UNIVERSITY OF PATRAS
DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

DIVISION: ELECTRONICS AND COMPUTERS

Thesis

of the Electrical and Computer Engineering Department of the Polytechnic School of the University of Patras student

Vasileios Tsouvalas

Registry Number: 227950

Subject

"Malware Classification Methodologies"

Supervisor

Dimitrios Serpanos

Thesis Number

227950/2020

Patras, October 2020

CERTIFICATION

It is hereby certified that the Thesis with subject

"Malware Classification Methodologies"

of the Department of Electrical Engineering student

Vasileios Tsouvalas

(R.N.: 227950)

was publicly presented and supported at the Department of Electrical and Computer Engineering on

..../..../....

Supervisor	Head of Division
Dimitrios Serpanos	Vasilis Paliouras
Professor	*Professor*

Thesis Number: 227950/2020

Subject: "Malware Classification Methodologies"

Student

Vasileios Tsouvalas

Supervisor

Dimitrios Serpanos

Abstract

Malware detection refers to the classification of a software as malicious or benign. Many attempts, employing diverse techniques, have been tried to tackle this issue. In the present thesis, we present a graph-based solution to the malware detection problem, which implements resources extraction from executable samples and applies machine learning algorithms to those resources so as to decide on the nature of the executable (malicious or benign). Given an unknown Windows executable sample, we first extract the calls that the sample makes to the Windows Application Programming Interface (API) and arrange them in the form of an API Call Graph, based on which, an Abstract API Call Graph is constructed. Subsequently, using a Random Walk Graph Kernel, we are able to quantify the similarity between the graph of the unknown sample and the corresponding graphs hailing from a labeled dataset of known samples (benign and malicious Windows executables), in order to carry out the binary classification using Support Vector Machines. Following the aforementioned process, we achieve accuracy levels up to 98.25%, using a substantially smaller dataset than the one proposed by similar efforts, while being considerably more efficient in time and computational power.

Keywords: Malware Detection, Machine Learning, SVM, Graphs, Graph Kernel



ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ: ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών της Πολυτεχνικής Σχολής του Πανεπιστημίου Πατρών

Βασιλείου Τσουβάλα του Κωνσταντίνου

Αριθμός Μητρώου: 227950

Θέμα

«Μεθοδολογίες Ταξινόμησης Κακόβουλου Λογισμικού»

Επιβλέπων

Δημήτριος Σερπάνος

Αριθμός Διπλωματικής Εργασίας

227950/2020

Πάτρα, Οκτώβριος 2020

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η Διπλωματική Εργασία με θέμα

«Μεθοδολογίες Ταξινόμησης Κακόβουλου Λογισμικού»

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

Βασιλείου Τσουβάλα

(Α.Μ.: 227950)

παρουσιάτηκε δημόσια και εξετάστηκε στο τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις

..../..../....

Επιβλέπων	Διευθυντής Τομέα
Δημήτριος Σερπάνος	Βασίλης Παλιουράς
Καθηγητής	*Καθηγητής*

Αριθμός Διπλωματικής Εργασίας: 227950/2020

Θέμα: «Μεθοδολογίες Ταξινόμησης Κακόβουλου Λογισμικού»

Φοιτητής

Βασίλειος Τσουβάλας

Επιβλέπων

Δημήτριος Σερπάνος

Περίληψη

Η ανίχνευση κακόβουλου λογισμικού αναφέρεται στη διαδικασία κατά την οποία, χρησιμοποιώντας δίαφορες μεθόδους και τεχνικές ανάλυσης λογισμικού, έχουμε τη δυνατότητα να κατηγοριοποιήσουμε ένα πρόγραμμα ως κακόβουλο ή καλόβουλο. Στην παρούσα εργασία, παρουσιάζουμε μία λύση, κατά την οποία εξάγουμε πληροφορίες από ένα εκτελέσιμο δείγμα, μοντελοποιούμε αυτές τις πληροφορίες σε γράφους και εφαρμόζοντας τεχνικές μηχανικής μάθησης αποφαινόμαστε για τη φύση του εκτελέσιμου (καλόβουλο ή κακόβουλο). Ξεκινώντας με ένα σύνολο δεδομενων από καλόβουλα και κακόβουλα Windows εκτελέσιμα δείγματα, εξάγουμε, μέσω στατικής ανάλυσης, τις κλήσεις που πραγματοποιεί το κάθε εκτελέσιμο στο Windows API, μοντελοποιούμε ένα Γράφο API Κλήσεων (API Call Graph) και εν συνεχεία έναν Αφηρημένο Γράφο API Κλήσεων (Abstract API Call Graph). Μέσω ενός Kernel Τυχαίας Διαδρομής Γράφων (Random Walk Graph Kernel), διανυσματικοποιούμε τη σύγκριση των Αφηρημένων Γράφων API Κλήσεων, ώστε να πραγματοποιήσουμε τη δυαδική ταξινόμηση με τη χρήση των Support Vector Machines (SVM's). Ακολουθώντας την παραπάνω διαδικασία, πετυχαίνουμε επίπεδα ακρίβειας μέχρι 98.25% χρησιμοποιώντας μικρότερο dataset από εκείνο που χρησιμοποιείται σε παρόμοιες προσπάθειες, αλλα και μειώνοντας τις απαιτήσεις σε χρόνο και υπολογιστική ισχύ.

Λέξεις Κλειδιά: Κακόβουλο Λογισμικό, Μηχανική Μάθηση, SVM, Γράφοι, Graph Kernel

Στούς γονείς μου, Κωνσταντίνο και Παναγιώτα

Η γλώσσα εκπόνησης της παρούσας εργασίας είναι η αγγλική. Στις τελευταίες σελίδες, παρέχεται συμπυκνωμένη έκδοση της εργασίας στα ελληνικά.

The present study is written in english. In the last pages, a compacted version of the study is provided in greek.



UNIVERSITY OF
PATRAS
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

Polytechnic School

Department of Electrical and Computer Engineering

Thesis

Malware Classification Methodologies

Vasileios Tsouvalas

Supervised by Dr.Dimitrios Serpanos

Patras, October 2020

Contents

1 Introduction **4**

2 The Malware Issue **5**

3 State of the Art **6**

3.1 Static Analysis 6

3.2 Dynamic Analysis 6

3.3 Machine Learning 6

4 Proposed Solution **7**

5 Tools and Dataset **9**

5.1 Tools 9

5.1.1 Disassembly 9

5.1.2 API Call Graph 10

5.1.3 Classification 11

5.1.4 Server 11

5.2 Dataset 11

5.2.1 Benign 12

5.2.2 Malware 12

6 Implementation **13**

6.1 Ghidra Analysis 13

6.1.1 Codeblocks 14

6.1.2 API Calls 14

6.2 Resources Extraction 16

6.3 Control Flow Graph 16

6.4 API Call Graph 16

6.5 Abstract API Call Graph 17
6.6 Graph Comparison: Random Walk Graph Kernels 18
6.6.1 Random Walk 18
6.6.2 Graph Kernels 18
6.6.3 Direct Product Graphs 19
6.6.4 Random Walk Graph Kernel (RWGK) 20
6.6.5 Generalized Definition of the Random Walk Graph Kernel for Labeled Graphs 21
6.6.6 Present Implementation of the RWGK 23
6.6.7 Kernel Normalization 23
6.7 Support Vector Machines 24
6.7.1 Training and Testing Splits, Number of Repetitions and Input 24
6.7.2 Training 25
6.7.3 Testing 26

7 Experiments **27**
7.1 Sample Size and API Calls 27
7.2 Sparsity Metrics 28
7.3 API Call Frequency 32
7.3.1 Distinct API Calls 32
7.3.2 Common API Calls 33

8 Classification Results **37**
8.1 Kernel Measurements 37
8.1.1 Benign-Benign 37
8.1.2 Malware-Malware 38
8.1.3 Benign-Malware 41

8.2 SVM Results 42
8.2.1 Accuracy 42
8.2.2 ROC Curve-AUC 43
8.2.3 Precision and Recal 45

9 Conclusion **47**

List of Figures **48**

References **50**

1 Introduction

Computer security or cybersecurity decribes the field that encompasses all the efforts pertaining to the protection of computers and networks. Given the magnitude of the issue and the potential risks, the field of computer security is an area where the scientific community, the industry and govermental agencies very often have convergent goals. One of the most significant and prolific challenges in computer security is malware and its detection. In the present thesis, we tackle the problem of malware detection, in the sense that we construct a pipeline in order to classify software as malicious or benign. The algorithm constructed features a graph-based approach to the malware detection problem, which has been attempted in similar efforts[1], and employs Machine Learning techniques for the classification.

2 The Malware Issue

It has been estimated that the total value at risk globally due to cyberattacks may reach up to 5.2 trillion USD until 2023[2]. Cyberattacks comprise of hacking, malware, phishing techniques and social engineering in order to gain unauthorized access to a system. While hacking, phishing and social engineering consist of actively working in order to bypass a system to reach a malicious goal, malware is a more passive approach to the same objective. Malware is defined as a software that has been designed to cause damage or a program that performs an action it should not, with either malicious intent or not. Malware-related cybercrime and breaches account for 28% of the all cyberattacks[3], thus making malware detection an essential tool in the front against cybercrime.

3 State of the Art

Malware analysis is divided into two categories; static and dynamic analysis. Static analysis involves examining the malware without actually running it, whilst in dynamic analysis, the malware is being run in a secure environment (usually a virtual machine) in order to gather information surrounding its execution. Increasingly, over the last decade, Machine Learning and other data mining techniques have been utilized in the malware detection schemes [4].

3.1 Static Analysis

The most common methods employed for static analysis of malware involve signature-based detection, where a predefined database of unique digital malware signatures allows the system to recognize threats, and code analysis, where the malware is reverse-engineered, using a disassembler or a decompiler, in order to examine the code and conclude on the nature of the program.

3.2 Dynamic Analysis

Dynamic analysis, on the other hand, allows for more information to be extracted, since the runtime behavior of the malware can be explored. As such, dynamic analysis offers a superior alternative to static analysis, since the malware is being observed during execution and more aspects of its nature may be evaluated. However, it is also the more costly and power demanding choice.

3.3 Machine Learning

Machine Learning approaches have been implemented in malware detection, at the level of classification, providing accuracy factors ranging from 80% to 99%, depending on the specific techniques and datasets used. Support Vector Machines account for 29% of the learning schemes employed[4].

4 Proposed Solution

Following the state-of-the-art approaches to the solution of the malware detection problem[5], as presented in the relative bibliography, our study focuses on proposing a graph-based detection scheme combining static analysis of executables[6][7], with well known machine learning tools such as Support Vector Machines (SVM).

We begin with a dataset of acknowledged benign and malicious MS Windows executable files, which we analyze one by one. For each executable, the analysis involves the disassembly of the executable file, which allows for the extraction of the API Calls that the executable makes to the Windows Operating System (OS). The API Calls as well as information regarding their connectivity, are consequently used in modeling the API Call Graph of the executable. After a graph-theoretic manipulation of the API Call Graph, we obtain an Abstract API Call Graph, and with the use of an appropriately defined graph-kernel[8], for the purpose of graph comparison, we arrive at the detection step of the algorithm. It is important to remark here, that in our study, we treat the malware detection problem as a binary classification one, meaning that, given an executable of unknown "intent", the goal is to classify it as being "malicious" or" benign". To this end, a Support Vector Machine scheme is implemented in order to achieve the classification goal.

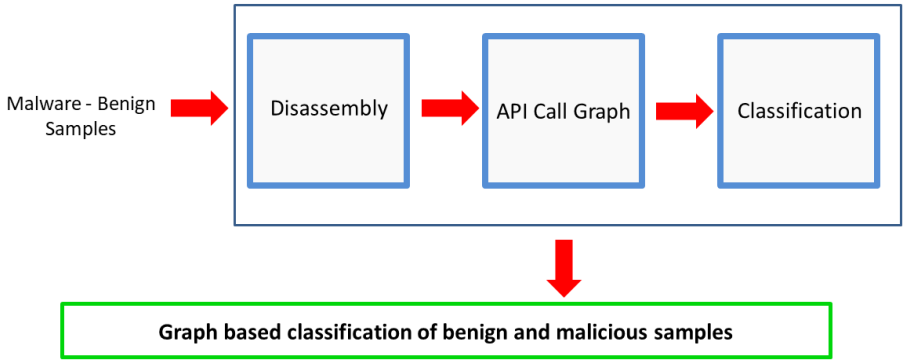


Figure 1: Bird's eye view of the proposed solution pipeline

5 Tools and Dataset

In the following sections, we will elaborate on the tools and the data used in the present study, and we will explicitly discuss each part of the pipeline of the proposed solution.

5.1 Tools

The toolbox that has been assembled for the graph-based malware detection application of the proposed solution consists of tools aimed at tackling the three main parts of the pipeline; Disassembly of the executables, API Call Graph modeling and handling, and Classification. We discuss the tools employed in each of those pipeline compartments below.

