is strongly connected to the density of the network. Moreover, throughout the experiments we can observe that such response-time intervals even if they are not directly able to prevent pandemic by cleaning all susceptible devices, end hence eliminating the spreading malware from the network, they tend to exhibit a behavior of stabilizing the *IS-rate* over a specific value. Additionally, comparing Figures 8.10(c)–8.9(c) and 8.10(d)–8.9(d), where both have the same initial Infected population, we conclude that in the first case (see, Figures 8.10(c) and 8.10(d) respectively) a denser network could act beneficially on the failure of a pandemic prevention.

Finally, comparing Figures 8.10(e) and 8.10(f) with Figures 8.8(i) and 8.8(j), respectively, we observe once again that an increase on the density of the network increases malware's spread. In both cases the counter-measure's activation, due to its larger response time, failed to prevent pandemic, similarly to the experiments presented in Figures 8.8(i) and 8.8(j). Moreover, in presence of a malware of smaller size, its propagation is even faster on a denser network where comparing Figures 8.10(e) and 8.8(i) the propagation of malware in a network of high density demands almost the half simulation steps, leading us to infer that the higher the density of a network, the earlier the response-time of a counter-measure should be.

Chapter 9

Duscussion

9.1 Evaluating the Proposed Model for Malware Detection and Classification

Next, in Table 9.1 we cite the detection rates exhibited by other models utilizing the dependencies between System-calls and proceed by a discussion on the proposed techniques the true positives and false positives achieved by each model. Additionally, in Table 9.2, we present the classification rates exhibited by other models utilizing the dependencies between System-calls and proceed by a discussion on the proposed techniques and the classification accuracy results exhibited by each model.

In:	Technique	True Positives	False Positives
[4]	Sequence Matching (ScDG)	64.00%	00.00%
[61]	Graph-Grading(ScDG)	80.09%	11.00%
[this thesis]	Δ-similarity (GrG graphs)	94.70%	13.10%

Table 9.1: Detection results; note that [this paper] uses the same dataset as[6] and [5].

In:	Technique	Classification Ratio
[38]	Behavior Profiles (ScDG)	95.90%
[68]	K-nearest neighbors (FCG)	69,90%
[39]	Discriminative Behaviors	88,00%
[42]	Function Length	83.95%
[this thesis]	SaMe- and NP-similarity (GrG graphs)	83.42%

Table 9.2: Classification results.

9.2 Discussion on Detection Results

In this section there are discussed the detection results exhibited by other models. There are presented the results concerning the detection rates exhibited by models that make use of the dependencies between system-calls independently if they are graph-based or not.

Non Graph-based Approaches. Kolbitch *et al.* [4] proposed an effective and efficient approach for malware detection, based on behavioral graph matching by detecting string matches in system-call sequences, that is able to substitute the traditional anti-virus system at the end hosts. The main drawback of this approach is the fact that although no false-positives where exhibited, their detection rates are too low compared with other approaches. Luh and Tavolato [61] presented one more detection algorithm based on behavioral graphs that distinguishes malicious from benign programs by grading the sample based on reports generated from monitoring tools. While the produced false-positives are very close to ours, the corresponding detection ratio is even lower.

Non Graph-based Approaches. In malware detection, there have been proposed similar models utilizing different non graph-based techniques like the one proposed by Alazab *et al.* [13], who developed a fully automated system that disassembles and extracts API-call features from executables and then, 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. In [67], Ye *et al.* described an integrated system for malware detection based on API-sequences. This is also a different model from ours since the detection process is based on matching the API-sequences on OOA rules (i.e., Objective-Oriented Association) in order to decide the maliciousness or not of a test program. Finally, an important work of Christodorescu *et al.*, presented in [14], proposes a malware detection algorithm, called $\mathcal{A}_{MD}$, based on instruction semantics. More precisely, templates of control flow graphs are built in order to demand their satisfiability when a program is malicious. Although their detection model exhibits better results than the ones produced by our model, since it exhibits 0 false-positives, it is a model based on static analysis and hence it would not be fair to compare two methods that operate on different objects.

9.3 Discussion on Related Classification Results

In this section there are discussed classification results exhibited by other models. There are presented the results concerning the classification accuracy exhibited by models that make use of the dependencies between system-calls independently if they are graph-based or not.

Graph-based Approaches. In [38] Bayer *et al.*, propose a scalable clustering approach to identify and group malware samples that exhibit similar behavior, serving as input to an efficient clustering algorithm *profiles* that characterize programs activity in more abstract terms. 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. However, the model proposed in [38] mainly aims on clustering malware samples rather that classifying unknown ones to known malware families, that is a slightly different process.

Hu *et al.* in [68], design implement and evaluate the Symantec's Malware Indexing Tree (SMIT), that classifies malwares based on their function call graphs using k nearest-neighbor search. While, as referred in [68], their success rate reaches the

91.3%, it is worth mentioning that this classification rate refers in the case where the actual labeling of test samples family in included in the k nearest families resulted by the model. Hence, since our model returns only one dominant family we compare our results (i.e., 83.47%) with the results referred in [68] as Dominant Family Rate (i.e., 69.9%), that is defined as the percentage where the most prevalent family among k returned nearest neighbors is also the family to which the query malware belongs.

A model for malware classification utilizing discriminative behavior specifications extracted by the samples is presented by Rieck *et al.* in [39]. Specifically, by monitoring malware samples in sandbox, they collect behaviors, and based on a corpus of malware labeled by an anti-virus scanner a malware behavior classifier is trained using learning techniques. Finally, discriminative features of the behavior models are ranked for explanation of classification decisions. To this point we ought to mention that, despite the fact that their classification results for known malware samples are almost 5% higher that ours, we recall that, as in [69] their experiments are performed using 14 malware families, where the impact of philogeny among different malware families is decreased the less different malware families in the training are.

In [42] Tian *et al.* present a scalable method of classifying Trojans based only on the lengths of their functions. The results achieved by the proposed technique indicate that function length may play a significant role in classifying malware, and combined with other features, may result in a fast, inexpensive and scalable method of malware classification. However, while their results are comparable to our model's, the main difference is the fact that in [42] the model has been evaluated using only Trojans.

Non Graph-based Approaches. In malware classification, there have been proposed other non graph-based malware classification models. Among them, a scalable automated approach for malware classification using pattern recognition algorithms and statistical methods, is presented by Islam *et al.* in [69], utilizing the combination of static features extracted by function length and printable strings. While their evaluation results are very high(i.e., 98.8% classification accuracy), however it is worth mentioning the fact that their experiments include samples from 13 malware families, while the classification accuracy of the model proposed in this paper has been evaluated over 48 malware families. Hence, concerning the impact of philogeny among different malware families the comparative difference between the classification rates achieved by these two models is totally justified, while increasing the number of families in the training set increases the chances of misclassifications. Recently, Nataraj *et al.* [70] classify malware samples using image processing techniques. Visualizing as gray-scale images the malware binaries, they utilize the fact that,for many malware families, the images belonging to the same family appear very similar in layout and texture. Obviously the results are better than the ones produce by our model however they use at most 25 malware families for their large scale experiments, where the impact of philogeny among different malware families is decreased the less different malware families in the training are. In [71] Nataraj *et al.* utilize a static analysis technique called binary texture analysis in order to classify malicious binary samples into malware families. They achieve a 72% rate of consistent classification when performing their evaluation on a data set of $60K$ to $685K$ samples comparing their labels with those provided by AV vendors, proving both the accuracy and the scalability of their model.

Remark. It is worth noting that, although the classification results presented in [69, 70, 39] are better than those achieved by the model proposed in this paper, their experiments are performed using less malware families, i.e., 13 malware families for [69], 25 malware families for [70], and 13 malware families for the [39]; recall that, our model has been evaluated over 48 malware families. Hence, concerning the impact of philogeny among different malware families the comparative difference between the classification rates achieved by these two models is totally justified, while increasing the number of families in the training-set increases the chances of misclassification.

9.4 Malware Pandemic Prevention Defining Optimal Response-time Bounds

Next we discuss some comparative results on the behavior of the malware's spread concerning the properties of the corresponding epidemic models studied in this paper. In Table 9.3 we present an accumulative view on our results for pandemic prevention (or not) while performing a series of repetitive experiments implementing the properties of either SIRpI or SIRpS epidemic models for various response-time intervals and for different spreading malware's sizes.

IS-rate $R(I,S)$, Network Density	Response-time Intervals R_t	SIRpI		SIRpS	
		$p=3$	$p=6$	$p=3$	$p=6$
$R(I,S)=0.25$ Density D	$R_t \in [1-5]$	yes	yes	yes	yes
	$R_t \in [6-10]$	yes	yes	yes	yes
	$R_t \in [11-20]$	yes	yes	no	no
	$R_t \in [21-40]$	yes	yes	no	no
	$R_t \in [41-80]$	yes	yes	no	no
$R(I,S)=0.66$ Density D	$R_t \in [1-5]$	yes	yes	yes	yes
	$R_t \in [11-20]$	yes	yes	no	no
	$R_t \in [41-80]$	no	no	no	no
$R(I,S)=0.25$ Density $2D$	$R_t \in [1-5]$	yes	yes	yes	yes
	$R_t \in [11-20]$	yes	yes	no	no
	$R_t \in [41-80]$	no	no	no	no

Table 9.3: Accumulated view on the exhibited results for pandemic prevention

Observing Table 9.3, we can see that as long as the response-time $R_t \leq 10$, both epidemic models fail to infect all the susceptible devices in the area. In rows 1 and 2 of the same table, where $R_t \in [1-5]$ and $R_t \in [6-10]$, respectively, despite the size of the spreading malware, a counter-measure's activation of low response-time leads to clean all the infected devices, since in all the cases the number of infected population has been eliminated demanding simulation steps of the same order of magnitude. On the other hand, for response-time $R_t > 10$, we observe that the two epidemic models SIRpI or SIRpS are differentiated concerning the success or not of the pandemic prevention. As we can see in rows 3, 4 and 5, where $R_t \in [11-20]$, $R_t \in [21-40]$ and $R_t \in [41-80]$, respectively, the pandemic on a malware's spread that follows the properties of SIRpI model, could be successfully prevented in all cases, otherwise if the spread follows the properties of SIRpS epidemic model in both cases the pandemic could not be prevented.