5.1.1 Disassembly

In this first part of the pipeline, we wish to diassemble the executable sample so as to perform a code analysis and extract certain crucial to the algorithm information. The tool that is used for the disassembly procedure is the open source program Ghidra[9].

Ghidra

Ghidra is an open source software analysis tool developped by the National Security Agency released on April 4, 2019. Ghidra is a powerful code analysis tool enabling a wide range of functionalities such as disassembly, and decompilation, as well as allowing for scripting and graph representation of data referring to the code analysis. One of the most important capabilities of Ghidra, is the fact that its API can be used to develop custom code for performing tasks tailored to the specific user's requirements. It is this particular aspect that allows for coding scripts in order to extract the data and information required in this

study, and more specifically, the API Calls that the executable makes and their intraconnectivity.

Programming Language

The code that has been developped in this part of the pipeline engages in the extraction of specific resources from Ghidra, after the disassembly process. Ghidra allows for scripting on top of its API, and thus, we are able to write code in Java, that allows us to manipulate the results of the analysis and extract the important information. We remark that the use of external libraries has been employed in Java coding, to account for the transfer of data and the connection to the Ghidra API.

5.1.2 API Call Graph

Moving on to the second component of the pipeline, we discuss the tools used in the resources extraction posterior to the disassembly, and the handling of said resources in the modeling and manipulation so as to reach the API Call Graph, as well as other necessary features. We discuss below, the programming languages employed and the theoretical elements that significant parts of the study are based upon.

Programming Languages

The programming language utilized in this compartment is Python, and its use takes place in several parts of the algorithm. Code has been written in Python for the handling of this data, the modeling of it into graphs, as well as representation of results. We use Python libraries to accomodate for several goals such as intraconnectivity of the code, handling of information, as well as results representation.

Graph Theory

After the extraction of resources from the executable sample, we produce graph representations of the different elements extracted. We manipulate and model these representations using Graph theory principles. For example, in order to reach an Abstract API Call Graph from an API Call Graph, we employ Graph Theory practices such as bipartite graph abstraction[10] and network component projection[11].

5.1.3 Classification

In the classification module of the proposed solution, we use Support Vector Machines (SVM's), treating the malware detection problem as a binary classification one. SVM's allow us to classify the samples as malicious or benign based on a specific aspect of their nature; in our case the API Call Graph. We remark that the code written for the SVM's is in Python and the corresponding classification libraries have been included.

5.1.4 Server

Certain modules of the pipeline are very demanding in computational power, and require a high performance CPU and RAM for their execution. In our case, we use a server, which allows for the more burdensome procedures of code analysis and resources extraction to be performed on a more powerful machine. The given server contains an 8-core CPU and has 16GB of RAM.

5.2 Dataset

In this section, we discuss the executable samples that were gathered for this study, which make up our dataset. The dataset contains labeled executables that are acknowledged and recognized to be either malicious or benign. We

remark that one of the most fundemental issues facing any attempt to tackle the malware detection problem is the lack of freely available and agreed-upon datasets. Great efforts have been made in this study to assemble a sufficiently large dataset for the training of the SVM classifier.

5.2.1 Benign

The benign samples collected are mainly installation and support files from an array of trusted sources such as Windows[12], Git[13], Cygwin[14], Codeblocks[15] and others. The amount gathered is 567 executables, ranging in size from several hundred kB to several MB.

5.2.2 Malware

The gathering of malware samples is an easier task , since there are numerous dedicated websites such as VirusShare[16] and VirusTotal[17], which provide these malware samples for academic and scientific purposes. We collected 997 malicious executables from VirusShare, again ranging for several hundred kB to several MB.

6 Implementation

In this section, we discuss in detail every step of the pipeline and explicitly elaborate on the inner mechanisms of the proposed solution.

6.1 Ghidra Analysis

In this first part of the algorithm, we use the malicious and benign samples as input to the Ghidra platform. Ghidra performs a code analysis and disassembly on all of the dataset samples recursively. A Ghidra analysis establishes that the sample can be correctly disassembled and its code can be represented in an assembler language. We note that out of the 567 benign samples, all are correctly analyzed and disassembled, whereas only 827 of the total 997 malware samples are correctly analyzed. This may occur due to corruption, or other flaws in the integrity of the problematic files.
We remark that in all cases, the analysis is static and the executable is never run. Following this, any reference made to actions following its execution refers to the possible execution actions based on information that has been statically gathered by Ghidra.

After the Ghidra analysis and disassembly, the executables take the form of assembly code which provides us with information on the data that the executable needs from the platform it runs on (Windows OS in our case) and other aspects such as the registers that it uses, the functions that it calls, etc.. In each of the executables' assembly code we notice the existence of codeblocks. Codeblocks are an essential part of this study and their functionality is at the center of it.

6.1.1 Codeblocks

Codeblocks are bundles of code which, after the Ghidra analysis, are revealed to perform one specific action each. In that sense, codeblocks could be characterized as internal functions. Codeblocks are also internal nodes of the executable and their directed connections represent possible execution paths. In this sense, they can also be characterized as Control Points of the system.

6.1.2 API Calls

A Codeblock may lead to another Codeblock or it may lead to an API Call, which is a function call to the Windows API. All API Calls are provided by the Windows API[18], which is the native platform for Windows applications and the executable uses them to ask the OS for certain resources or to ask for certain actions to be performed; i.e. user input and messaging, data access and storage, system services, etc. Each API Call has a unique name and a unique ordinal number. In our study we use the name of an API Call as its label. It is important to mention that API Calls cannot lead to other API Calls. The process in which their call takes place is the following:

1. A Codeblock reaches an API Call
2. The program "branches" to execute the API Call
3. After the API Call has finished its execution, the program returns to the original Codeblock and resumes its path

This means that only Codeblocks are connected and through their connection, can API Calls be considered to take place in sequence.

Concluding on the analysis and disassembly, we have for each sample:

1. Code in assembly
2. Codeblocks - Control Points
3. All the possible execution paths
4. The API Calls it makes

6.2 Resources Extraction

After the analysis, we extract the following information from the disassembled executable:

- Codeblocks names and addresses
- Incoming and outgoing Codeblocks or API Calls from each Codeblock
- Addresses of those incoming and outgoing Codeblocks or API Calls
- The size of the executable

Using the above mentioned information, we are able to model the Control Flow Graph.

6.3 Control Flow Graph

The Control Flow Graph (CFG) is simply the directed graph whose nodes are the executable's Codeblocks and its edges are the connections with other Codeblocks (based on the incoming and outgoing Codeblocks information). It can be represented as a tuple $G = (N, E)$, where N is a finite set of nodes and E is a finite set of edges. Following this, (n_1, n_2) represents that the Codeblock n_1 can lead to Codeblock n_2.

6.4 API Call Graph

Going from a CFG to an API Call Graph, we have to note that each Codeblock may lead to more than one API Calls, more Codeblocks may lead to the same API Calls and a CFG may contain loops.

An API Call Graph is a directed graph that is extracted from the CFG of the sample, by replacing each node with the API Calls that the specific Codeblock

leads to. Following the example from before, let us assume that Codeblock n_1 leads to API Call f_1 and Codeblock n_2 leads to API Call f_2. In this sense, $((n_1, f_1), (n_2, f_2))$ represents the equivalency between Codeblock and API Call connection. For the API Call Graph, we have that $G_{APICall} = (F_{API}, E_{API})$, where F_{API} is a finite set of nodes, which represents the API Calls reachable from the source (initial) Codeblock, and E_{API} is a finite set of edges, which represents the API Calls reachable from the destination Codeblock. It becomes apparent that the size of the API Call Graph is much greater than the size of the Control Flow Graph.

6.5 Abstract API Call Graph

We wish to compare graphs between them so as to conlude on their classification, but it becomes apparent that the size of the API Call Graph poses a problem. As the size of the API Call Graph is very big, we perform an abstraction, in order to make it more manageable computationally. We address the API Call Graph as a series of bipartite graphs and perform an abstraction on those to reach an Abstract API Call Graph. The Abstract API Call Graph merges all the nodes referring to the same API Call in one node. Therefore, there are as many nodes in the Abstract API Call Graph, as there are distinct API Calls in the executable. The new merged nodes are connected to each other following the connections of every instance of the API Call in the previous API Call Graph. If (f_1, f_2) , (f_1, f_3) and (f_3, f_1) described the API Call Graph, then the Abstract API Call Graph will be represented by (f_1, f_2) and (f_1, f_3). Following this abstraction, we lose the directionality of the graph model of API Calls, but we gain in computational cost, which allows us to perform comparisons among graphs and eventually be able to classify them as belonging to malicious or benign executables. By performing this abstraction, we reduce considerably the

size of the graph in question.

6.6 Graph Comparison: Random Walk Graph Kernels

6.6.1 Random Walk

Random Walk (RW) is a well known and widely used algorithm that provides random paths on a graph. Its basic function involves beginning at a node and choosing at random a neighboring node to visit. This procedure in succesion allows for a traced path on a graph that has been acquired in a random fashion.

6.6.2 Graph Kernels

A graph kernel is a kernel function that calculates the inner product of two graphs. It provides a measure of similarity between the two graphs[19][20] and is especially useful in kernelized learning and classification algorithms such as Support Vector Machines, since it allows us to operate directly on the graphs. We use a special case of graph kernels, namely Random Walk Graph Kernels, in order to compare all possible pairs of the Abstract API Call Graphs so as to use Support Vector Machines to classify them as hailing from malicious or benign executables.

The idea behind Random Walk Graph Kernels is extensively discussed in the relative bibliography[21]. A Random Walk Graph Kernel between two graphs performs random walks on both of the graphs and counts the number of matching paths. Performing a random walk on the direct product graph of a pair of graphs is equivalent to performing random walk simultaneously on that pair of graphs[22]. This is being achieved through the use of direct product graphs, which we define as following:

6.6.3 Direct Product Graphs

Given two graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$, their direct product graph $G_\times = (N_\times, E_\times)$ is defined as a graph over pairs of nodes from G_1 and G_2, where two nodes of $G_\times$ are neighboring if and only if the corresponding nodes in G_1 and G_2 are both neighbors. Let A_1 and A_2 be the adjacency matrices of G_1 and G_2 respectively. Following this, the adjacency matrix of $G_\times$ is

$$A_\times = A_1 \times A_2 \tag{1}$$

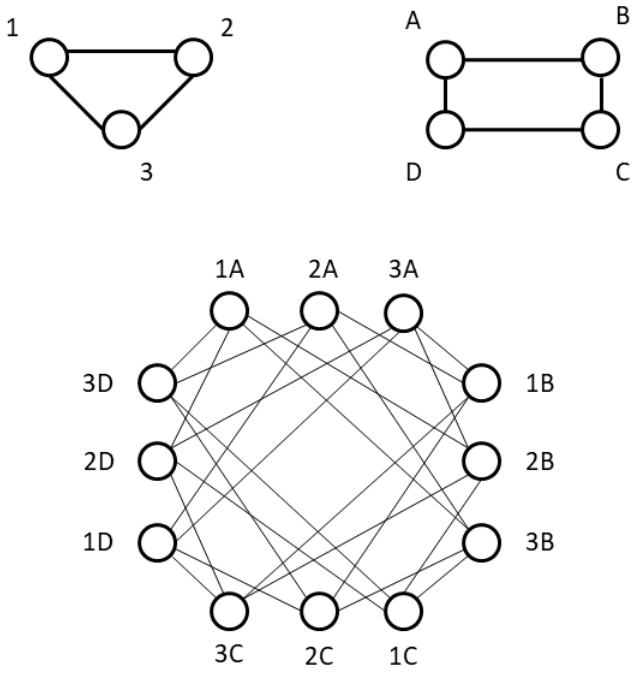


Figure 2: Graph G_1 (on the top left), Graph G_2 (on the top right), Direct product graph $G_\times$ of G_1, and G_2 (bottom)

6.6.4 Random Walk Graph Kernel (RWGK)

Following the relative bibliography, let us define the initial distribution of probabilities over the nodes in G_1 and G_2, as p_1 and p_2 respectively, and the stopping probabilities (meaning the probability that a random walk ends at the given node), as q_1 and q_2 [1][21][23]. Considering these probabilities, we calculate the equivalent probabilities of the direct product graph:

$$p_\times = p_1 \otimes p_2 \quad and \quad q_\times = q_1 \otimes q_2 \tag{2}$$

, where $\otimes$ denotes the Kronecker product.