Next, rows 6, 7, 8 and 9, 10, 11 of Table 9.3 accumulate the results for pandemic prevention (or not) while performing a series of experiments implementing the properties of either SIRpI or SIRpS epidemic model for various response-time intervals and different malware's sizes exploring the impact of *IS-rate* and network density D, respectively.

Increasing the *IS-rate* over the second category of our experiments, in rows 6, 7 and 8 of Table 9.3, we observe that the increase on the *IS-rate* has a different effect regarding each epidemic model. More precisely, for the minimum and maximum response time-intervals (i.e., $R_t \in [1-5]$ and $R_t \in [41-80]$), the spread follows the same behavior despite the malware's size and the underlying epidemic model. However, for the median case of a response-time interval (i.e., $R_t \in [11-20]$) the behavior of the spread is differentiated analogously to the underlying epidemic model. In the case where the spread follows the properties of SIRpI epidemic model the pandemic is prevented, contrary to a spread that follows the properties of SIRpS epidemic model. Comparing now rows 1, 3, 5 and 6, 7, 8, respectively, we can see that the first two pairs are matched while the third pair is differentiated, inferring that an increase on the *IS-rate* regarding the epidemic model finally affects only the cases where the a lately activated counter-measure is applied.

Similarly to the increase on the *IS-rate*, in rows 9, 10 and 11 of Table 9.3, we observe that compared to 1, 3, 5, respectively, an increase on the network's density D acts differently across both epidemic models. More precisely, for the minimum and maximum response time-intervals (i.e., $R_t \in [1-5]$ and $R_t \in [41-80]$) the spread follows the same behavior, despite the malware's size and the underlying epidemic model, while for the median case of a response-time interval (i.e., $R_t \in [11-20]$) the behavior of the spread is strictly correlated to the properties of the underlying epidemic model.

9.5 Trusted Systems

As long as the use of distributed systems and Cloud Computing is significantly increasing, the amount of threats concerning the security of such systems and the data stored in them set great challenges on the application of Trusted Computing. In this work, we aim to discuss the Trusted Computing approaches applied on Cloud Computing security and focusing on their drawbacks on hardware verification (i.e., to attest hardware's integrity). We propose a model for hardware integrity attestation applied on Cloud Computing systems, presenting the main protocol based on the use of Endorsement Keys (EK) known from Trusted Computing. The validity of our protocol and its potentials against hardware based attacks is proved by the combined use of verified Public-Key encryption algorithms.

One of the major challenges on Cloud Computing is the assurance of properties as confidentiality, integrity, availability, and privacy. Indirectly, such texture indicates the issues arising from the fact that the data access many entities (multi-tenancy) and the supply of sensitive data to third entities that are trusted, or not. Thus, as the Cloud Computing enables handling and inter-operability between major resources which are owned and managed by different entities, security is the fundamental demand on the infrastructure of Cloud Computing. Moreover, prerequisite of such demand is the subsistence of trust between the entities that exchange data composing the Cloud Computing and so the corresponding Cloud Computing Environment.

As long as Cloud Computing by itself poses important issues on the axes of confidentiality, integrity, availability and privacy, the arising security issues can be divided into those related to the provider (Infrastructure as a Service - IaaS, Software as a Service - SaaS and Platform as a Service - PaaS) and those related to the client-user. Different service models yield different demands on the security of data and applications. Regarding the case of PaaS model, Cloud Service Providers (CSP) are responsible to ensure the security of the computing platform as well as the development environment, while, users (clients) are required to ensure applications. On the other hand, in the IaaS model, CSPs are required to provide users with a trusted host as well as a trusted Virtual Machine Monitoring (VMM) environment, and users must ensure the trustworthiness of their VMs by themselves. Finally, for the case of SaaS model, the user is responsible for the applications provided by the CSP, the privacy of the data, and the wider adoption of data security rules[6].

9.5.1 Trust Threats

Several security issues arise in Cloud Computing security [90, 91, 92, 93]. Cloud providers in order to ensure the confidentiality and integrity of data provided by their users, they have deployed a series of mechanisms (i.e., monitoring and auditing of data access) to prevent the disposal of confidential data to unauthorized third entities.

Due to the separation of tasks, a service provider, say SP_1, may use another provider (e.g. infrastructure provider) say SP_2, for the required infrastructure. Thus, since SP_1, being unable to access the data center of SP_2, leaves the security of his customer's data on SP_2, and hence SP_2 oughts to guarantee data privacy on transfer and access level as also the auditing regardless of whether the application that manages the data has been compromised or not.

Briefly, the confidentiality of data is achieved through cryptographic protocols, while the auditing is achieved through the use of remote attestation techniques requiring the use of Trusted Platform Module (TPM) [94, ?, 95, 96], while the remote attestation techniques may be added as another level on top of the virtualized OS. Therefore, confidentiality and auditing can be achieved by providing another level, maintaining a corresponding software. In our work we are mainly interested in investigating the necessity of such techniques and their applications on virtual environments, where VMs can dynamically change their location (migration) from one host to another. Hence, the major security issue is indicated on Trust consolidation mechanisms among different levels of cloud.

9.5.2 Establishing Trust

First, concerning the cases of hardware based attacks, we provide a cryptographic protocol that leveraging measurement lists and based on the use of Endorsement Keys (EK) in a combination with time-stamps and verified Public-Key encryption algorithms provides a remote attestation for hardware's integrity. So, in our work we propose an applicable cryptographic protocol that is able to ensure hardware-integrity attestation for devices that need to be remotely attested before being connected to cloud.

Fundamental approaches [97, 98, 96, 99] proposed in recent years and utilize Trusted Computing in order to ensure trustworthiness finding application in the form of IaaS Cloud Computing where the main objective is to secure the virtual machines. Below we discuss some of them that intrigued our work.

9.5.3 Routing Trust In Cloud Computing With TVEM

Trusted Virtual Environment Module (TVEM) [100] is a software module that provides trusted services to a virtual machine or a virtual environment of an IaaS Cloud Computing environment. More precisely, TVEM provides a set of features for the virtual environments of Cloud, via virtualization techniques of the existing TPM, containing an improved API, flexible cryptographic algorithms and a modular architecture. TVEM solves one of the major security issues of Cloud Computing, the establishment of trust relations between the information owner and a service provider when the former creates and uses a virtual environment on the provided platform. It easily follows that the aforementioned problem arises mainly in the IaaS form of Cloud Computing. To ensure that the information provided in the Cloud is protected, the client should verify the "trustworthiness" of both the platform and the virtual environment. TVEM as well as Virtual Trust Network (VTN) provide the necessary mechanisms to verify the trustworthiness of platform and the virtual environment in IaaS Cloud, providing in addition the corresponding results to information owner.

The Trusted Platform Module (TPM), which is the root of trust on a platform (i.e., component of the computing platform which explicitly considered trusted), is designed to support a single operating system on a standard platform and therefore yields scalability issues in virtualization, due to the existence of several virtual environments, which simultaneously try to access the TPM's resources. TVEM addresses this demand by a method for virtualization of the TPM functions for sharing among several virtual environments, reproducing substantially its Trusted Platform Module resources on software [100].

By Virtual Environment Trust define trust in the cloud that combines the confidence in the platform provider (service provider) and confidence on the part of the information holder (information owner). So the problem that solves the application of TVEM is the establishment of trust in a virtual environment (virtual environment trust) which is unique for each virtual environment and independent from the platform that hosts it. To obtain the Virtual Environment Trust set the Trusted Environment Key (TEK) which is used as the Endorsement Key (EC), which we mentioned in the previous section, for Trusted Virtual Environment Module. The ECF is created by the owner of the virtual environment and ensure the Platform Storage Key (PSK) to the service provider (service provider) so am creating "combinatorial” trust (compound trust) which is unique and independent of the platform. So, just like the EP TEK has a unique value, which is used as a Root of Trust for Reporting (RTR) to identify the TVEM and demonstrate the virtual environment.

In order to protect the TVEM, are utilized both VT-d (Intel’s Virtualization Technology for Directed I/O) and TXT (Trusted eXecution Technology). TVEM supports attestations as well as trusted data entry for the virtual environment similar to the one provided by the Virtual Trusted Platform Module (VTPM) [100]. TVEM aims, using a single TPM via multiplexing, to give the virtual machines the “illusion” that each one has exclusive access to the TOM [95]. The flexibility and the scalability of TVEM are indicated on its property that does not require its alignment with the settings of TPM. The elasticity of TVEM arises from its contained API that actually carries the Trusted Software Stack (TSS) in TVEM releasing it from its implementation. The advantages of TVEM enable system designers to modify at will so the TVEM as the overall virtual environment in order to fulfill the confidentiality and integrity requirements of information. Definitely TVEM is not stand-alone, including besides TVEM, a TVEM manager to host hypervisors for accessing the TPM of the host platform and availability of TVEM, a control level of VTNs serving the overall management system, as well as a TVEM Factory (TF) which generates the TVEM, manage keys and provides safely the TVEM to platforms that will use it.

9.5.4 Trusted Cloud Computing Platform (TCCP)

Similarly to TVEM, Trusted Cloud Computing Platform (TCCP) [101] developed to ensure the IaaS form of Cloud Computing, as its design focuses on node management and the management of VMs. Describing the situation addressed by TCCP, the problem is caused by the fact that it is difficult to guarantee the confidentiality of data calculations provided by the services which are in the upper levels of the software stack on the SaaS form of Cloud Computing, as the services themselves provide the software which directly manages customer’s data. Due to this reason, TCCP aims at the lower levels (IaaS), where the security of customer’s VMs is more manageable [101].

The application of TCCP aims to ensure the confidentiality and integrity of the calculations performed by users in IaaS services. Describing the approach abstractly, TCCP provides the structure of a ”Closed Box Execution Environment” for virtual client machines in order to guarantee that it will not be possible, even for a manager of the provider service, supervise or skew data used in calculations performed by virtual machines. A Closed Box Execution Environment, ensures that, in the environment where the VMs are hosted, even a user with full rights is not able to monitor or modify a guest-VM [101]. Another important feature of TCCP, is that it allows the clients of the service to be aware, even remotely, of whether the infrastructure has an implementation of the TCCP, before launching a VM, extending the concept of attestation throughout the service.

9.5.5 Security Issues of TVEM and TCCP

Both TVEM and TCCP are intended to secure VMs and virtual environments. One important difference is that TVEM is a software device, while TCCP is a platform. TVEM, along with its structural components, with which it interacts, (TVEM Manager, Trusted Factory etc.) are all implemented in software. On the other hand, TCCP, being a platform has been implemented as a system consisted by computers, and more precisely by trusted nodes and a node (Trusted Coordinator) that coordinates all the system. The main difference between TVEM and TCCP is their actual aims. TVEM aims on establishing trusted relationship among two entities, i.e., information owner and IaaS provider who will provide the former with a virtual environment. On the other hand, TCCP by its implementation aims on imposing the “closed box” execution environment in the back-end of IaaS services to ensure the confidentiality and integrity of VMs running on the environment of IaaS provider. Despite the TVEM’s modular design that makes it more customizable and flexible than TCCP, it is worth noting that, unlike TCCP, TVEM is more susceptible to hardware-based attacks.

9.5.6 Defending against Hardware-based Attacks: The Hardware Integrity Attestation Protocol

As we referred previously, a major situation which address the discussed approaches as many other utilizing Trusted Computing technology to enhance the Cloud Computing security, is that there can not be any guarantee about hardware’s integrity in provider’s infrastructure, being vulnerable to attacks based on undermining the operation of the equipment (hardware-based attacks). Thus, we propose a protocol

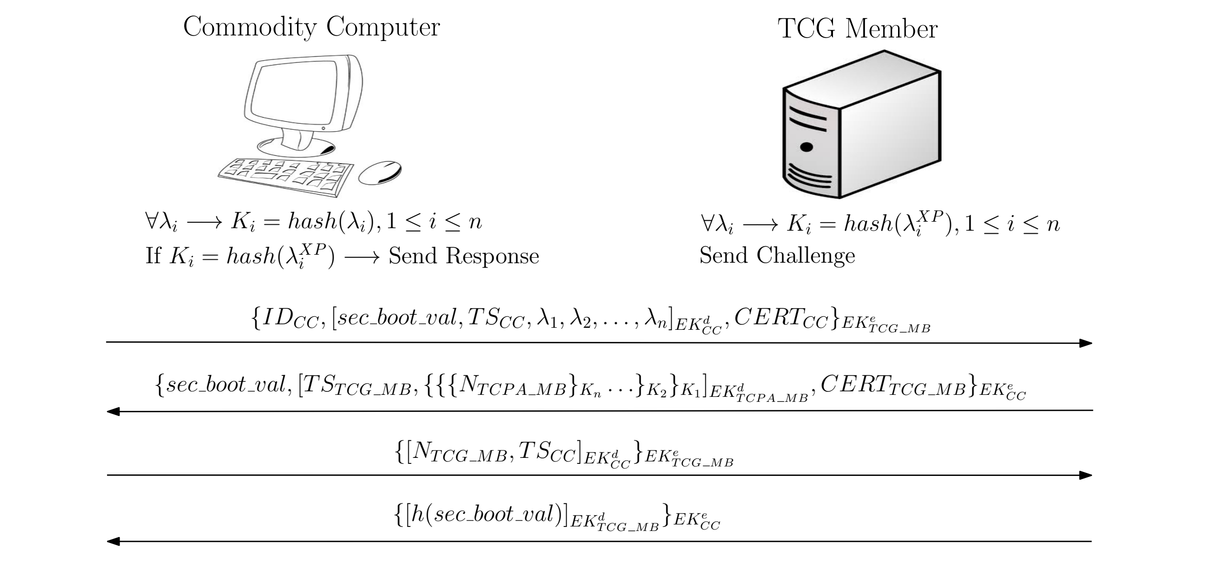

Commodity Computer
TCG Member
$\forall\lambda_i \longrightarrow K_i = hash(\lambda_i), 1 \leq i \leq n$
If $K_i = hash(\lambda_i^{XP}) \longrightarrow$ Send Response
$\forall\lambda_i \longrightarrow K_i = hash(\lambda_i^{XP}), 1 \leq i \leq n$
Send Challenge
$\{ID_{CC}, [sec\_boot\_val, TS_{CC}, \lambda_1, \lambda_2, \ldots, \lambda_n]_{EK^d_{CC}}, CERT_{CC}\}_{EK^e_{TCG\_MB}}$
$\{sec\_boot\_val, [TS_{TCG\_MB}, \{\{\{N_{TCPA\_MB}\}_{K_n} \ldots\}_{K_2}\}_{K_1}]_{EK^d_{TCPA\_MB}}, CERT_{TCG\_MB}\}_{EK^e_{CC}}$
$\{[N_{TCG\_MB}, TS_{CC}]_{EK^d_{CC}}\}_{EK^e_{TCG\_MB}}$
$\{[h(sec\_boot\_val)]_{EK^d_{TCG\_MB}}\}_{EK^e_{CC}}$

Figure 9.1: The Hardware Integrity Attestation Protocol.

which adopts the overall mentality of technology Trusted Computing regarding the measurements and the expected values of these, as well as exploiting and themselves the elements of Trusted Computing belonging to TPM. In this chapter we present two versions of the protocol created and which aims to ensure that hardware is in a suitable (expected) state when the computer starts.

The main idea behind the protocol is to create a method by which we can collect and then register information related to changes corresponding to the format and state of hardware, proceeding then with measurements regarding the configuration of each hardware component. For example, let n hardware-components {1: CPU, 2: Network Card, $\cdots$, n: motherboard } we compute the corresponding measurement lists (Measurement List -M_L) by a measurement for each component separately, in which we register information related to its configuration. Throughout the paper we shall denote with λ_i the measurement for the $i-$th component of clients system, with λ_i^{XP} will denote the expected measurement for the $i-$th component, while we shall denote with K_i we shall denote the hash value of a measurement λ.

As we can observe in Figure 9.1 the second entity involved is a member of the Trusted Computing Platform Alliance (TCG). At this point we demand that each member (company) of the TCG has the expected (XP) value list $\{\lambda_1^{XP}, \lambda_2^{XP}, \cdots \lambda_n^{XP}\}$ (those provided by hardware manufacturer) for all kinds of material and their corresponding versions. The aim of the protocol is to ensure that before the Secure Boot procedure of a computer (in our case we shall denote this computer as Commodity Computer, or, for short CC), this should provide an assurance that the hardware used has not changed its functionality, while retaining its original settings. To achieve this, we create a unique value used (sec_boot_val) and, after being certified, we try to take signed by the member of the TCG, the hash value of it as input to launch the Secure Boot process.

More precisely, since the machine itself knows this value it requires to obtain its hash value signed by a TCG member. Hence, the only action left to be done by the TCG member is to compare the current values of each list by the default (expected ones), which are registered, and if matching the TCG Member sign and return the the hash of the parameter value.

Concerning the cryptographic notation, in our protocol we shall denote with the EK the Endorsement Key (public / private) which uniquely defines each TPM and consequently each computer that has a particular TPM, while p and P denote the private and public keys used for sign and encryption respectively, with $CERT$ denoting its corresponding certificate. Moreover, and ID is used as an identification for each CC, while TS represents the time-stamp used to avoid replay attacks. At last, after a computer has been verified that its hardware has not been undermined, the next step is to proceed to the Secure Boot process. Below we enumerate and discuss step-by-step our protocol for remote hardware-integrity attestations.

1. The TPM of the computer that needs to be verified by certifying that its hardware components have the anticipated (expected) configuration sends its ID (i.e., ID_{CC}),in order for the TCG Member to be able to search its public Endorsement Key (i.e., EK^{e}_{CC}), signed with its private Endorsement Key (i.e., EK^{d}_{CC}) the unique used value (sec_boot_val) of which it expects the signed hash as mentioned above, a time-stamp TS_{cc} to ensure the freshness of the message and avoid the potential replay attacks, and measurement lists (i.e., $\lambda_1, \lambda_2, \cdots, \lambda_n$) for each hardware component, all given in a particular order (i.e., CPU, Network Card, Motherboard, ..., etc), and its ID along with the signed message and its corresponding certificate ($CERT_{CC}$) are sent to the TCG Member encrypted with its public Endorsement Key (i.e., $EK^{e}_{TCG\_MB}$).

2. On the other side, the TPM of TCG Member decrypts the message, verifies the signature (TPM Signature) and then locate the corresponding expected mea-

surement list (i.e., $\lambda_1^{XP}, \lambda_2^{XP}, \cdots, \lambda_n^{XP}$) for each hardware component of the computer. Having a set of expected measurements for each hardware component generates as many symmetric keys as the cardinality of the expected measurement list's set. The lists are creating each key by calculating the hash of any expected list as: $K_i = hash(\lambda_i^{XP}) \, \forall i, 1 \leq i \leq n$. Then the TPM of TCG Member creates a challenge (a nonce value, we shall denote its as N_{TCG_MB}) and sequentially encrypts it using one-by-one all K keys created according to the above procedure. Then, the TPM of TCG Member signs the sequentially encrypted message (encrypts it with $EK_{TCG\_MB}^{d}$ i.e., its private Endorsement Key) and along with the certificate if its signature (i.e., $CERT_{TCG\_MB}$) and the unique value sec_boot_val (sends it again to prevent repeat attacks) re-sends them encrypted with its public Endorsement Key (EK_{CC}^{e})) of its TPM as challenge for verification of CC's hardware integrity. Though this challenge-response procedure it is guaranteed that if and only if the hardware of CC has not been compromised (i.e., its measurement list will be the expected) then it will be able to decrypt the message and send the response to TCG member's challenge.

3. Through the response procedure on CCs site, the TPM of the computer requesting authentication of its hardware integrity will decrypt the message using its private Endorsement Key (i.e., EK_{CC}^{d}), will verify the signature of the TCG member, trying then through the procedure we mentioned above to create the particular series of K keys but this time using its own measurement lists M_L consisted by the set $\{\lambda_1, \lambda_2, \cdots, \lambda_n\}$. If and only if the measurements are the expected, CC will be able to create the appropriate K (i.e., computes $K_i = hash(\lambda_i) \, \forall i, 1 \leq i \leq n$) to decrypt consequently find the "nonce" value N_{TCG_MB} used as challenge. As expected, in case of failure the CC computer will not be able to start properly as to enter the session. On the other hand, in case of successful decryption, the TPM of CC will retrieve the challenge value (i.e., N_{TCG_MB}) and send it along with a times-tamp (TS_{CC}), to prevent replay attacks ensuring the freshness of its response,signed with the private Endorsement Key (i.e., EK_{CC}^{d}) and the whole message encrypted with the public Endorsement Key of TCG Member (i.e., $EK_{TCG\_MB}^{e}$).