We define the Random Walk Graph Kernel as

$$\kappa(G_1, G_2) = \sum_{k=0}^{T} \mu(k) q_\times^\intercal A_\times^k p_\times \tag{3}$$

where:

- $A_\times$ is the adjacency matrix of the direct product graph as defined in in (1) and subsequently A^k represents the probability of simultaneous length k random walks on G_1 and G_2.
- $p_\times$ and $q_\times$ represent the inital probability distribution and stopping probability of the direct product graph, as mentionned in (2)
- T is the maximum length of the random walks
- $\mu(k) = \lambda^k \in [0, 1]$ is a coefficient that controls the importance of length in random walks which we use to ensure that the sum converges and the kernel value is well defined

The binary classification task at hand requires comparing each graph over every other. To this end, we recursively choose all possible pairs of graphs in order

to compute the kernel value of each pair. The kernel value is a measure of the similarity of the two graphs and by gathering all the kernel values, we manage to turn our problem from a graph (i.e. non-linear) classification one to a linear one, for which a linear classifier may be used. More specifically, given the kernel values of all possible pairs, we can directly use these kernel values as input to the SVM classifier. The principle of kernelization of the data, so as to transform a non-linear problem to a linear one, is also referred to as the kernel trick[24][25].

6.6.5 Generalized Definition of the Random Walk Graph Kernel for Labeled Graphs

Every continuous, symmetric, positive definite kernel $\kappa\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ has a corresponding Hilbert space $\mathcal{H}$, called the Reproducing Kernel Hilbert Space (RKHS)[21].
Providing a different representation of the RKHS involves using feature maps, where a feature map is a mapping function such that $\phi\colon \mathcal{X} \to \mathcal{H}$

A feature map also defines a kernel as:

$$\kappa(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{H}} \tag{4}$$

We note that every positive definite function that correponds to a RKHS has infinite feature maps associated with it.

Extending the feature maps to matrices, we have

$$\Phi\colon \mathcal{X}^{n \times m} \to \mathcal{H}^{n \times m} and |\Phi(A)|_{ij} \colon = \phi(A_{ij}) \tag{5}$$

The generalized definition of the RWGK refers to including a weight matrix $W_\times$

$$\kappa(G_1, G_2) = \sum_{k=0}^{T} \mu(k) q_\times^\intercal W_\times^k p_\times \tag{6}$$

where $W_\times$ is a weight matrix for which we have

$$W_\times = \Phi(X_1) \otimes \Phi(X_2) \tag{7}$$

Given the definition of $\Phi(X_1)$ and $\Phi(X_2)$, and understanding that in the present study the nodes of the graphs are labeled (the node labels of the Abstract API Call Graph are the names of distinct API Calls), we are able to deduce that $W_\times = A_\times$. This is also recognized in the bibliography[21][23][26]. Following this and utilizing the Sylvester Equation Methods, the kernel definition may be given by

$$\kappa(G_1, G_2) = q_\times^\intercal (\mathbf{I} - \lambda A_\times)^{-1} p_\times \tag{8}$$

Given the solution of the Sylvester equation associated with the kernel definition above, we are able to reach $\mathcal{O}(n^2)$ time, whereas the straight computation of inverting $(\mathbf{I} - \lambda A_\times)$ would take $\mathcal{O}(n^6)$.

6.6.6 Present Implementation of the RWGK

We remark that the RWGK is an adjustable quantity and allows us to regulate several parameters in order to fit each classification scheme. In the present implementation we can exploit two aspects of the graphs we wish to classify:

- The initial probability distributions and stopping probabilities are neglected[19] since we only handle the Random Walk algorithm by regulating the importance of its depth, namely λ. This allows us for more efficient computing and faster Random Walks.

- When computing the adjacency matrix of a pair of graphs $A_{\times}$, we accomodate for all the nodes of both graphs. However, given that the node labels are API Calls (unquely named) and that the Abstract API Call Graph only contains each API Call once, we can eliminate all the nodes that are different between the two graphs. Only common API Calls matter in the graph comparison of a pair of graphs since any different API Call nodes will augment the size of $A_{\times}$ by inserting zeros. This allows us to reduce the computation time, since we avoid an enormous amount of element-wise operations that lead to zero and give us no information, thus enabling us to utilize the resources (Server RAM) more efficiently.

6.6.7 Kernel Normalization

Even without a classification scheme, the kernel values allow for the extraction of conclusions, since they provide afterall a measure of similarity between graphs[19][20]. In this sense, we normalize the kernel values acquired from the aforementioned procedure so as to examine the results at this level as well. The normalization of the kernel values[20][27][28][29] allows for the correct examination of similarity or lack thereof since it ensures that a graph always matches

itself with a normalized kernel value of 1 and all the other graphs with a value $\in (0, 1)$. The normalized kernel value is given by

$$\hat{\kappa}(G_1, G_2) = \frac{\kappa(G_1, G_2)}{max(\kappa(G_1, G_1), \kappa(G_2, G_2))} \tag{9}$$

Details on the Implementation It is important to remark that after an extensive search in the existing Python libraries that allow for graph kernel calculation[30][31], none where found to match the criteria placed above, and thus the implementation presented in this study is a Random Walk Graph Kernel for labeled graphs that was specifically written and compiled for the problem at hand.

6.7 Support Vector Machines

Having computed the kernel values of all possible pairs of graphs, we can use them in a Support Vector Machine (SVM) classification scheme. Given that SVM's are kernelized learning algorithms, the calculation of the kernel values have transformed a graph classification problem into a linear problem. The objective of the SVM is to classify the linearized acknowledged data in a way that allows for accurate predictions on new data. In this section, we discuss certain basic principles of SVM's and describe the implementation details. Given the fact that we have already linearized the problem via the kernel trick, we proceed with the description of a linear SVM.

6.7.1 Training and Testing Splits, Number of Repetitions and Input

We train and test the SVM classifier by using 9 different training to testing set splits, meaning that we begin with a certain number of kernel values (equal number of benign and malware originating kernel values) and split these kernel values in a random fashion beginning with a 10%-90% train-to-test split, and

with 10% intervals, arrive to a 90%-10% train-to-test split. It is apparent that the two sets - training and testing- are complementary between them to the total number of kernel values.

The whole procedure is repeated 100 times, so as to eliminate the flaws of the randomized train-to-test split and also allow for a larger number of predictions in order to minimize the effect of random spikes or drops in measurements.

The kernel values are split in groups containing an equal number of malware and benign originating kernel values. Those groups contain 50, 100, 150, 200, 250, and 300 benign-benign and malware-malware kernel values.

6.7.2 Training

Given a set of training data of n samples $(x_1, y_1)...(x_n, y_n)$, x_i represents a feature and y_i is its class label and can take values in $\{-1, 1\}$. The training of the SVM describes the procedure during which the algorithm finds the optimal classifier that seperates the labeled data, while maximizing the margin between them. This optimal classifier is a hyperplane and letting w be the normal to that hyperplane, we arrive at the SVM decision function

$$h(x) = w^{\intercal}x + b \quad , where \;\; x = \sum_{i_1}^{n} \alpha_i x_i y_i \tag{10}$$

In the hyperplane function definition above, b represents the offset of the hyperplane from the origin and α_i represents the SVM training parameters. Having a linearized problem, we can substitute the kernel values for x_i. The kernel values that we use are two different sets; the ones computed between benign executable originating graphs among themselves and malware originating graphs again among themselves. The class values of y_i take 1 as a value if the corre-

sponding x_i is a benign-benign graph kernel value, and -1 if the corresponding x_i is a malware-malware graph kernel value.

6.7.3 Testing

Having trained the classifier, we proceed to the testing procedure of the SVM, during which the classifier takes as input never before seen kernel values and predicts on their class based on the hyperplane that was developped for the training set before. We measure the accuracy, precision, recall and ROC Area Under Curve of the SVM and accordingly conclude on the prediction it makes on the testing set.

7 Experiments

In this part, measurements performed on the dataset after the analysis procedure are presented. These measurements indicate the differences in essential characteristics of the malware and the benign executables and emphasize divergent qualities of the two types of samples. In the effort to classify executables, accumulating knowledge on irregularities, even smaller ones, allows us to enhance the aspects of the already established criteria. Managing to accumulate enough discrepancies, from different measurements, provides a more in depth comprehension of the importance and gravity of certain features of the malware detection problem. More specifically, we examine the size of the samples compared to the number of distinct API Calls in each sample, we look at the sparsity of the Abstract API Call Graphs, and we assess the differences in appearance frequency of specific API Calls.

7.1 Sample Size and API Calls

The size of both the benign executables and the malwares range from several KB to several MB. In Figure 3, we compare the size of each executable sample over the number of API Calls it makes. Examining Figure 3, we quickly notice the 3 distinct spikes produced by the malware at around 0, 180, and 520 API Calls, whereas the benign executables do not show any irregular behavior in the given plot. Analyzing the irregularities by the malware, it becomes apparent that some of them do not follow the reasonable assumption that a larger number of API Calls is related with a bigger size. Certain malware are very big in size and have the same amount of API Calls as other much smaller malware. The reason behind this phenomenon, is related to the fact that most malware are newer, updated versions of older ones. Considering this, along with the fact that obfuscating techniques and multiple additions of "dead" code are often employed

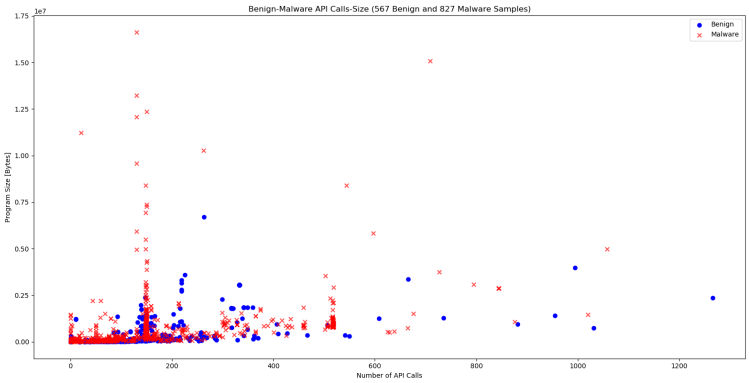


Figure 3: Benign-Malware API Calls - Executable Size

in the malware development, it is understandable why such large sized malwares present such few API Calls.

7.2 Sparsity Metrics

Measurements were performed at different stages of the pipeline. Two sparsity metrics of the Abstract API Call Graphs were computed:

Sparsity Metric 1:

$$SM1: = \frac{number\ of\ edges}{number\ of\ nodes} \tag{11}$$

Sparsity Metric 2:

$$SM2: = \frac{number\ of\ nodes}{number\ of\ edges} \tag{12}$$

The sparsity measurements gives us an understanding on how dense or sparse the Abstract API Call Graphs are, i.e. on how heavily intraconnected the API Calls are. The number of connections between the distinct API Call nodes of the Abstract API Call Graph informs us on the cyclomatic complexity of the executable. The cyclomatic complexity of an executable is a measure of

linearly independent execution paths through the code and is a software metric indicating the complexity of the program. In this manner, it can also be used to recognize maliciousness[32][33].

In Figures 4 and 5, we present the aforementionned Sparsity measurements compared to the number of the edges and the number of nodes of the Abstract API Call Graphs of the samples.

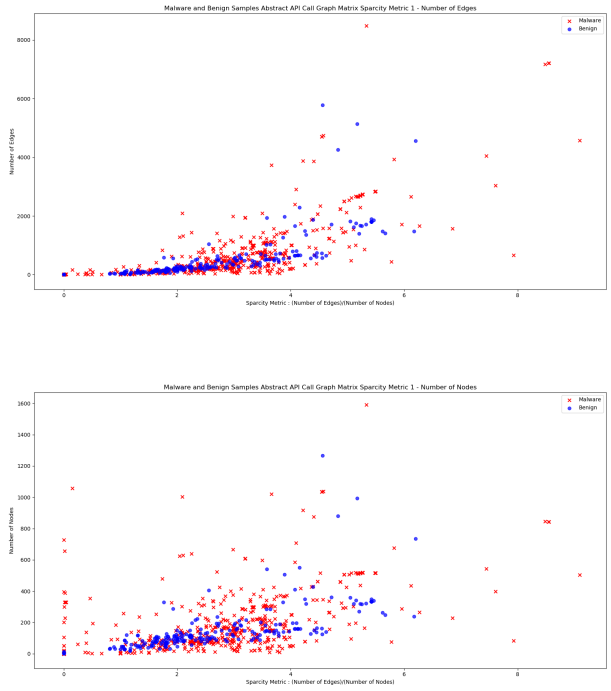


Figure 4: Benign-Malware Sparsity Metric 1 - Number of Edges (top), Number of Nodes (bottom)

On all four plots (Figures 4, 5), we remark that the malware exhibit a slightly divergent behavior compared to the benign executables. Although there seems to be an area in which they behave in the same manner, there are malware that have very large sparsity value, meaning that they are very lightly intraconnected

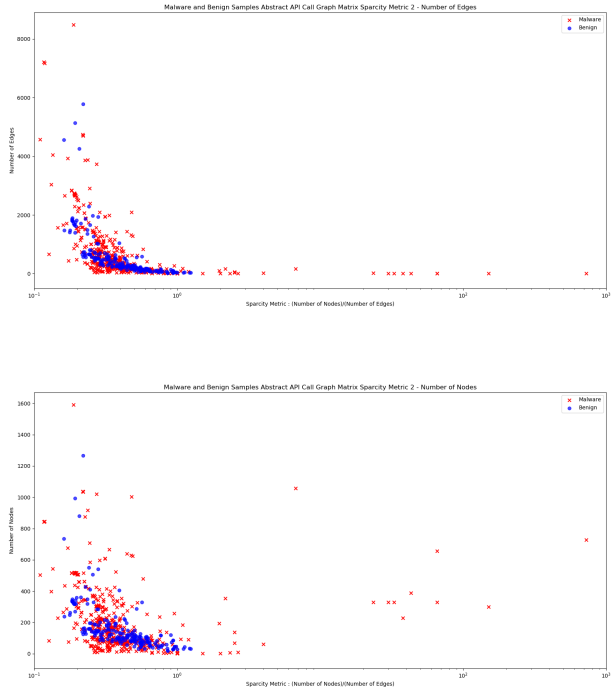


Figure 5: Benign-Malware Sparsity Metric 2 - Number of Edges (top), Number of Nodes (bottom)

in comparison with the benign executables which follow a more predictable evolution.