4. Finally, in step 4, when the TCG Member decrypts the reply message sent from CC on step 3 of our proposed protocol, it can verify (or not) that the computer

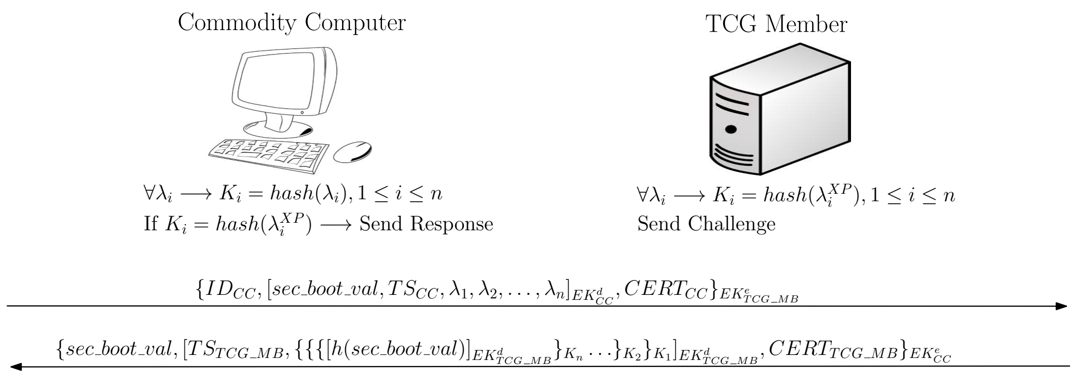

Commodity Computer
TCG Member
$\forall\lambda_i \longrightarrow K_i = hash(\lambda_i), 1 \leq i \leq n$
If $K_i = hash(\lambda_i^{XP}) \longrightarrow$ Send Response
$\forall\lambda_i \longrightarrow K_i = hash(\lambda_i^{XP}), 1 \leq i \leq n$
Send Challenge
$\{ID_{CC}, [sec\_boot\_val, TS_{CC}, \lambda_1, \lambda_2, \ldots, \lambda_n]_{EK_{CC}^d}, CERT_{CC}\}_{EK_{TCG\_MB}^e}$
$\{sec\_boot\_val, [TS_{TCG\_MB}, \{\{\{[h(sec\_boot\_val)]_{EK_{TCG\_MB}^d}\}_{K_n} \ldots\}_{K_2}\}_{K_1}]_{EK_{TCG\_MB}^d}, CERT_{TCG\_MB}\}_{EK_{CC}^e}$

Figure 9.2: The Two-Step Hardware Integrity Attestation Protocol.

was able to create the correct, or anticipated, K keys, and thus its hardware is properly formatted having its expected configuration ensuring that its is not undermined. The TPM of TCG Member signs (encrypts its private Endorsement Key $EK_{TCG\_MB}^d$) the hash of the *sec_boot_val* value required for starting the Secure Boot process and sends encrypted with the public Endorsement Key (i.e., EK_{CC}^e) of the TPM located in the certified computer CC. Once this process has been completed, $hash(sec\_boot\_val)$ signed by the TPM of TCG Member, is introduced (i.e., passes as input) to the Secure Boot process that verifies the signature, and having a signature verified by the TCG Member, the TPM computer can start properly by launching the Secure Boot process.

One improvement we can do in this protocol in order to speed up the whole process during booting the system by the time protocol is applied, is to integrate / embed the signed hash value (*sev_boot_val*) in the challenge procedure of step 2. Since the computer's TPM, managed created the appropriate keys K for sequential encryption/decryption, following that the hardware expresses the expected configuration, no further authentication is required. So,in Figure 9.2 we present another implementation of our proposed protocol. The steps (1) and (2) follow the same pattern except that only the TPM, the computer requesting certification of material integrity, managed to create the right key will decrypt sequential messages and only then find the signed the hash value (sec_boot_val). Then passes as input to the Secure Boot and the procedure followed as before. Below we discuss the procedure followed on the two-step implementation of our proposed protocol for remote hardware-integrity attestations.

1. The TPM computer that need to be verified by certifying that its hardware components have the anticipated (expected) configuration sends its ID (i.e., ID_{CC}),in order for the TCG Member to be able to search its public Endorsement Key (i.e., EK^{e}_{CC}), signed with its private Endorsement Key (i.e., EK^{d}_{CC}) the unique used value (sec_boot_val) of which it expects the signed hash as mentioned above, a time-stamp TS_{cc} to ensure the freshness of the message and avoid the potential replay attacks, and measurement lists (i.e., $\lambda_1, \lambda_2, \cdots, \lambda_n$) for each hardware component, all given in a particular order (i.e., CPU, Network Card, Motherboard, ..., etc), and its ID along with the signed message and its corresponding certificate ($CERT_{CC}$) are sent to the TCG Member encrypted with its public Endorsement Key (i.e., $EK^{e}_{TCG\_MB}$).

2. The TPM of TCG member decrypts the message, verifies the signature (TPM Signature) and locates the corresponding expected measurement list (i.e., λ^{XP}_1, λ^{XP}_2, $\cdots$, λ^{XP}_n) for each hardware component of the computer. Having a set of expected measurements for each hardware component generates as many symmetric keys as the cardinality of the expected measurement list's set. The lists are creating each key by calculating the hash of any expected list as $K_i = hash(\lambda^{XP}_i)\ \forall i, 1 \leq i \leq n$. Then the TPM of TCG Member computes the hash value (i.e., $hash(sec\_boot\_val)$) of the sec_boot_val and signs it with its private Endorsement Key (i.e, $EK^{d}_{TCG\_MB}$). Then, the TPM of TCG Member, sequentially encrypts this signed message using one-by-one all K keys created according to the above procedure. Then, the TPM of TCG Member signs the sequentially encrypted message (encrypts it with $EK^{d}_{TCG\_MB}$ i.e., its private Endorsement Key) and along with the certificate if its signature (i.e., $CERT_{TCG\_MB}$) and the unique value sec_boot_val (sends it again to prevent repeat attacks) re-sends them encrypted with its public Endorsement Key (EK^{e}_{CC})) of its TPM as challenge for verification of CC's hardware integrity. Though this challenge-response procedure it is guaranteed that if and only if the hardware of CC has not been compromised (i.e., its measurement list will be the expected) then it will be able to decrypt the message and send the response to TCG Member's challenge. The CC if and only if has computed the proper K keys implying the expected configuration on its hardware, can decrypt the message, and pass as input to the Secure Boot process the $hash(sec\_boot\_val)$ signed by the TPM of

TCG Member, launching the Secure Boot process.

9.5.7 Potentials

Throughout this work its is also proposed a protocol to eliminate the possibility of an attack based on the hardware's configuration changes (hardware-based attack) which could undermine the operation of the system and then the security of the data hosted on this. The solution proposed was the creation of an attestation method for hardware integrity which enables a commodity computer that has a built-in Trusted Platform Module (TPM) to prove through a challenge-response procedure that its hardware has not been compromised.The main points in this protocol is mainly the configuration of Secure Boot process to expect the signed by a TCG Member hash value of a unique sec_boot_val required to launch Secure Boot process, i.e., a property can be ensured by the same TPM since it handles the process, and then to develop a method which, utilizing hardware measurement lists, is plugged in between of system's startup and the launch of Secure Boot process.

9.6 System Integration

The increasing security threats on the protection of privacy, integrity and confidentiality of systems as also of the data stored in them constitute the key incentives for research and thorough study on information system security. Security of information systems is one of the most important issues of concern in maintaining the smooth and persistent operation of IT. This research proposal is developed in the field of protection against malicious software and the prevention of its spread, which consists the dominant tactic of cyber-attacks. Therefore, our basic aim is the in-depth and multi-level study on the protection against malicious software as also the prevention of its spread by proposing, so inside the scope of this thesis as also through its further extensions, effective graph-based algorithmic techniques which ensure the protection of privacy, integrity and confidentiality of the systems.

9.6.1 An Algorithmic Framework for Protection against Malicious Software

Considering the continued increase in the use of mobile devices, it is advisable to study the spread of malware among them. In this direction, we developed epidemiological models which, using information related to a particular subset of devices (critical nodes), will aim on coordinated prevention as also on tracking the spread through graph-based algorithmic techniques.

Approaching the problem, the methodology we follow develops an algorithmic framework consisting of two axes: protection against malicious software and prevention from its spreading between mobile devices. So, at the first level, it has already been studied, designed and finally developed a protection system that implement a set of algorithmic methods for detecting and classifying malware. On the second level, based on known epidemiological models represented by graphs depicting snapshots of the networks that are dynamically formed between the mobile devices, we developed a system to define the maximum permitted response time for a countermeasure to react in order to suppress the spread of malware between mobile devices, preventing finally its pandemic propagation. In order to achieve this goal, the ongoing research is developed on two main axes, as shown in Figure 9.3 - initially on the development of algorithmic techniques for protection against malicious software and then on the development of algorithmic techniques to prevent its spread between mobile devices as to avoid pandemics.

9.6.2 Methodology

In the concept of a clearly critical threat on the security of IT operation, we are called upon to investigate, recommend and implement techniques that, approaching the problem algorithmically, will provide protection against malicious software and also suppress to its spread between computing devices.

It is well known that one of the most important challenges in detecting and then classifying malicious software is the resilience of each technique against its mutations (strain variation), with significant success rates being occupied by the so-called behavioral techniques. The research team has studied and recently proposed such malware detection and classification techniques [7], utilizing System-call Dependency Graphs as representations of its behavior through specific abstractions of these graphs

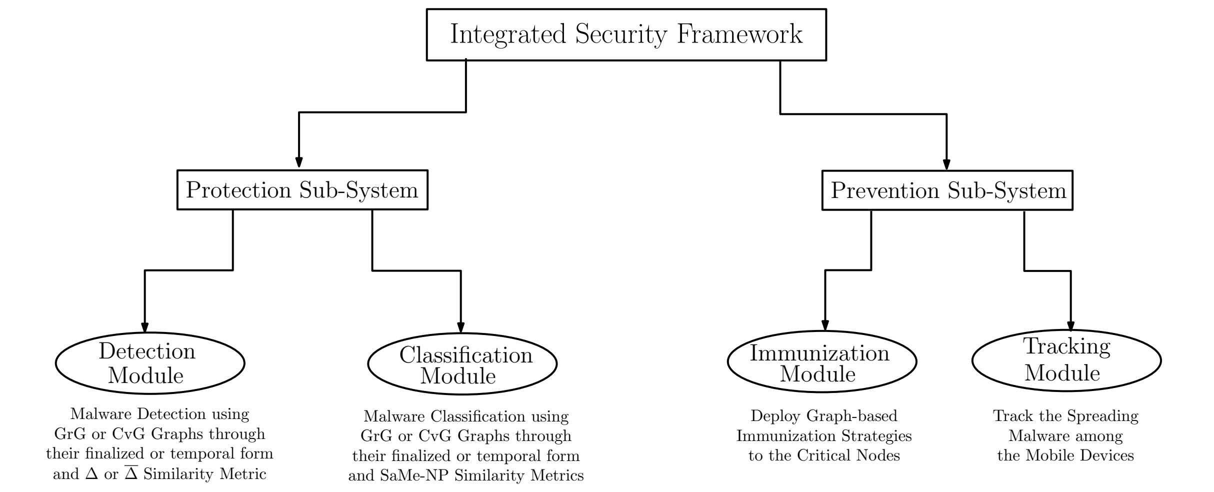

Integrated Security Framework
Protection Sub-System
Prevention Sub-System
Detection Module
Classification Module
Immunization Module
Tracking Module
Malware Detection using GrG or CvG Graphs through their finalized or temporal form and Δ or $\overline{\Delta}$ Similarity Metric
Malware Classification using GrG or CvG Graphs through their finalized or temporal form and SaMe-NP Similarity Metrics
Deploy Graph-based Immunization Strategies to the Critical Nodes
Track the Spreading Malware among the Mobile Devices

Figure 9.3: Architecture of the proposed algorithmic framework for protection against malware and pandemic prevention.

in mutant-resistant hyper-graphs.

Moreover, tt is widely accepted that prevention is an invaluable tactic, and therefore it is clearly intended to be applied in the case of suppressing the spread of malicious software between mobile computing devices. Having already studied the phenomenon, from the aspect of the influence of the counter-measure's response time to avoid pandemic spread, it is estimated that further study at the level of nodes of the network with the greatest influence on the spread of malicious software (critical nodes) would contribute significantly the research level of the field. Therefore, in the second part of the proposed algorithmic framework, we aim to incorporate innovative algorithmic techniques that, as an evolution of the already proposed ones, will initiate the launch of graph-based strategies targeting the immunization of critical nodes of the network, utilizing the position of computing devices in the dynamic network.

Towards the development of an integrated security framework, the principles developed so far in designing algorithmic techniques for protection against malicious software and algorithmic techniques for detecting and classifying digital objects, will be the components in the development of the proposed protection methods. Additionally, the credible methods of simulation, through graph-based approaches, of the malicious software's spreading environment developed so far and the study conducted concerning the effect of counter-measure's response-time on pandemic prevention provide the basis for expanding the research into the field of suppressing the spread of malicious software while ensuring the effectiveness of the proposed techniques.

9.6.3 Protection against Malicious Software through Graph-based techniques

On the first part of the integrated algorithmic framework, malware protection methods have their basis in algorithmic techniques aimed primarily at addressing the malware mutations throughout an abstract generalized representation, and then the abilities of detection and a further classification in a family of known malicious software. To achieve the primary goal of developing a structure that represents malicious software remaining unchanged after mutations of the strain, there will be incorporated the Temporal Graph instances of the Group Relation Graph (GrG) Graphs, as derivatives of the System-Call Dependency Graph (SCDG). GrG graphs have the ability to detect malware even if they have been mutated, since in order for the functionality of malware to be preserved, some system-call functions from specific System-call Groups, used in its primary form, will also be reused in the mutant strain. Hence, even in the case of switching/replacing System-calls from the same System-call group (i.e., the structure of the primary SCDG graph is modified), the interaction and consequently the association between the System-call Groups depicted by its corresponding GrG graph will remain unaltered [7]. Hence, evolving the existing research approach, based on the use of similarities metrics between GrG in order to implement our proposed malware detection and classification techniques, the research will focus on the utilization of these components and their further integration into a system that would automatically deploy the corresponding processes of the proposed design are depicted in Figure ??.

9.6.4 Pandemic Prevention through Graph-based Immunization Strategies

It is obvious that the most effective defense strategy in the evolution (potentially pandemic) of an epidemiological phenomenon, in our case the spreading of malicious software, is the method of prevention. Due to the increased use of mobile devices, we focus on developing, on the second part of the proposed algorithmic framework, graph-based strategies that, based on known epidemiological models, aim on the immunization of nodes/devices that express a specific set of properties, in order to prevent a pandemic in cases of malware spreading between mobile devices.

Utilizing real spatial data through satellite images to represent the urban structure of a city, we assign points of variable attraction to different locations of its urban composition, representing it as a non-directional weighted graph, in which shortest path algorithms simulate the movement of mobile devices. Based on such modeling, we have developed Device Mobility Models and Malware Propagation Models that simulate the motion of devices and the corresponding dynamic links formed between them as they move into the city [102]. Additionally, applying epidemiological models describing different types of spread, we incorporate the proposed algorithmic technique that, given the total number of devices, the size of the infected and susceptible population, and the graph representation of a city, determines the upper permitted time limit within which a countermeasure can respond effectively avoiding the pandemic of malware between susceptible mobile devices.

Graph-based Strategies. In the research area of network security, there is an important knowledge basis that describes information concerning the properties of network nodes as they are expressed by their influence on the dissemination of information within the network (i.e., adopting an idea, information transmission , spreading an epidemic, etc.). A set of nodes whose endogenous characteristics define a behavior-profile suggesting that they have an increased impact on the considered critical nodes and for the process of finding them have been proposed various algorithmic techniques, mostly adopted and used in the fields of both computer and social networks. Extending our research approach, we inspect the feasibility of extending the pandemic prevention model through the use of additional information concerning the interconnection between nodes in dynamic networks. In the second part of our proposed algorithmic framework, the study focuses on exploring the scope of protection of such dynamical networks with respect to the subset of nodes which seem to be the most influential in the spread of malicious software (critical nodes) and therefore appear to have higher degree of risk. A key component of this development is the use of information related to the vulnerabilities that specific nodes present as also related to their topology regarding its effect on the spread of malicious software. Our main goal is the development of algorithmic techniques that will control and determine the coordinated launching of immunization strategies on the critical nodes, utilizing the existing graph-based approaches. More specifically, we aim on developing techniques that, by identifying critical nodes in a network of mobile devices, will coordinate the deployment of graph-based strategies for the cure and / or immunization against the spread of malware to ultimately suppress its spread and avoid a pandemic.

Chapter 10

Conclusion

10.1 Protection against Malicious Software

In this work there have been proposed several graph-based techniques to build the defense line against malicious software. In order to perform the elementary task of detection the maliciousness of an unknown software sample and proceed to its further classification to a malware family since it has been detected as malicious the main goal is to compare the unknown object against something that is known to be malicious. The digital object throughout this thesis is set to be represented by a graph-object that in some fashion depicts its behavior (i.e., since in that case the digital object is a software, its behavior is depicted with its interaction with its host environment - O.S.). The set of what is known so far to be malicious (or not) has been described as the knowledge base, from which, elements are selected in order to be compared using the proposed graph-based similarity techniques with the unknown sample.

The proposed similarity techniques presented on this thesis have been designed over the scope of the utilization of specific characteristics exhibited in behavioral graphs and are referenced namely as relational, quantitative, qualitative, and evolutional characteristics, and have been leveraged as to measure the similarity by assigning in some fashion different weights on different types of factors that characterize the nature of an edge of a behavioral graph, taking into account various information resulted over the inspection of the graph's properties. Namely, the proposed graph-based similarity metrics are the Δ-Similarity metric, the $\overline{\Delta}$-Similarity Metric which are based on the δ- and $\overline{\delta}$- distances respectively, the SaMe Similarity Metric and the NP Similarity Metric. Additionally, the main goals of this work, i.e., malware detection and classification, and pandemic prevention are discussed in order to make clear the focus of the research by the aspect of its theoretical background.

10.2 Graph-based Techniques for Malicious Software Detection and Classification

The core component of this thesis on the axis of development efficient algorithmic techniques for malware detection and classification, is the proposal of generalized graph-structures that represent malware's behavior in such a way that even in extreme mutation setting they would be able to preserve their structural characteristics that depict malware's behavior regarding its functionality, achieving finally high detection and classification accuracy. In this thesis, there have been proposed three types of graphs, namely, the Group Relation Graph, the Coverage Graph and the Temporal Graphs. The Group Relation Graph, or, for short GrG, results after merging disjoint vertices of its ancestor graph, the so called System-call Dependency Graph, or for short ScDG, as it is discussed in Section 3 and has the ability (as shown from the experimental evaluation) to maintain its detection and classification accuracy against its "not easy to manage" and mutation fragile ancestor ScDG graph. Next, the Coverage Graph, or, for short CvG, is presented as an evolution of GrG, which represents the dominating relations exhibited between the vertex set of GrG regarding their in/out weights and degrees. Finally, an innovative graph representation at the specific field of malware detection and classification is proposed through this thesis, aiming to catch the structural evolution of a GrG or CvG graph, the so called Temporal Graphs.

Such graphs demand the computation of the proposed similarity metrics over all their instances, that are defined as epochs and due to the resulted space complexity are hence more difficult to store and manage when compared to GrG or CvG graphs.