In Figure 6, we show both Sparsity Metrics compared with the size of the samples.

Observing the Sparsity Metrics to executable size plots, we notice that there are again certain spikes appearing in the size of the malware samples, which reveal an equivalence in sparsity of the Abstract API Call Graph for samples with vastly different sizes. This, along with the recognition that there is a similar behavior in malware upon looking at the comparison of size to API Calls (Figure 3), supports the argumentation that in malware, some are equivalent to others

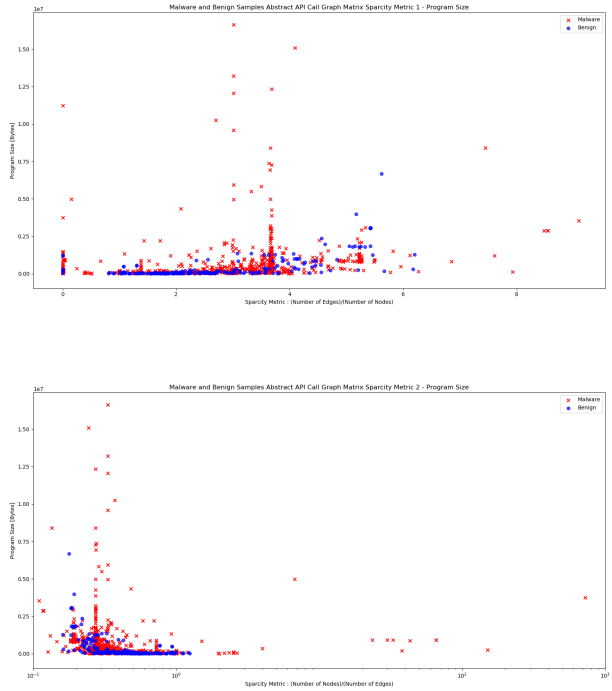


Figure 6: Benign-Malware Sparsity Metric 2 - Number of Edges (top), Number of Nodes (bottom)

in API Calls metrics, even though they are different in size. As mentioned before, one explanation of this could be that malware are usually developped upon older versions of similar malware, with the addition of code that does not change the behavior examined by the static analysis.

7.3 API Call Frequency

A measurement that presents interest in the context of malware detection is also the API Calls themselves and the frequency in which they appear in the dataset. We perform two types of measurements; API Call frequency for API Calls appearing only in Benign or Malware, and API Call frequency for API Calls appearing in both sets of data (common API Calls).

7.3.1 Distinct API Calls

In Figure 7, we plot the frequency of appearance of API Calls unique to benign executables and malware. Let the number of executable samples in a dataset be N and the number of appearances of a single distinct API Call, named i, in all the executable samples being n_i, then the frequency of appearance of the API Call i is given by

$$frequency_i = \frac{n_i}{N} \tag{13}$$

Examining the Figure 7 top plot, we notice that the frequency of API Calls appearing only in benign executables or only malware is generally very low. At the bottom plot of Figure 7, we display the same plot only allowing for API Calls of a frequency higher than 15%. In this plot, we have a clearer picture on the importance of API Call frequency, since we can extract certain API Calls that seem to be appearing at a distinctly high rate in benign executables and malware. We conclude on 4 API Calls for malware and 5 for benign executables. This metric allows us a hypothesis on the basis of the existence of these specific API Calls in relationship with their classification as malicious or benign. This constitutes that the extraction of such a feature may be significant.

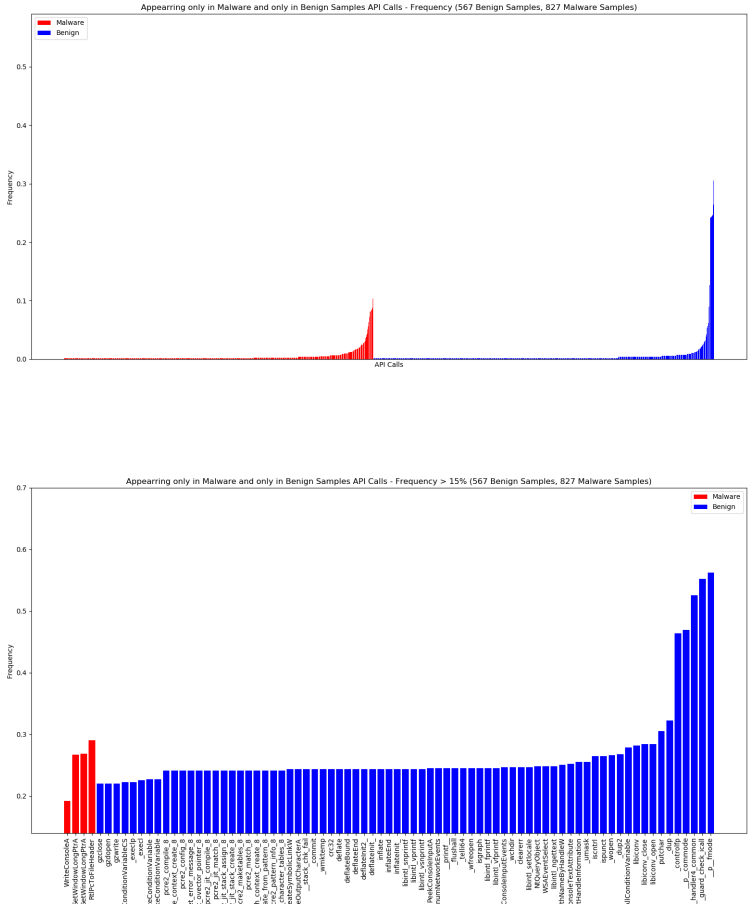


Figure 7: Appearing only in Malware or Benign - API Call Frequency all(top), frequency above 15% (bottom)

7.3.2 Common API Calls

In this section of the API Call frequency measurements, we examine the frequency of common to the benign executables and malware API Calls. In Table 1, we present the number of Common API calls in malware and benign samples in relationship to the frequency difference that they appear in the malware and the benign samples.

In Table 1, cutoff appearance frequency is the frequency difference above which

Cutoff Appearance Frequency	**Number of API Calls**
0	3031
1	1251
2	994
5	761
10	492
15	345
20	274
25	218
30	122
40	33
50	9
60	3
70	1
80	0

Table 1: Malware (827) and Benign(567) Samples Common API Calls based on Appearance Frequency Difference

there is a given number of common API Calls appearing; i.e. at a 10% cutoff appearance frequency, we have 492 distinct API Calls, among common API Calls between benign and malware, with an appearance frequency difference equal to or greater than 10%.

This measurement provides insight on two important aspects:

- For a big difference in appearance frequency among common API Calls, we can conclude that some API Calls are favored by benign and others by malware.
- The choice of frequency difference is important, since it establishes the tendency of API Calls to appear in benign or malware more. If the frequency of a common API Call is comparable to the frequencies of API Calls appearing exclusively in malware or benign executables (see Figure 7), then this API Call is inherently important and its existence can constitute a feature for classification.

In Figure 8, we plot the API Calls that appear in both malware and benign samples, whose appearance frequency difference in malware and benign is greater than 35%.

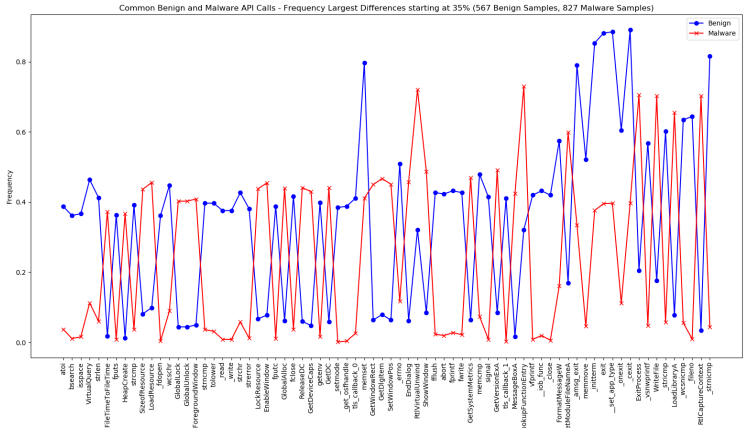


Figure 8: Common API Calls Appearance Frequency - Cutoff Appearance Frequency 35%

Based on the analysis of the common API Call frequency as a classification feature, we notice that there are certain common API Calls whose absolute fre-

quency is large enough and the frequency difference between malware and benign appearance is also substantial. These API Calls are memset, RtlLookupFunctionEntry,_initterm, exit, and __set_app_type.

Conclusion on API Call Frequency In this section, the effort has been focused on the extraction of possible classification features, using the information acquired on the API Calls appearance frequency in malware and benign executables. Although in isolation, these features may not suffice for the correct classification of the sample, they could play a significant role as weights of a classification scheme.

8 Classification Results

In this section, we provide the results that have been attainned by the malware detection pipeline of the proposed solution. We divide the results into preliminary ones -the kernel values for benign-benign, malware-malware, and benign-malware- and the final Support Vector Machine prediction results on accuracy, precision, recall and ROC AUC.

8.1 Kernel Measurements

As discussed before, the kernel is a measure of similarity between graphs[19][20]. The normalized kernels of different dataset sizes and combinations are being presented in the following sections.

8.1.1 Benign-Benign

We calculate the kernel values among benign executables for 50, 100, 150, 200, 250, and 300 benign samples. The resulting pairwise similarities are plotted in Figure 9, whereby each vertical sequence of dots represents the kernel of the given sample over all the other samples; we have on each line the kernel value (similarity) it shares with all the other. The kernel values are normalized so as to allow for correct cross-examination with the corresponding plots concerning the malware.

On each plot we also give a histogram of the distribution of the kernel values, in order to conclude on how similar the samples are overall (we refer to the histogram on the right side of the plots).

For the plots of Figure 9, we can easily notice that for all the different dataset sizes, the average similarity value is around 20%. This demonstrates experimentally that the datasets chosen are valid and do not contain benign samples that are overly similar to each other (as far as Abstract API Call Graphs go).

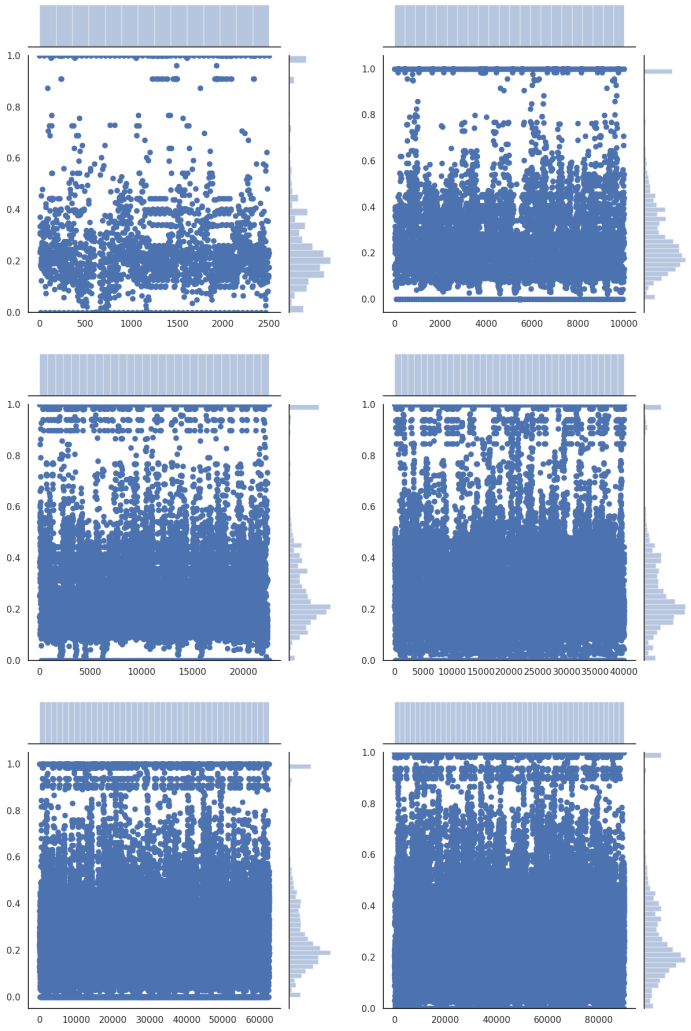


Figure 9: Benign-Benign Kernel Values for Dataset Size 50 (top-left), 100 (top-right), 150 (middle-left), 200 (middle-right), 250 (bottom-left), and 300 (bottom-right)

8.1.2 Malware-Malware

As calculated before for the benign samples, we compute the kernel values among malware for 50, 100, 150, 200, 250, and 300 malware each time and plot the resulting kernel values in Figure 10.