In order to perform the procedures of malware detection and classification, through this thesis there have been designed, implemented and evaluated two graph-based models for malware detection and classification based on the proposed graph structures and the corresponding graph-based similarity metrics. In this thesis there have been presented and implemented the architectural properties demanded to develop a system that, given an unknown software sample, efficiently and effectively would detect if it is malicious or not, and then classify it to a malware family. More precisely, based on the computation of its similarity, (i.e., utilizing the Δ- or the $\overline{\Delta}$-Similarity metrics) between the generalized graph representation of the test sample and the set of known malware samples stored in th knowledge base (i.e., their corresponding GrG, or CvG, or Temporal Graphs) the proposed system is able to distinguish malicious from benign unknown software samples and to a further extent, if they have been detected as malicious, to classify them (i.e., utilizing the SaMe- or the NP-Similarity metrics) to one of a set of known malware families.

10.3 Pandemic Prevention from the Malware spreading to Proximal Mobile Devices

In the scope of this thesis, there have been proposed specific models to simulate the spread of a malicious software between proximal mobile devices and to develop further an approach to prevent the pandemic of the spreading malware based on the effect that has on it the time required from a countermeasure to take effect in order to suppress malware's propagation, the so called response-time. Specifically, through the thesis it is proposed a graph-based model to establish the maximum permitted time interval (i.e., response-time bound) inside which a counter measure may react effectively in order to prevent pandemic. More precisely there have on the scope of this thesis there have been proposed a graph-based model for simulating town's planning, a graph-based model for simulating the mobility patterns of the mobile devices over a city area and a graph-based model for simulating the propagation patterns that follow the underlying compartmental epidemic model. Incorporating these three models, it has been proposed an integrated framework to establish the maximum permitted response-time bound required from a countermeasure to take effect in order to prevent pandemic. In order to tune the overall system that integrates the aforementioned models a simulator that incorporates the has been implemented and by involving several other parameters that affect the spread of the malicious software between proximal mobile devices (i.e., malware's size, device range, network density, initial infected population etc.) is able, given a set of parameters to decide if the pandemic could be avoided or not.

10.4 Model Evaluation: Potentials and Limitations

The evaluation of the proposed models regarding the accuracy of the detection and classification procedures have been performed over a data-set of more that 2500 malware samples pre-classified. or equivalently indexed, according to the widest in use antivirus industries and their respective heuristic rules into 48 malware families, and also a set of 35 benign samples ranging to various commodity software type that cover the wider range of desktop applications. The data-set that actually consists the knowledge base to evaluate the proposed model is that same as the one used in [5], and in order to perform the evaluation procedures it has been divided into 5 sub-sets as to proceed by a five-fold cross validation procedure which is the most applicable and would made the achieved results comparable to the ones exhibited by other approaches regarding the detection and respectively the classification procedures. The results achieved averaging the detection and classification accuracy over the five-folds where quite positive since in many cases overcame the ones achieved from other graph-based and non-graph based approaches proving the potentials of the proposed detection and classification techniques.

On the other hand, in order to evaluate the proposed model for pandemic prevention over the propagation of a malicious software that spreads between proximal mobile devices, a series of Monte Carlo simulation experiment where performed in order to eliminate as much as possible the non deterministic character of the experiments, referencing mostly the probabilistic type of the so called device mobility model. Particularly, for several values of other factors that also affect the malware's spread, namely, malware's size, device range, network density and the size of the initially infected population the proposed model established successfully the upper bounds concerning the maximum time permitted for a countermeasure to take effect in order to sanitize the infected devices suppressing the spread and finally achieve pandemic prevention.

10.4.1 Refining Coverage-Similarity Metric

As we observe from the obtained results, regardless the underlying computation on the weight of each node and the variances occurred through different values of threshold, the detection rates behave in both case as been almost stabilized. In Figure 10.1 we present an overall comparison of the detection rates achieved by our proposed graph-based model for malware detection, where the detection accuracy of the two approaches concerning vertex weight measurement has been shown for selected values of threshold. As we can observe from this comparative representation the approach following the sum of the weights on its vertex performs better, regarding the detection rates, over the one that computes the average value of the weight over the degree of this vertex despite that the FP rates where even lower. An interesting aspect over the analysis of our experimental results could be developed on how outlier observations affect our first approach concerning the measurement of the weight on each vertex (i.e., sum) and how we could enforce our second approach (i.e., mean) as to be more resilient to outlier observations that strongly affect the produced domination set for each vertex.

10.4.2 Deployment and Manageability of Temporal Graphs

Next are discussed the potentials and limitations concerning the deployment of the Temporal Graphs and their utilization over the proposed similarity metrics as also their limitations referencing the manageability regarding the required storage when having unbalanced the trade-off set by the tuning of epoch sizes.

Potentials. Several modeling alternates have been arise during the theoretical construction of our graph-based proposed model regarding the temporal evolution of behavioral graphs that represent software samples, regarding their structural modification during time. Our approaches that we discuss briefly next, mostly concern the representation of the structural modifications on the GrG and CvG graphs during time, and how they could also be represented with other structures that do not co-

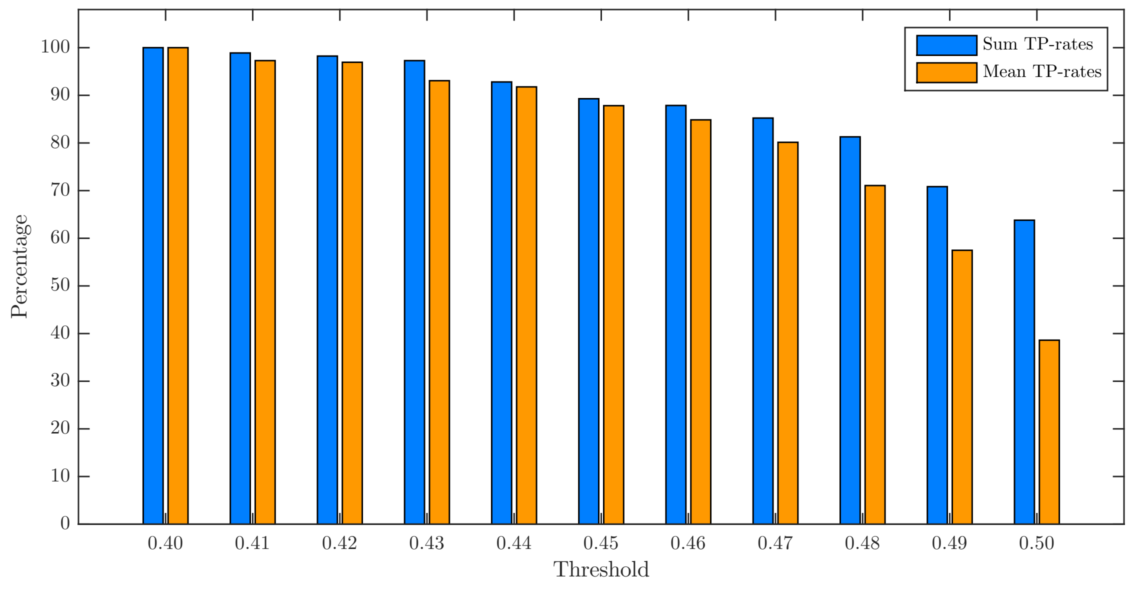


Figure 10.1: Detection rates for multiple values of threshold achieved by applying the sum and the mean on the computation of the weight on each vertex.

operate graphs, and consequently deserve the application of different manipulation methods.

In the first alternate approach, we could denote the structural evolution of a given by plotting by a discrete distribution of the addition of edges over the graph on specific time buckets (i.e., similar to `epochs`) and create patterns that could be utilized in order to perform pattern-matching over the plot of any given pair of samples (i.e., test and known malware sample). These plots should be construct for the temporal evolution of each corresponding edge pair of two given graphs in order for the patterns to be comparable.

On the other hand, in the second approach of our model, we need to simulate the structural modification of a given graph during time (i.e., temporal evolution of the graph). Similarly to our approach, rather than constructing several graph instances equal to the number of the defined `epochs` and structurally relevant to the applied method regarding the discrete or cumulative modification approach, we could also represent these structural modification (i.e., addition of edges) over the time for each edge (i.e., edge on either GrG or CvG). More precisely, we could define a binary sequence for each edge, where 0 denotes absence and 1 denote adition of this edge on the overall graph, and the length of the sequence equals the size of the ScDG (i.e., System-call Depenency Graph). Then, various alignment algorithms could be adopted in order to retrieve similarity patterns among any pair of such sequences, that represent corresponding edges on the graphs of the test and the known malicious samples.

Limitations. Our proposed graph-based model for malware detection and classification using temporal graphs, despite its theoretical basis, has also some limitations concerning any implementation drawbacks that may arise. The main issue encountered regarding the implementation design concerns the spatial complexity of our approach. More precisely, defining a fine-grained or a coarse-grained quantization of time (i.e., number of `epochs`) would affect to a great extent the space required to store the corresponding Temporal Graph instances. AS easily someone can understand, an implementation of our proposed model on a fine-grained time quantization scheme, would be more precise against a more coarse-grained once. Additionally, further tuning issues arise over the trade-off between the precision on temporal structural modifications and the construction of more distinguishing patterns. However, more sophisticated approaches, such an implementation that utilizes the maximum length of a binary tree in order to bound the quantization would lead to a more stable, rational, effective and efficient approach.

10.5 Model Alignment and System Integration

In order to integrated the approaches proposed throughout this thesis, an algorithmic framework for ensuring security against malicious software is also proposed. The framework is designed having its basis on the theoretical background that rules the proposed graph-based similarity techniques (i.e., the Δ-Similarity metric, the $\overline{\Delta}$-Similarity Metric, the SaMe Similarity Metric and the NP Similarity Metric) that compute the similarity between any given pair of digital objects that are represented utilizing the proposed graph structures (i.e., Group Relation Graphs, Coverage Graphs and Temporal Graphs) where the detection and classification procedure rely on. On the other hand the utilization of trusted computing alongside with the implementations of graph-based strategies for establishing early warning concerning counter measure's response-time, cooperates in order to build a set pf graph-based principles that would deploy effective strategies in order to not only avoid pandemic caused by malware's spread but to a further extent to efficiently suppress malware's spread.

10.6 Further Research

As a prime further research target there has left the development of the algorithmic framework for integrated protection against malicious software described on the previous section. Moreover, on the same goal is embedded the investigation of utilizing different combinations of similarity metrics over different types of behavioral graphs as also the evaluation of the proposed similarity metrics over the Temporal Graphs utilizing both GrG and CvG graphs. Additionally on the same concept several string alignment algorithms could also be deployed as a result of the evolution of Temporal Graphs over their corresponding `epochs` where the exploration of such informative data would be valuable indeed.