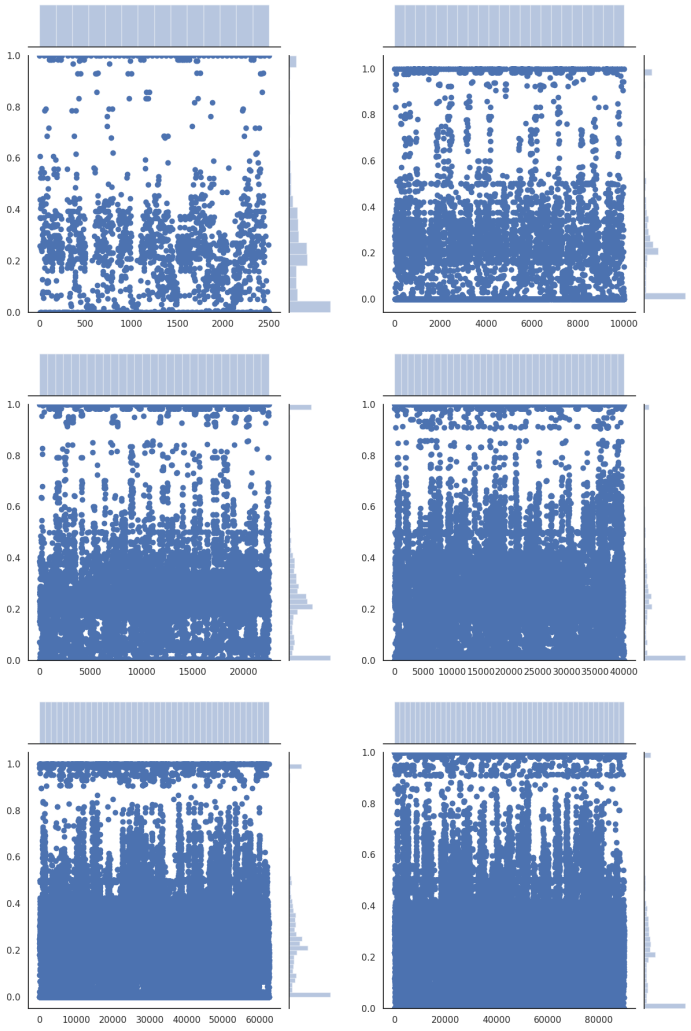


Figure 10: Malware-Malware Kernel Values for Dataset Size 50 (top-left), 100 (top-right), 150 (middle-left), 200 (middle-right), 250 (bottom-left), and 300 (bottom-right)

We notice that the similarity, in the case of the malware, centrers around 20% for the smaller datasets, but as the dataset size grows, the similarity tends to become smaller. We notice also a very large spike on the histogram around zero , which indicates that there is a large amount of samples that are very

different among them (in respect to Abstract API Call Graphs).

8.1.3 Benign-Malware

Following the same principle as before, but this time examining benign over malware Abstract API Call Graph kernel values, we plot the results in Figure 11. Note that this type of combination is not required for the SVM classification.

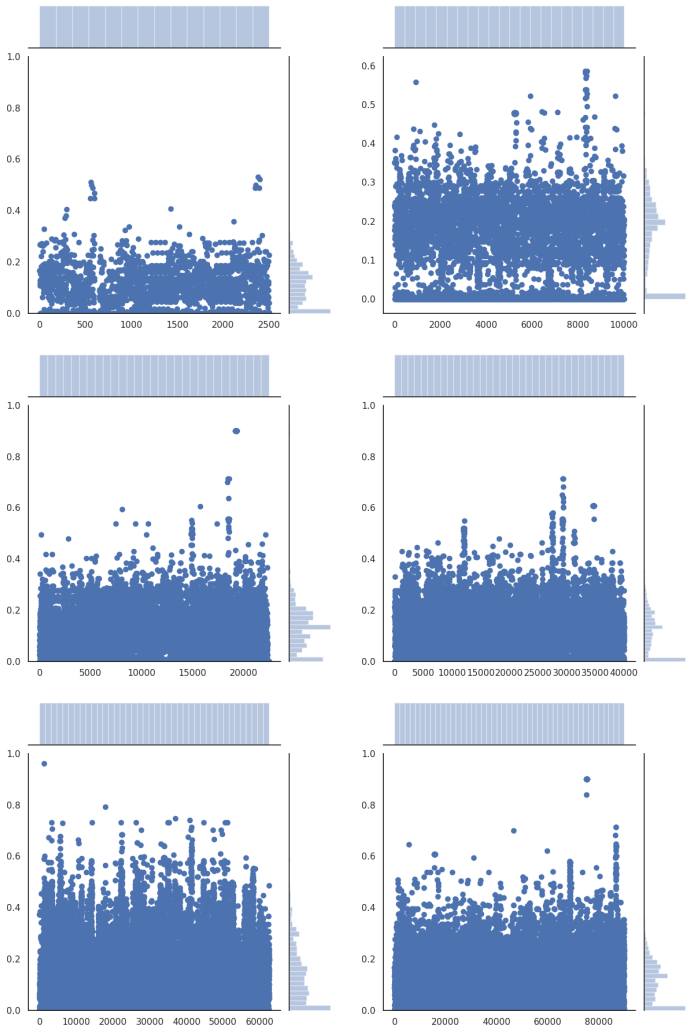


Figure 11: Benign-Malware Kernel Values for Dataset Size 50-50 (top-left), 100-100 (top-right), 150-150 (middle-left), 200-200 (middle-right), 250-250 (bottom-left), and 300-300 (bottom-right)

Nevertheless, it provides very significant information. Examining the benign over malware kernel value plots of Figure 11, we notice that the similarity centers around 10% and 15% and it becomes possible to define an upper bound with a reasonable degree of confidence around the 35% mark. This allows us to better understand the inner workings of a classification scheme, since with this cross-examination of benign to malware kernel values, we notice similarity bounds that can work as detection mechanisms.

8.2 SVM Results

After having build the SVM classifier for the different training to testing splits 100 times over, as described in section 6.7.1, we proceed with the graphical representation of the results. The best accuracy achieved by the SVM classifier is 98.25% with the dataset of 300 benign and 300 malware samples, meaning that the maximum error is at 1.75%, which is in line with the relative bibliography[1].

8.2.1 Accuracy

The accuracy of a classification scheme is defined as the number of correct predictions over the total number of predictions. In the case of a binary classifiers, such as the one implemented in the present study, we have the following

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{14}$$

where TP is the number of True Positives, TN is the number of True Negatives, FP is the number of False Positives, and FN is the number of False Negatives.

Examining the SVM Accuracy plot of Figure 12, we notice that, with the exception of the 50-50 dataset, achieved performance lies generally above 95%. We also notice that as the training size becomes a larger percentage of the dataset itself, the accuracy is higher, as is expected. The highest accuracy takes place for the 300-300 dataset (the largest one) and for the best possible training to testing split (90% of the dataset is used for training).

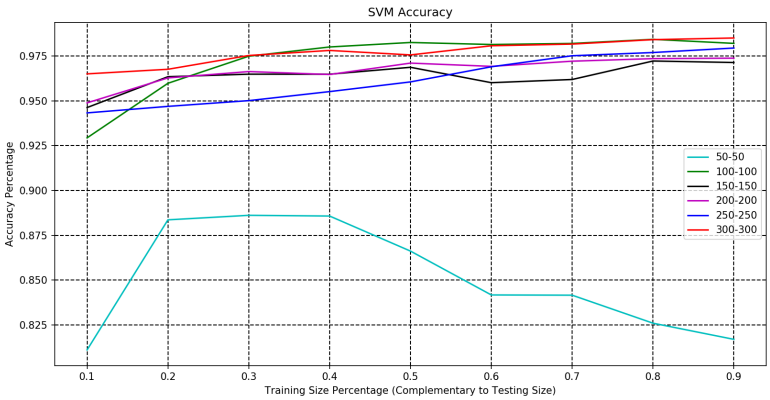


Figure 12: SVM Accuracy for the Different Dataset Sizes and Training to Testing Splits

8.2.2 ROC Curve-AUC

A Receiver Operating Characteristic (ROC) curve shows the performance of a classifier at all classification thresholds. We introduce the notions of True Positive Rate (TPR) and False Positive Rate (FPR) as

$$TPR = \frac{TP}{TP + FN} \tag{15}$$

$$FPR = \frac{FP}{FP + TN} \tag{16}$$

The ROC curve plots the TPR against the FPR at different classification

thresholds. The Area Under the ROC Curve (AUC), measures the area covered by the ROC curve. AUC measures the accumulated performance across all thresholds of classification. The ROC-AUC is a measurement on the quality of the classifier no matter what classification threshold is chosen. In Figure 13, we present the ROC-AUC for the SVM of the 6 aforementioned datasets (50,100,150,200, 250, 300).

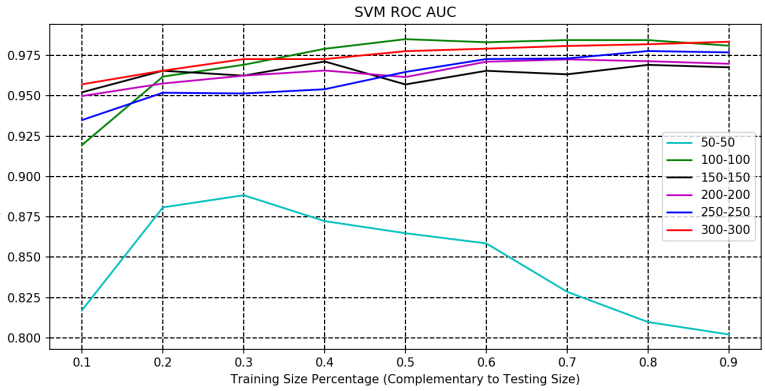


Figure 13: SVM ROC-AUC for the Different Dataset Sizes and Training to Testing Splits

In the Figure 13 plot, we notice that apart from the smaller dataset (50-50), the performance only gets better as the datasets increase in size and as the training to testing split becomes more favorable. We note that the ROC-AUC of the 50-50 dataset degrades in a quicker manner than the degradation of its accuracy in Figure 12, which shows the qualitative difference in the two measurements.

8.2.3 Precision and Recal

The precision, also known as positive predictive value, and recall, or sensitivity, are measurements usually examined together based on the proximity of their definitions. The precision of a classifier is defined as the ratio of positive identifications that were actually correct, whereas recall is defined as the ratio of actual positives that was correctly identified. We represent this, as

$$Precision = \frac{TP}{TP + FP} \tag{17}$$

$$Recall = \frac{TP}{TP + FN} \tag{18}$$

In Figure 14 and 15, we present the precision and recall measurements of the SVM classifier, which show, very high values of precision and recall as the training setup becomes more favorable (larger dataset and training split). It is important to note that a high precision relates to a low false positive rate and a high recall to a low false negative rate. Examining both plots, we conclude that the classifier is returning accurate results as well as returning a majority of positive results.