Bibliography

[1] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Control flow graphs as malware signatures," in *International workshop on the Theory of Computer Viruses*, 2007.

[2] S. Cesare and Y. Xiang, "A fast flowgraph based classification system for packed and polymorphic malware on the endhost," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pp. 721–728, IEEE, 2010.

[3] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in computer virology*, vol. 7, no. 4, pp. 233–245, 2011.

[4] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host.," in *USENIX security symposium*, vol. 4, pp. 351–366, 2009.

[5] D. Babić, D. Reynaud, and D. Song, "Malware analysis with tree automata inference," in *International Conference on Computer Aided Verification*, pp. 116–131, Springer, 2011.

[6] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *2010 IEEE Symposium on Security and Privacy*, pp. 45–60, IEEE, 2010.

[7] S. D. Nikolopoulos and I. Polenakis, "A graph-based model for malware detection and classification using system-call groups," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 29–46, 2017.

[8] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.

[9] P. Szor and P. Ferrie, "Hunting for metamorphic," in *Virus bulletin conference*, Prague, 2001.

[10] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: from encryption to metamorphism," *International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74–83, 2012.

[11] M. Mungale and M. Stamp, "Software similarity and metamorphic detection," in *Proceedings of the International Conference on Security and Management (SAM)*, p. 1, The Steering Committee of The World Congress in Computer Science, Computer ..., 2012.

[12] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, 2006.

[13] M. Alazab, R. Layton, S. Venkataraman, and P. Watters, "Malware detection based on structural and behavioural features of api calls," 2010.

[14] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 32–46, IEEE, 2005.

[15] K. Mathur and S. Hiranwal, "A survey on techniques in detection and analyzing malware executables," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 4, 2013.

[16] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM computing surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[17] K. Kendall and C. McMillan, "Practical malware analysis," in *Black Hat Conference, USA*, p. 10, 2007.

[18] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," tech. rep., WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

[19] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 421–430, IEEE, 2007.

[20] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *2007 IEEE Symposium on Security and Privacy (SP'07)*, pp. 231–245, IEEE, 2007.

[21] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[22] 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)," in *2010 IEEE Symposium on Security and Privacy*, pp. 317–331, IEEE, 2010.

[23] L. Aneja and S. Babbar, "Research trends in malware detection on android devices," in *International Conference on Recent Developments in Science, Engineering and Technology*, pp. 629–642, Springer, 2017.

[24] M. Elingiusti, L. Aniello, L. Querzoni, and R. Baldoni, "Malware detection: A survey and taxonomy of current techniques," *Cyber Threat Intelligence*, pp. 169–191, 2018.

[25] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European Symposium on Research in Computer Security*, pp. 62–79, Springer, 2017.

[26] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.

[27] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, p. 3, 2018.

[28] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.

[29] B. Alsulami, A. Srinivasan, H. Dong, and S. Mancoridis, "Lightweight behavioral malware detection for windows platforms," in *2017 12th International*

Conference on Malicious and Unwanted Software (MALWARE), pp. 75–81, IEEE, 2017.

[30] M. L. Bernardi, M. Cimitile, D. Distante, F. Martinelli, and F. Mercaldo, "Dynamic malware detection and phylogeny analysis using process mining," *International Journal of Information Security*, pp. 1–28, 2018.

[31] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 611–620, ACM, 2009.

[32] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors.," in *LEET*, 2009.

[33] R. M. H. Ting and J. Bailey, "Mining minimal contrast subgraph patterns," in *Proceedings of the 2006 SIAM International Conference on Data Mining*, pp. 639–643, SIAM, 2006.

[34] K. Kim and B.-R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 1211–1218, ACM, 2010.

[35] B. David, E. Filiol, and K. Gallienne, "Structural analysis of binary executable headers for malware detection optimization," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 2, pp. 87–93, 2017.

[36] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in computer Virology*, vol. 4, no. 3, pp. 251–266, 2008.

[37] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 5–14, ACM, 2007.

[38] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering.," in *NDSS*, vol. 9, pp. 8–11, Citeseer, 2009.

[39] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 108–125, Springer, 2008.

[40] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2018.

[41] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *European Symposium on Research in Computer Security*, pp. 326–343, Springer, 2003.

[42] R. Tian, L. M. Batten, and S. Versteeg, "Function length as a tool for malware classification," in *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 69–76, IEEE, 2008.

[43] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*, pp. 297–300, IEEE, 2010.

[44] L. S. Grini, A. Shalaginov, and K. Franke, "Study of soft computing methods for large-scale multinomial malware types and families detection," in *Recent Developments and the New Direction in Soft-Computing Foundations and Applications*, pp. 337–350, Springer, 2018.

[45] B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, "Empowering convolutional networks for malware classification and analysis," in *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 3838–3845, IEEE, 2017.

[46] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, "Fast malware classification by automated behavioral graph matching," in *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, p. 45, ACM, 2010.

[47] J. Moubarak, M. Chamoun, and E. Filiol, "Comparative study of recent mea malware phylogeny," in *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, pp. 16–20, IEEE, 2017.

[48] J. Liu, P. Dai Xie, M. Z. Liu, and Y. J. Wang, "Having an insight into malware phylogeny: Building persistent phylogeny tree of families," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 4, pp. 1199–1202, 2018.

[49] Y. S. Sun, C.-C. Chen, S.-W. Hsiao, and M. C. Chen, "Antsdroid: Automatic malware family behaviour generation and analysis for android apps," in *Australasian Conference on Information Security and Privacy*, pp. 796–804, Springer, 2018.

[50] J. Liu, Y. Wang, P. Dai XIE, and Y. J. Wang, "Inferring phylogenetic network of malware families based on splits graph," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 6, pp. 1368–1371, 2017.

[51] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, pp. 13–23, 2005.

[52] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny using maximal pi-patterns," in *EICAR 2005 Conference: Best Paper Proceedings*, pp. 156–174, 2005.

[53] A. Walenstein and A. Lakhotia, "The software similarity problem in malware analysis," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[54] M. Avlonitis, E. Magkos, M. Stefanidakis, and V. Chrissikopoulos, "Treating scalability and modelling human countermeasures against local preference worms via gradient models," *Journal in computer virology*, vol. 5, no. 4, p. 357, 2009.

[55] M. Avlonitis, E. Magkos, M. Stefanidakis, and V. Chrissikopoulos, "A spatial stochastic model for worm propagation: scale effects," *Journal in Computer Virology*, vol. 3, no. 2, pp. 87–92, 2007.

[56] P. A. Dreyer Jr and F. S. Roberts, "Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion," *Discrete Applied Mathematics*, vol. 157, no. 7, pp. 1615–1627, 2009.

[57] J. Kleinberg and D. Easley, "Networks, crowds, and markets: Reasoning about a highly connected world," *Cambridge University Press. C*, vol. 1, no. 2, p. 3, 2010.

[58] G. Yan, G. Chen, S. Eidenbenz, and N. Li, "Malware propagation in online social networks: nature, dynamics, and defense implications," in *Proceedings of the 6th ACM symposium on information, computer and communications security*, pp. 196–206, ACM, 2011.

[59] M. Garetto, W. Gong, and D. Towsley, "Modeling malware spreading dynamics," in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, vol. 3, pp. 1869–1879, IEEE, 2003.

[60] M. Salathé, M. Kazandjieva, J. W. Lee, P. Levis, M. W. Feldman, and J. H. Jones, "A high-resolution human contact network for infectious disease transmission," *Proceedings of the National Academy of Sciences*, vol. 107, no. 51, pp. 22020–22025, 2010.

[61] R. Luh and P. Tavolato, "Behavior-based malware recognition," *Forschungsforum der österreichischen Fachhochschulen*, 2012.

[62] A. Makandar and A. Patrot, "Trojan malware image pattern classification," in *Proceedings of International Conference on Cognition and Recognition*, pp. 253–262, Springer, 2018.

[63] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 239–248, ACM, 2017.

[64] L. Sikora and I. Zelinka, "Swarm virus, evolution, behavior and networking," in *Evolutionary Algorithms, Swarm Dynamics and Complex Networks*, pp. 213–239, Springer, 2018.

[65] Y. Ding, X. Xia, S. Chen, and Y. Li, "A malware detection method based on family behavior graph," *Computers & Security*, vol. 73, pp. 73–86, 2018.

[66] S. D. Mukesh, J. A. Raval, and H. Upadhyay, "Real-time framework for malware detection using machine learning technique," in *International Conference*

on Information and Communication Technology for Intelligent Systems, pp. 173–182, Springer, 2017.

[67] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: Intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1043–1047, ACM, 2007.

[68] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 611–620, ACM, 2009.

[69] R. Islam, R. Tian, L. Batten, and S. Versteeg, "Classification of malware based on string and function feature selection," in *2010 Second Cybercrime and Trustworthy Computing Workshop*, pp. 9–17, IEEE, 2010.

[70] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*, p. 4, ACM, 2011.

[71] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang, "A comparative assessment of malware classification using binary texture analysis and dynamic analysis," in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, pp. 21–30, ACM, 2011.

[72] D. Balcan, V. Colizza, B. Gonçalves, H. Hu, J. J. Ramasco, and A. Vespignani, "Multiscale mobility networks and the spatial spreading of infectious diseases," *Proceedings of the National Academy of Sciences*, vol. 106, no. 51, pp. 21484–21489, 2009.

[73] G. Yan, T. Zhou, J. Wang, Z.-Q. Fu, and B.-H. Wang, "Epidemic spread in weighted scale-free networks," *arXiv preprint cond-mat/0408049*, 2004.

[74] K. Scaman, A. Kalogeratos, and N. Vayatis, "A greedy approach for dynamic control of diffusion processes in networks," in *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 652–659, IEEE, 2015.

[75] C. Kamp, M. Moslonka-Lefebvre, and S. Alizon, "Epidemic spread on weighted networks," *PLoS computational biology*, vol. 9, no. 12, p. e1003352, 2013.