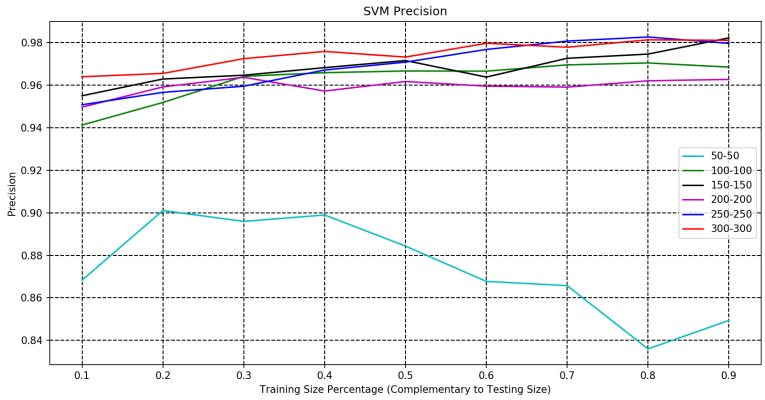


Figure 14: SVM Precision for the Different Dataset Sizes and Training to Testing Splits

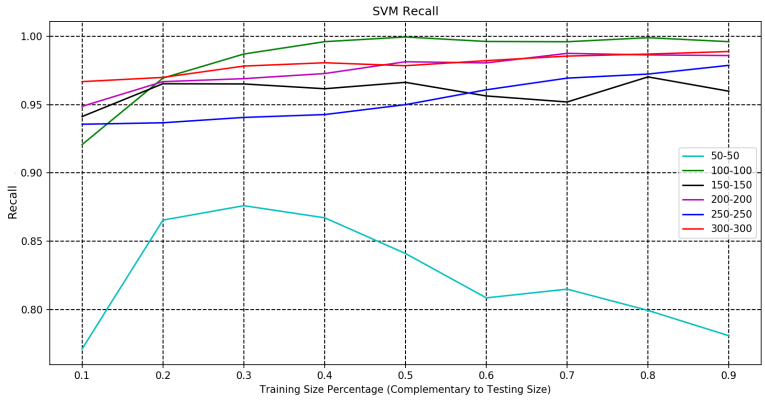


Figure 15: SVM Recall for the Different Dataset Sizes and Training to Testing Splits

9 Conclusion

The goal of the present study, was to tackle the malware detection problem in an efficient and accurate way. The proposed solution utilized static analysis techniques to disassemble benign and malware executable samples, study them, and extract important characteristics and features. Based on those, we developped ad hoc graph-based mechanisms to overcome computational limits and manage to fabricate subject specific solutions for the kernelization of non-linear data. The classification scheme employed to function as a malware detection configuration yielded very promising results. The importance of those results is only expanded if we consider that the dataset size used is an objectively small one (for such endeavors) and that the computational time is much smaller compared to relative studies. We remark that the results attainned in combination with the measurements collected, allow for solid hypotheses on the basis of the proposed solution compounded with extracted features for future research.

List of Figures

1 Bird's eye view of the proposed solution pipeline 8

2 Graph G_1 (on the top left), Graph G_2 (on the top right), Direct product graph $G_\times$ of G_1, and G_2 (bottom) 19

3 Benign-Malware API Calls - Executable Size 28

4 Benign-Malware Sparsity Metric 1 - Number of Edges (top), Number of Nodes (bottom) 29

5 Benign-Malware Sparsity Metric 2 - Number of Edges (top), Number of Nodes (bottom) 30

6 Benign-Malware Sparsity Metric 2 - Number of Edges (top), Number of Nodes (bottom) 31

7 Appearing only in Malware or Benign - API Call Frequency all(top), frequency above 15% (bottom) 33

8 Common API Calls Appearance Frequency - Cutoff Appearance Frequency 35% 35

9 Benign-Benign Kernel Values for Dataset Size 50 (top-left), 100 (top-right), 150 (middle-left), 200 (middle-right), 250 (bottom-left), and 300 (bottom-right) 38

10 Malware-Malware Kernel Values for Dataset Size 50 (top-left), 100 (top-right), 150 (middle-left), 200 (middle-right), 250 (bottom-left), and 300 (bottom-right) 39

11 Benign-Malware Kernel Values for Dataset Size 50-50 (top-left), 100-100 (top-right), 150-150 (middle-left), 200-200 (middle-right), 250-250 (bottom-left), and 300-300 (bottom-right) 41

12 SVM Accuracy for the Different Dataset Sizes and Training to Testing Splits 43

13 SVM ROC-AUC for the Different Dataset Sizes and Training to Testing Splits 44
14 SVM Precision for the Different Dataset Sizes and Training to Testing Splits 46
15 SVM Recall for the Different Dataset Sizes and Training to Testing Splits 46

References

[1] K.-H.-T. Dam and T. Touili, "Malware detection based on graph classification," in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, SCITEPRESS - Science and Technology Publications, 2017.

[2] P. D. C. Kelly Bissel, Ryan LaSalle, *Ninth Annual Cost of Cybercrime Study*. The Cost of Cybercrime, Ponemon Institue LLC, Accenture plc, 2019.

[3] Verizon, "Data breach investigations report 2020." `https://enterprise.verizon.com/resources/reports/dbir/`, year = 2020, note = Online; accessed 27-September-2020.

[4] 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, Jan. 2018.

[5] H. E. Merabet and A. Hajraoui, "A survey of malware detection techniques based on machine learning," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 1, 2019.

[6] Y. Ding, S. Zhu, and X. Xia, "Android malware detection method based on function call graphs," in *Neural Information Processing*, pp. 70–77, Springer International Publishing, 2016.

[7] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, Aug. 2019.

[8] N. M. Kriege, F. D. Johansson, and C. Morris, "A survey on graph kernels," *Applied Network Science*, vol. 5, Jan. 2020.

[9] `https://www.nsa.gov/resources/everyone/ghidra/`, Released on April 4th, 2019.

[10] I. Boneva, A. Rensink, M. Kurbán, and J. Bauer, "Graph abstraction and abstract graph transformation," *Measurement Science Review - MEAS SCI REV*, 01 2007.

[11] P. Moriano, J. Pendleton, S. Rich, and L. Camp, "Stopping the insider at the gates: Protecting organizational assets through graph mining," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 9, pp. 4–29, 03 2018.

[12] `https://www.microsoft.com/el-gr/software-download/windows10ISO`.

[13] `https://git-scm.com/downloads`.

[14] `https://www.nsa.gov/resources/everyone/ghidra/`.

[15] `http://www.codeblocks.org/downloads/26`.

[16] `https://virusshare.com/`.

[17] `https://www.virustotal.com/gui/`.

[18] `https://docs.microsoft.com/en-us/windows/win32/apiindex/api-index-portal`.

[19] K. Avrachenkov, P. Chebotarev, and D. Rubanov, "Kernels on graphs as proximity measures," vol. 10519, pp. 27–41, 09 2017.

[20] J. Ah-Pine, "Normalized kernels as similarity indices," pp. 362–373, 06 2010.

[21] S. V. N. Vishwanathan, K. M. Borgwardt, I. R. Kondor, and N. N. Schraudolph, "Graph kernels," *CoRR*, vol. abs/0807.0093, 2008.

[22] W. Imrich and S. Klavzar, *Product Graphs, Structure and Recognition*. 01 2000.

[23] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," vol. 129-143, pp. 129–143, 01 2003.

[24] B. Schölkopf, "The kernel trick for distances," vol. 13, pp. 301–307, 01 2000.

[25] T. Hofmann, B. Schölkopf, and A. Smola, "Kernel methods in machine learning," *The Annals of Statistics*, vol. 36, 01 2007.

[26] M. Sugiyama and K. Borgwardt, "Halting in random walk kernels," in *NIPS*, 2015.

[27] M. Fisher, M. Savva, and P. Hanrahan, "Characterizing structural relationships in scenes using graph kernels," *ACM Trans. Graph.*, vol. 30, p. 34, 07 2011.

[28] L. Jia, B. Gaüzère, and P. Honeine, "Graph kernels based on linear patterns: Theoretical and experimental comparisons," 2019.

[29] G. Simões, H. Galhardas, and D. Martins de Matos, "A labeled graph kernel for relationship extraction," 02 2013.

[30] M. Sugiyama, M. E. Ghisu, F. Llinares-López, and K. Borgwardt, "graphkernels: R and python packages for graph comparison," *Bioinformatics*, vol. 34, no. 3, pp. 530–532, 2017.

[31] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis, "Grakel: A graph kernel library in python," *Journal of Machine Learning Research*, vol. 21, no. 54, pp. 1–5, 2020.

[32] A. Calleja, J. Tapiador, and J. Caballero, "The malsource dataset: Quantifying complexity and code reuse in malware development," *IEEE Transactions on Information Forensics and Security*, vol. 14, pp. 3175–3190, 2019.

[33] M. Protsenko and T. Müller, "Android malware detection based on software complexity metrics," 09 2014.



ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Πολυτεχνικη Σχολη

Τμημα Ηλεκτρολογων Μηχανικων και Τεχνολογιας Υπολογιστων

Διπλωματικη Εργασια

Μεθοδολογίες Ταξινόμησης Κακόβουλου Λογισμικού

Βασίλειος Τσουβαλας

Επιβλέπων Καθηγητής: Δημήτριος Σερπανος

Πάτρα, Οκτώβριος 2020

Περιεχόμενα

1 Εισαγωγή **2**

1.1 Το Πρόβλημα του Κακόβουλου Λογισμικού 2

1.2 Σύγχρονες Λύσεις 2

2 Προτεινόμενη Λύση **3**

3 Εργαλεία και Δείγματα **4**

3.1 Εργαλεία 4

3.2 Δείγματα 4

4 Υλοποίηση **5**

4.1 Ανάλυση Κώδικα 5

4.2 Γράφοι Κλήσεων API 6

4.3 Ταξινόμηση 7

4.3.1 Σύγκριση Γράφων - Random Walk Graph Kernel (RDWGK) 7

4.4 Support Vector Machines 9

5 Μετρήσεις - Πειράματα **10**

6 Αποτελέσματα **11**

6.1 Kernel 11

6.2 Support Vector Machine 13

7 Συμπεράσματα **14**

Βιβλιογραφία **15**

1 Εισαγωγή

1.1 Το Πρόβλημα του Κακόβουλου Λογισμικού

Υπολογίζεται πως μέχρι το 2023, τα περουσιακά στοιχεία που βρίσκονται σε ρίσκο λόγω του παγκόσμιου κυβερνοεγκλήματος ενδέχεται να φτάσουν τα 5.2 τρισεκατομμύρια USD [1]. Δεδομένου ότι το 28% των συνολικών κυβερνοεπιθέσεων αφορά παραβιάσεις με τη χρήση κακόβουλου λογισμικού[2], γίνεται προφανές ότι η ανίχνευση κακόβουλου λογισμικού είναι ένα από τα πιό σημαντικά κεφάλαια της κυβερνοασφάλειας διεθνώς.

1.2 Σύγχρονες Λύσεις

Η ανίχνευση κακόβουλου λογισμικού γίνεται μέσω δύο βασικών τύπων ανάλυσης: στατική και δυναμική ανάλυση. Κατά τη στατική ανάλυση, επιχειρείται η συλλογή πληροφορίας μέσω ανάλυσης κώδικα χωρίς να εκτελείται το πρόγραμμα, ενώ κατά τη δυναμική ανάλυση, το πρόγραμμα εκτελείται σε ένα ασφαλές περιβάλλον και η συλλογή πληροφορίας γίνεται κατά την εκτέλεση αυτή, επιτρέποντας την παρατήρηση περαιτέρω στοιχείων κατά την αλληλεπίδραση του εκτελέσιμου με του υπολογιστή. Την τελευταία δεκαετία, η χρήση τεχνικών μηχανικής μάθησης και εξόρυξης δεδομένων λαμβάνει χώρα όλο και περισσότερο στις προσπάθειες ανίχνευσης κακόβουλου λογισμικού[3].

2 Προτεινόμενη Λύση

Στην παρούσα εργασία, ακολουθώντας τις σύγχρονες προσπάθειες επίλυσης του προβλήματος[4], εφαρμόζεται μια λύση βασισμένη στη στατική ανάλυση και τη μοντελοποίηση γράφων[5][6], η οποία σε συνδυασμό με γνωστές τεχνικές ταξινόμησης, επιτρέπει την ανίχνευση του κακόβουλου λογισμικού.

Ξεκινάμε με ένα σύνολο από κακόβουλα και καλόβουλα εκτελέσιμα MS Windows λογισμικά. Για κάθε λογισμικό πραγματοποιούμε την αποσυναρμολόγησή του (disassembly) και εξάγουμε πληροφορίες, οι οποίες μας επιτρέπουν να μοντελοποιήσουμε τις κλήσεις του εκτελέσιμου δείγματος στο Windows API σε Γράφο Ελέγχου Ροής (Control Flow Graph). Με βάση το Γράφο Ελέγχου Ροής και άλλες, εξαχθείσες από τη στατική ανάλυση, πληροφορίες λαμβάνουμε το Γράφο API Κλήσεων. Εν συνεχεία, χρησιμοποιώντας βασικές αρχές της Θεωρίας Γράφων, καταλήγουμε σε μία αφηρημένη μορφή του Γράφου Κλήσεων API, τον Αφηρημένο Γράφο API Κλήσεων. Η διαδικάσια που προτείνεται ως λύση βασίζεται στη σύγκριση αυτών των γράφων. Η διανυσματικοποίηση της σύγκρισης των Αφηρημένων Γράφων API Κλήσεων επιτυγχάνεται μέσω ενός συγκεκριμένου graph kernel, του οποίου τα αποτελέσματα εισάγονται για ταξινόμηση σε ένα Support Vector Machine (SVM), όπου και τα εκτελέσιμα λογισμικά ταξινομούνται σε καλόβουλα ή κακόβουλα. Σημειώνεται πως στην παρούσα εργασία, αντιμετωπίζουμε το πρόβλημα της ανίχνευσης κακόβουλου λογισμικού, ως ένα πρόλημα δυαδικής ταξινόμησης και άρα ο στόχος είναι να κατηγοριοποιηθεί ένα λογισμικό ως κακόβουλο ή καλόβουλο.

3 Εργαλεία και Δείγματα

3.1 Εργαλεία

Παρουσιάζουμε την εργαλειοθήκη που συγκροτήθηκε για την προτεινόμενη λύση και ειδικότερα τα εργαλεία που χρησιμοποίηθηκαν στα τρία βασικά μέρη αυτής.

- Ανάλυση εκτελέσιμων δειγμάτων: NSA Ghidra[7] για ανάλυση και αποσυναρμολόγηση, κώδικας σε Java για εξαγωγή πόρων
- Γράφοι API Κλήσεων: κώδικας σε Python για μοντελοποίηση γράφων και εφαρμογή βασικών αρχών θεωρίας γράφων
- Ταξινόμηση: SVM's με τη χρήση της γλώσσας Python

Σημειώνουμε πως για μεγάλο μέρος της διαδικασίας χρησιμοποιήθηκε server (8-core CPU, 16GB RAM), λόγω της ανάγκης για μεγαλύτερη υπολογιστική ισχύ.

3.2 Δείγματα

Συγκεντρώθηκαν καλόβουλα και κακόβουλα εκτελέσιμα δείγματα για Windows. Ειδικότερα, συγκεντρώθηκαν 997 κακόβουλα δείγματα από το VirusShare[8] και 567 καλόβουλα δείγματα, τα οποία αποτελόυνται από αρχεία εγκατάστασης έγκυρων και αξιόπιστων λογισμικών όπως το λειτουργικό σύστημα των Windows[9], το Git[10], Cygwin[11], Codeblocks[12] κ.α.

4 Υλοποίηση

Παρακάτω περιγράφουμε πιό αναλυτικά τα βασικά μέρη της προτεινενης λύσης.

4.1 Ανάλυση Κώδικα

Κατά την ανάλυση κώδικα κάθε εκτελέσιμου, μέσω της χρήσης του Ghidra, λαμβάνουμε τον αποσυναρμολογημένο κώδικα για το συγκεκριμένο εκτελέσιμο δείγμα. Τα βασικά στοιχεία που προκύπτουν από την ανάλυση κάθε δείγματος είναι:

- Τα Μπλοκ Κώδικα (Codeblocks): σημεία ελέγχου του προγράμματος, των οποίων η αλληλοσύνδεση δίνει όλες τις πιθανές διαδρομές εκτέλεσης του προγράμματος
- API κλήσεις: κλήσεις συναρτήσεων του API των Windows[13] τις οποίες πραγματοποιεί το εκτελέσιμο δείγμα

Από τα Μπλοκ Κώδικα και την πληροφορία για τη διασυνδεσιμότητά τους, λαμβάνουμε το Γράφο Ελέγχου Ροής του εκτελέσιμου. Σημειώνουμε πως κάθε Μπλοκ Κώδικα δύναται να οδηγεί (με την έννοια της διαδρομής εκτέλεσης) σε άλλο Μπλοκ Κώδικα ή σε κλήση API. Ο Γράφος Ελέγχου Ροής ορίζεται ως ο κατευθυντικός γράφος $G = (N, E)$, όπου το N είναι ένα πεπερασμένο σύνολο κόμβων και E ένα πεπερασμένο σύνολο ακμών. Με τον παραπάνω ορισμό, έχουμε ότι το (n_1, n_2) σημαίνει ότι το Μπλοκ Κώδικα n_1 οδηγεί στο Μπλοκ Κώδικα n_2.

4.2 Γράφοι Κλήσεων API

Δεδομένου του Γράφου Ελέγχου Ροής του εκτελέσιμου δείγματος και γνωρίζοντας τις κλήσεις API που πραγματοποιεί το εκτελέσιμο καθώς και το συγκεκριμένο Μπλοκ Κώδικα από όπου καλούνται αυτές, είμαστε σε θέση να μοντελοποιήσουμε το Γράφο API Κλήσεων. Η λογική που ακολουθείται έγκειται στην αντικατάσταση των κόμβων του Γράφου Ελέγχου Ροής, με κόμβους που αναπαριστούν τις API κλήσεις που πραγματοποιεί το Μπλοκ Κώδικα του συγκεκριμένου κόμβου. Γίνεται εύκολα αντιληπτό πως το μέγεθος του Γράφου API Κλήσεων (αριθμός κόμβων και ακμών) είναι πολύ μεγαλύτερο από εκείνο του Γράφου Ελέγχου Ροής. Ο Γράφος API Κλήσεων είναι ένας κατευθυντικός γράφος $G_{APICall} = (F_{API}, E_{API})$, όπου το F_{API} είναι ένα πεπερασμένο σύνολο κόμβων, το οποίο αναπαριστά τις API κλήσεις στις οποίες μπορεί να φτάσει το πρόγραμμα εκκινώντας από το αρχικό Μπλοκ Κώδικα και E_{API} είναι ένα πεπερασμένο σύνολο ακμών, το οποίο αναπαριστά τις API κλήσεις στις οποίες μπορεί να φτάσει μέσω του Μπλοκ Κώδικα προορισμού.

Το μέγεθος του Γράφου API Κλήσεων είναι πολύ μεγάλο και δεν επιτρέπει τη σύγκριση δύο τέτοιων γράφων. Για το λόγο αυτό, χρησιμοποιώντας βασικές αρχές της θεωρία γράφων (αντιμετωπίζουμε το Γράφο API Κλήσεων ώς πολλά επίπεδα διμερών γράφων), μετασχηματίζουμε το Γράφο API Κλήσεων σε έναν Αφηρημένο Γράφο API Κλήσεων. Ο Αφηρημένος Γράφος API Κλήσεων έχει τόσους κόμβους όσες API Κλήσεις καλούνται από το εκτελέσιμο (ένας κόμβος για κάθε μοναδική κλήση) και οι ακμές του εκφράζουν την ύπαρξη σύνδεσης ανάμεσα στις κλήσεις αυτές στο Γράφο API Κλήσεων. Με τη μετατροπή αυτή, χάνεται η κατευθυντηκότητα, όμως διατηρούνται όλες οι συνδέσεις των API Κλήσεων και ο γράφος πλέον έχει διαχειρήσιμο μέγεθος.

4.3 Ταξινόμηση

Η διαδικασία ταξινόμησης χωρίζεται σε δύο μέρη. Το πρώτο περιγράφει τη μετάβαση από τη σύγκριση δύο Αφηρημένων Γράφων API Κλήσεων σε ένα γραμμικό μέγεθος, το kernel, το οποίο εισάγεται στο δευτερο και τελικό μέρος της ταξινόμησης, το οποίο αφορά τα Support Vector Machines (SVM).

4.3.1 Σύγκριση Γράφων - Random Walk Graph Kernel (RDWGK)

Έχοντας εξάγει τον Αφηρημένο Γράφο API Κλήσεων για καθένα από τα εκτελέσιμα δείγματα, προχωράμε στη σύγκριση των γράφων.

Ο αλγόριθμος Τυχαίας Διαδρομής (Random Walk) είναι ένας ευρέως διαδεδομένος αλγόριθμος, ο οποίος επιστρέφει τυχαίες διαδρομές ενός γράφου, με τη λογική να βασίζεται στην εκκίνηση από ένα κόμβο του γράφου και στην τυχαία επιλογή για την επίσκεψη γειτονικών κόμβων.

Ένα graph kernel είναι μία συνάρτηση η οποία υπολογίζει το εσωτερικό γινόμενο δύο γράφων και με τον τροπο αυτο παρέχει μία μέτρηση της ομοιότητας των δύο γράφων[14][15]. Η χρησιμότητα του graph kernel έγκειται στο ότι επιτρέπει να γραμμικοποιηθεί ένα μη-γραμμικό πρόβλημα[16][17], όπως είναι η σύγκριση γράφων, και εν συνεχεία, επιτρέπει τη χρήση αλγόριθμων ταξινόμησης με το παρεχόμενο αποτέλεσμα. Στην παρούσα εργασία, χρησιμοποιείται μία ειδική περίπτωση graph kernel, το Random Walk Graph Kernel (RDWGK)[18], ώστε να συγκριθούν όλοι οι Αφηρημένοι Γράφοι API Κλήσεων ανά δύο, και έπειτα να εισαχθούν οι τιμές των kernel στο SVM για ταξινόμηση.

Για δύο γράφους $G_1 = (N_1, E_1)$ και $G_2 = (N_2, E_2)$, ορίζεται ως Γράφος Γι-

νομένου $G_\times = (N_\times, E_\times)$ ένα γράφος των ζευγαριών των κόμβων των G_1 και G_2, όπου δύο κόμβοι του $G_\times$ συνδέονται, αν και μόνον αν οι αντιστοιχοι κομβοι των G_1 και G_2 γειτνιάζουν. Ο λόγος που χρησιμοποιείται ο Γράφος Γινομένου είναι γιατί το RDWGK ανάμεσα σε δύο γράφους είναι το ίδιο με το RDWGK του Γράφου Γινομένου των δύο γράφων[19]. Σημειώνουμε ότι εάν A_1 και A_2 οι πίνακες γειτνίασης των G_1 και G_2 αντίστοιχα, τότε ο πίνακας γειτνίασης του Γράφου Γινομένου $G_\times$ είναι $A_\times = A_1 \times A_2$.

Το RDWGK του Γράφου Γινομένου ορίζεται ως:

$$\kappa(G_1, G_2) = \sum_{k=0}^{T} \mu(k) q_\times^\intercal A_\times^k p_\times$$

όπου:

- $A_\times$ είναι ο πίνακας γειτνίασης του Γράφου Γινομένου και κατέπέκταση το A^k αναπαριστά την πιθανότητα ταυτόχρονων τυχαίων διαδρομών μήκου k στους G_1 και G_2.
- $p_\times$ και $q_\times$ αναπαριστούν την αρχική κατανομή πιθανότητας (αρχικός κόμβος) και την πιθανότητα να σταματήσει η διαδρομή στο συγκεκριμένο κόμβο του Γράφου Γινομένου, αντίστοιχα - οι συγκεκριμένες πιθανότητες δεν λαμβάνονται υπόψιν στην παρούσα εργασία, καθώς δεν γνωρίζουμε αυτές τις πιθανότητες[14]
- T είναι το μέγιστο μήκος των τυχαίων διαδρομών
- $\mu(k) = \lambda^k \in [0, 1]$ είναι ένας παράγοντας που ελέγχει τη σημαντικότητα του μήκους των τυχαίων διαδρομών μέσω του οποίου πάντα επιτυγχάνεται σύγκλιση του αθροίσματος και το RDWGK είναι καλά ορισμένο.

Με δεδομένα ορισμένα ειδικά χαρακτηριστικά των Αφηρημένων Γράφων API Κλήσεων (κάθε κόμβος είναι μοναδικός και αναπαριστά μία κλήση API, κάθε κλήση API

έχει μοναδικό όνομα), επιτυγχάνεται να μειωθεί σημαντικά το απαιτούμενο κόστος υπολογισμού και ο χρόνος εκτέλεσης των πράξεων για την εύρεση των RDWGK.

4.4 Support Vector Machines

Αφού υπολογιστούν οι τιμές του RDWGK γιά όλα τα πιθανά ζευγάρια γράφων, τις χρησιμοποιούμε στην ταξινομητική διαδικάσια των Support Vector Machines (SVM). Δεδομένου ότι ο SVM είναι αλγόριθμος μάθησης ο οποίος βασίζεται σε kernel, έχοντας ήδη γραμμικοποιήσει το μη-γραμμικό πρόβλημα σύγκρισης γράφων επιτρέπει τη χρήση γραμμικού ταξινομητη. Ο στόχος του SVM είναι να ταξινομηθεί η γραμμικοποιημένη πληροφορία με τρόπο που επιτρέπει υψηλή ακρίβεια για την ταξινόμηση νέων δεδομένων.

Εκπαιδεύουμε (training) και πραγματοποιούμε ελέγχους (testing) στο ταξινομητή SVM χρησιμοποιώντας 9 διαφορετικές υποδιαιρέσεις του συνόλου εκπαίδευσης-ελέγχου, με την έννοια ότι ξεκινώντας από ένα συγκεκριμένο αριθμό τιμών kernel (ίσος αριθμός τιμών kernel από καλόβουλα και κακόβουλα εκτελέσιμα) χωρίζουμε τυχαία τις τιμές αυτές κατά 10%-90% για εκπαίσευση-έλεγχο και με διαστήματα 10% φτάνουμε σε 90%-10% εκπαίδευση-έλεγχο για τις τιμές του kernel.