[76] M. J. Keeling and K. T. Eames, "Networks and epidemic models," *Journal of the Royal Society Interface*, vol. 2, no. 4, pp. 295–307, 2005.

[77] E. Volz and L. A. Meyers, "Susceptible–infected–recovered epidemics in dynamic contact networks," *Proceedings of the Royal Society B: Biological Sciences*, vol. 274, no. 1628, pp. 2925–2934, 2007.

[78] W. Liu, C. Liu, X. Liu, S. Cui, and X. Huang, "Modeling the spread of malware with the influence of heterogeneous immunization," *Applied mathematical modelling*, vol. 40, no. 4, pp. 3141–3152, 2016.

[79] W. Liu, C. Liu, Z. Yang, X. Liu, Y. Zhang, and Z. Wei, "Modeling the propagation of mobile malware on complex networks," *Communications in Nonlinear Science and Numerical Simulation*, vol. 37, pp. 249–264, 2016.

[80] S. Hosseini, M. A. Azgomi, and A. T. Rahmani, "Malware propagation modeling considering software diversity and immunization," *Journal of computational science*, vol. 13, pp. 49–67, 2016.

[81] P. Van den Driessche and J. Watmough, "Reproduction numbers and sub-threshold endemic equilibria for compartmental models of disease transmission," *Mathematical biosciences*, vol. 180, no. 1-2, pp. 29–48, 2002.

[82] Z. Chen and C. Ji, "Spatial-temporal modeling of malware propagation in networks," *IEEE Transactions on Neural networks*, vol. 16, no. 5, pp. 1291–1303, 2005.

[83] A. Bose and K. G. Shin, "On mobile viruses exploiting messaging and bluetooth services," in *2006 Securecomm and Workshops*, pp. 1–10, IEEE, 2006.

[84] C. Fleizach, M. Liljenstam, P. Johansson, G. M. Voelker, and A. Mehes, "Can you infect me now?: malware propagation in mobile phone networks," in *Proceedings of the 2007 ACM workshop on Recurring malcode*, pp. 61–68, ACM, 2007.

[85] K. Channakeshava, D. Chafekar, K. Bisset, V. Kumar, and M. Marathe, "Epinet: a simulation framework to study the spread of malware in wireless networks," in *proceedings of the 2nd international conference on simulation tools and techniques*, p. 6, ICST (Institute for Computer Sciences, Social-Informatics and ..., 2009.

[86] N. R. Zema, E. Natalizio, G. Ruggeri, M. Poss, and A. Molinaro, “Medrone: On the use of a medical drone to heal a sensor network infected by a malicious epidemic,” *Ad Hoc Networks*, vol. 50, pp. 115–127, 2016.

[87] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in *International Workshop on Recent Advances in Intrusion Detection*, pp. 207–226, Springer, 2005.

[88] G. Jacob, H. Debar, and E. Filiol, “Behavioral detection of malware: from a survey towards an established taxonomy,” *Journal in computer Virology*, vol. 4, no. 3, pp. 251–266, 2008.

[89] S. D. Nikolopoulos and I. Polenakis, “A graph-based model for malware detection and classification using system-call groups,” *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 29–46, 2017.

[90] Y. Chen, V. Paxson, and R. H. Katz, “What’s new about cloud computing security,” *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, vol. 20, no. 2010, pp. 2010–5, 2010.

[91] K. V. Madhavi, R. Tamilkodi, and K. J. Sudha, “Cloud computing: Security threats and counter measures,” *International Journal of Research in Computer and Communication technology, IJRCCT*, vol. 1, no. 4, pp. 125–128, 2012.

[92] V. K. Reddy, B. T. Rao, and L. Reddy, “Research issues in cloud computing,” *Global Journal of Computer Science and Technology*, 2011.

[93] A. E. Youssef and M. Alageel, “A framework for secure cloud computing,” *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 4, p. 487, 2012.

[94] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, “Mobile trusted computing,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1189–1206, 2014.

[95] B. Parno, J. M. McCune, and A. Perrig, “Bootstrapping trust in commodity computers,” in *2010 IEEE Symposium on Security and Privacy*, pp. 414–429, IEEE, 2010.

[96] Z. Shen, L. Li, F. Yan, and X. Wu, "Cloud computing system based on trusted computing platform," in *2010 International Conference on Intelligent Computation Technology and Automation*, vol. 1, pp. 942–945, IEEE, 2010.

[97] X.-Y. Li, L.-T. Zhou, Y. Shi, and Y. Guo, "A trusted computing environment model in cloud architecture," in *2010 International Conference on Machine Learning and Cybernetics*, vol. 6, pp. 2843–2848, IEEE, 2010.

[98] M. Sanjay Ram and V. Vijayaraj, "Analysis of the characteristics and trusted security of cloud computing," *International Journal on Cloud Computing*, vol. 1, pp. 61–69, 2011.

[99] Z. Shen and Q. Tong, "The security of cloud computing system enabled by trusted computing technology," in *2010 2nd International Conference on Signal Processing Systems*, vol. 2, pp. V2–11, IEEE, 2010.

[100] F. J. Krautheim, D. S. Phatak, and A. T. Sherman, "Introducing the trusted virtual environment module: a new mechanism for rooting trust in cloud computing," in *International Conference on Trust and Trustworthy Computing*, pp. 211–227, Springer, 2010.

[101] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing.," *HotCloud*, vol. 9, no. 9, p. 3, 2009.

[102] S. D. Nikolopoulos and I. Polenakis, "A graph-based model for malware detection and classification using system-call groups," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 29–46, 2017.

Author's Publications

- Mpanti, A., Nikolopoulos, S. D., and Polenakis, I.: Malicious Software Detection utilizing Temporal-Graphs. (Accepted) To be presented in the 20th International Conference on Computer Systems and Technologies 2019 ACM.
- Mpanti, A., Nikolopoulos, S. D., & Polenakis, I.: Malicious Software Detection and Classification utilizing Temporal-Graphs of System-call Group Relations. arXiv preprint arXiv:1812.10748, 2018
- Mpanti, A., Nikolopoulos, S. D., & Polenakis, I.: A Graph-based Model for Malicious Software Detection Exploiting Domination Relations between System-call Groups. In Proceedings of the 19th International Conference on Computer Systems and Technologies 2018 ACM. - BEST PAPER AWARD
- Nikolopoulos, S. D. and Polenakis, I.: "Preventing malware pandemics in mobile devices by establishing response-time bounds". Journal of Information Security and Applications, Elsevier, 2017, 1-14
- Mpanti, A., Nikolopoulos, S. D. and Polenakis, I.: "Defending Hardware-based Attacks on Trusted Computing using a Hardware-Integrity Attestation Protocol". In: Proceedings of the 18th International Conference on Computer Systems and Technologies 2017 ACM.
- Nikolopoulos, S. D. and Polenakis, I. "Preventing Malware Pandemics in Mobile Devices by Establishing Response-time Bounds". arXiv preprint arXiv:1607.00827, 2016
- Nikolopoulos S. D. and Polenakis, I.: "A Model for Establishing Response-time Bounds to Prevent Malware Pandemics in Mobile Devices". In Proceedings of the 17th International Conference on Computer Systems and Technologies, 2016 ACM - BEST PAPER AWARD

- Nikolopoulos, S. D. and Polenakis I.: ”A graph-based model for malware detection and classification using system-call groups”. In Journal of Computer Virology and Hacking Techniques, Springer, 2016, 1-18.
- Nikolopoulos S. D., and Polenakis, I.: ”Malicious software classification based on relations of system-call groups”. In Proceedings of the 19th Panhellenic Conference on Informatics (pp. 59-60), 2015 ACM.
- Nikolopoulos S. D. and Polenakis, I.: ”A graph-based model for malicious code detection exploiting dependencies of system-call groups”. In Proceedings of the 16th International Conference on Computer Systems and Technologies (pp. 228-235), 2015 ACM - BEST PAPER AWARD
- Nikolopoulos S.D. and Polenakis I.: ”Detecting Malicious Code by Exploiting Dependencies of System-call Groups, arXiv preprint arXiv:1412.8712, 2014.
- Chionis I., Nikolopoulos S.D., and Polenakis I.: ”A Survey on Algorithmic Techniques for Malware Detection”, In 2nd International Symposium on Computing in Informatics and Mathematics ISCIM, 2013.

Short Biography

Iosif Polenakis was born in Athens in 1990. In 2008 he was introduced in the Department of Informatics of the Ionian University, from which he graduated in 2012 with an excellent score of 8.78 / 10 and having received during these years scholarships and awards from the State Scholarship Foundation. His Bachelor Thesis was conducted under the supervision of Assistant Professor Emmanuel V. Magos, on the experimental study of the spread of malicious software between handheld devices.

In 2012 he received his postgraduate degree in the Department of Computer Science and Engineering of the Polytechnic School of the University of Ioannina, acquiring the postgraduate degree (MSc) in Theory of Computer Science with a degree of 9.15/10. His Master Thesis, entitled ”Algorithmic Techniques of Detection and Classification of Malicious Software”, was conducted under the supervision of Professor Stavros D. Nikolopoulos.

After his graduation of his MSc degree, he was admitted as a PhD Candidate at the same department, in order to conduct his PhD thesis, entitled ”Algorithmic Techniques of Detection and Classification of Digital Objects”, under the supervision of Professor Stavros D. Nikolopoulos. During the Ph.D. dissertation, his main research was performed over the protection against malicious attacks and the defense against malicious software, while his work has been published in international conferences, awarding to some of them best paper awards, as well as internationally recognized scientific journals of this scientific field, most of which have received significant references. Additionally, this PhD dissertation, as a research proposal, was selected for funding by achieving the 3^{rd} place in the Computer Engineering sector during the 1^{st} call of Hellenic Foundation for Research and Innovation (H.F.R.I.) for PhD Candidates.

In the sector of education, since 2015 he has been a laboratory supervisor in the undergraduate course of Design and Analysis of Algorithms of the Department of Computer Science and Engineering of the University of Ioannina, as well as a professor of computer science in vocational training institutes. In 2018 he was admitted as Adjunct Lecturer (academic scholar) in the Department of Informatics and Telecommunications of the University of Ioannina.

His research interests focus on information security, and more specifically on the research field of coping and preventing the spread of malicious software, while there is a strong interest in both cryptography and trusted systems.