Όλη η παραπάνω διαδικασία πραγματοπιείται 100 φορές και ως αποτελέσματα του SVM, παίρνουμε τις μέσες τιμές για accuracy, precision, recall, ROC-AUC. Κάθε σχήμα ταξινόμησης γίνεται για τιμές kernel οι οποίες περιέχουν ίσο αριθμό προερχόμενο από καλόβουλα και κακόβουλα εκτελέσιμα δείγματα και το πειράμα γίνεται για σύνολο δεδομένων των 50, 100, 150, 200, 250 και 300 καλόβουλων και κακόβουλων δειγμάτων.

5 Μετρήσεις - Πειράματα

Κατά την εξαγωγή πληροφοριών από την στατική ανάλυση, παρουσιάζουμε τις παρακάτω μετρήσεις:

- Αριθμός Κλήσεων API σε σχέση με το μέγεθος του δείγματος
- Μετρήσεις Αραιότητας των Αφηρημένων Γράφων API Κλήσεων σε σχέση με τον αριθμό κόμβων, τον αριθμό ακμών και το μέγεθος του δείγματος
- Συχνότητα εμφάνισης API Κλήσεων αποκλειστικά σε καλόβουλα ή κακόβουλα δείγματα καθώς και τη συχνότητα εμφάνισης των κοινών Κλήσεων API

Με βάση τις παραπάνω πειραματικές μετρήσεις, εξάγουμε συμπεράσματα βασισμένα στις αποκλίσεις που παρουσιάζονται ανάμεσα στα καλόβουλα και τα κακόβουλα δείγματα. Ειδικότερα, εξάγουμε τα παρακάτω συμπεράσματα:

– Παρατηρείται ότι ένας σημαντικός αριθμός κακόβουλων δειγμάτων έχουν τον ίδιο αριθμό API Κλήσεων, αλλά μεγάλη απόκλιση στο μέγεθός τους, εν αντιθέσει με τα καλόβουλα δείγματα, τα οποία παρουσιάζουν μία αναμενόμενη σχέση στη συγκεκριμένη μέτρηση (το μέγεθος του δείγματος μεγαλώνει με τον αριθμό των API Κλήσεων).

– Μεγάλες αποκλίσεις παρατηρούνται στις μετρήσεις αραιότητας (οι μετρήσεις αραιότητας δίνονται από τα κλάσματα: κόμβοι προς ακμές και ακμές προς κόμβους) σε σχέση με τον αριθμό των ακμών και τον αριθμό των κόμβων, καθώς τα κακόβουλα δείγματα παρουσιάζονται λιγότερο διασυνδεδεμένα, σε σχέση με τα καλόβουλα, όσον αφορά τους Αφηρημένους Γράφους API Κλήσεων.

– Διακρίνονται συγκεκριμένες API Κλήσεις, οι οποίες είτε εμφανίζονται συχνά μόνο στα καλόβουλα ή μόνο στα κακόβουλα δείγματα, ή παρουσιάζουν

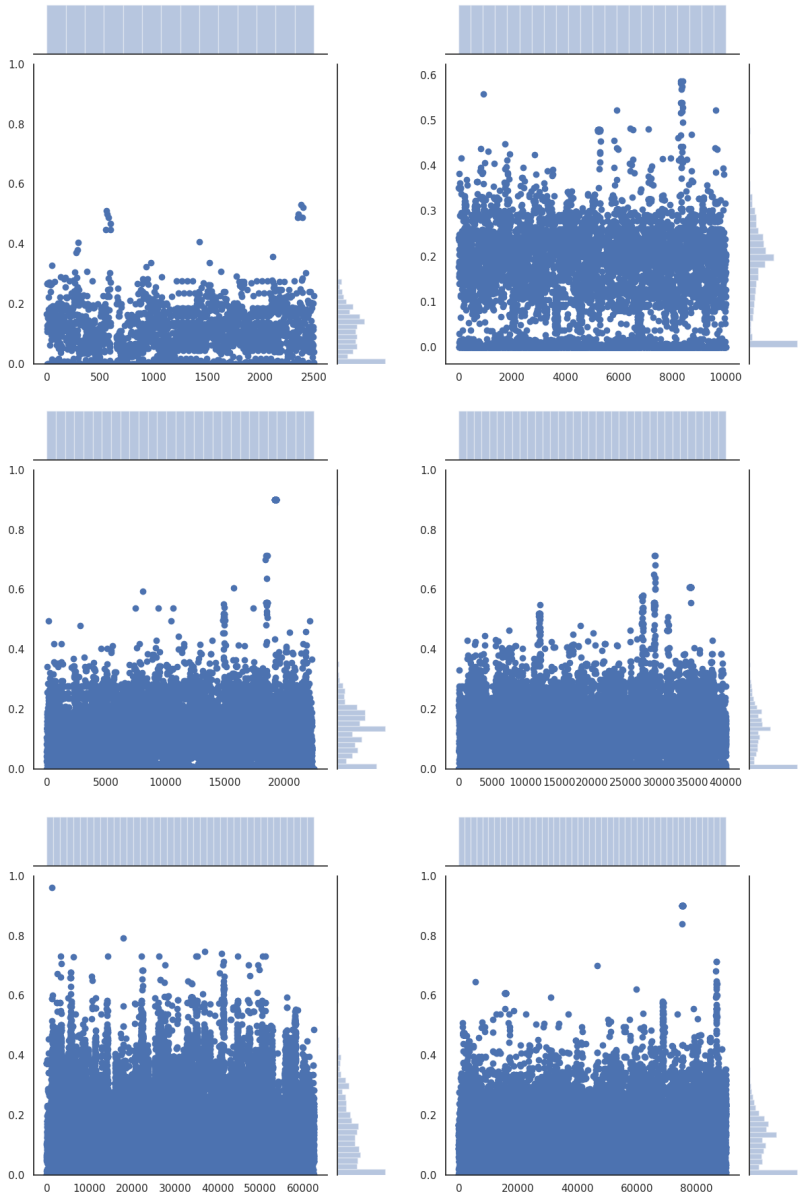
μεγάλη διαφορά στη συχνότητα εμφάνισης τους ανάμεσα σε καλόβουλα και κακόβουλα δείγματα.

Οι παραπάνω παρατηρήσεις δύναται να χρησιμοποιηθούν ως προκαταρκτικά κριτήρια ταξινόμησης, καθώς επίσης μπορούν να χρησιμοποιηθούν για να βελτιώσουν την απόδοση ενός ταξινομητή.

6 Αποτελέσματα

6.1 Kernel

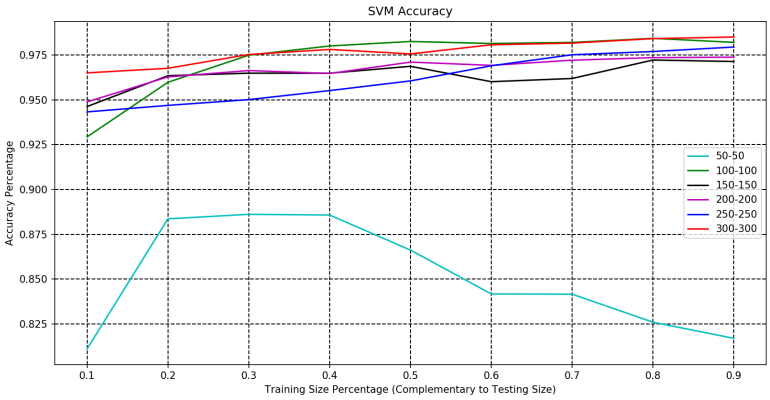
Όπως αναφέρθηκε παραπάνω, η τιμή του kernel είναι μία μέτρηση της ομοιότητας δύο γράφων. Στο Σχήμα 1, παρουσιάζονται οι τιμές kernel για τα διαφορετικά μεγέθη δειγμάτων και σημειώνεται πως οι τιμές αυτές είναι κανονικοποιημένες στη μονάδα (μηδενική τιμή σημαίνει μηδενική ομοιότητα και τιμή ίση με 1 σημαίνει απόλυτη ομοιότητα-ταύτιση). Παρατηρείται πως οι τιμές kernel συγκεντρώνονται γύρω από χαμηλα ποσοστά ομοιότητας, το οποίο υποδεικνύει ότι ο Αφηρημένος Γράφος API Κλήσεων των δειγμάτων είναι ένα έγκυρο κριτήριο ταξινόμησης.



Σχήμα 1: Τιμές kernel Καλόβουλων-Κακόβουλων δειγμάτων για 50 (πάνω αριστερά), 100 (πάνω δεξιά), 150 (μεσαίο αριστερά), 200 (μεσαίο δεξιά), 250 (κάτω αριστερά) και 300 (κάτω δεξιά) δείγματα

6.2 Support Vector Machine

Τα καλύτερα αποτελέσματα του ταξινομητή SVM παρατηρούνται για την πιό ευνοϊκή περίπτωση ταξινόμησης, η οποία συμβαίνει για το μεγαλύτερο δυνατό σύνολο δειγμάτων, 300 καλόβουλα και 300 κακόβουλα, και για τη διαίρεση των δειγμάτων κατά 90%-10% ανάμεσα σε εκπαίδευση και έλεγχο αντίστοιχα. Επιτυγχάνεται ποσοστό ακρίβειας 98.25% (Σχήμα 2), το οποίο σημαίνει πως το μέγιστο περιθώριο λάθους είναι 1.75%. Σημειώνουμε πως οι μετρήσεις απόδοσης του ταξινομητή



Σχήμα 2: Ακρίβεια SVM για Διαφορετικά Μεγέθη Συνόλου Δειγμάτων και Διαφορετικές Υποδιαιρέσεις Δειγμάτων Εκπαίδευσης-Ελέγχου

SVM αναφορικά με Precision, Recall και ROC AUC παρουσιάζουν παρόμοια χαρακτηριστικά καθώς όσο γίνονται ευνοϊκότερα τα δεδομένα ταξινόμησης, τόσο βελτιώνεται η απόδοση του ταξινομητή σε όλα τα επίπεδα.

7 Συμπεράσματα

Ο στόχος της παρούσας εργασίας ήταν να προταθεί μία λύση στο πρόβλημα της ανίχνευσης κακόβουλου λογισμικού. Η προτεινόμενη λύση χρησιμοποιεί τεχνικές της στατικής ανάλυσης για να αποσυναρμολογηθούν καλόβουλα και κακόβουλα δείγματα, ώστε να καταστεί δυνατή η μελέτη τους. Βασιζόμενοι στη μελέτη αυτη, αναπτύχθηκαν ad hoc μηχανισμοί μοντελοποίησης γράφων για να ξεπεραστούν προβλήματα όπως η απαίτηση μεγάλης υπολογιστικής ισχύς και η γραμμικοποίηση μη-γραμμικών στοιχείων, ώστε να επιτυγχάνεται αποδοτικότερος χειρισμός της πληροφορίας. Το σχήμα ταξινόμησης το οποίο χρησιμοποιήθηκε απέφερε πολλά υποσχόμενα αποτελέσματα και οι περαιτέρω πειραματικές μετρήσεις που πραγματοποιήθηκαν επιτρέπουν τη στοιχειοθέτηση βάσιμων υποθέσεων για μελλοντική έρευνα.

Βιβλιογραφία

[1] P. D. C. Kelly Bissel, Ryan LaSalle, *Ninth Annual Cost of Cybercrime Study*. The Cost of Cybercrime, Ponemon Institue LLC, Accenture plc, 2019.

[2] Verizon, "Data breach investigations report 2020." https://enterprise.verizon.com/resources/reports/dbir/, year = 2020, note = Online; accessed 27-September-2020.

[3] 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, Jan. 2018.

[4] H. E. Merabet and A. Hajraoui, "A survey of malware detection techniques based on machine learning," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 1, 2019.

[5] Y. Ding, S. Zhu, and X. Xia, "Android malware detection method based on function call graphs," in *Neural Information Processing*, pp. 70–77, Springer International Publishing, 2016.

[6] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, Aug. 2019.

[7] https://www.nsa.gov/resources/everyone/ghidra/, Released on April 4th, 2019.

[8] https://virusshare.com/.

[9] https://www.microsoft.com/el-gr/software-download/windows10ISO.

[10] https://git-scm.com/downloads.

[11] https://www.nsa.gov/resources/everyone/ghidra/.

[12] http://www.codeblocks.org/downloads/26.

[13] https://docs.microsoft.com/en-us/windows/win32/apiindex/api-index-portal.

[14] K. Avrachenkov, P. Chebotarev, and D. Rubanov, "Kernels on graphs as proximity measures," vol. 10519, pp. 27–41, 09 2017.

[15] J. Ah-Pine, "Normalized kernels as similarity indices," pp. 362–373, 06 2010.

[16] B. Schölkopf, "The kernel trick for distances," vol. 13, pp. 301–307, 01 2000.

[17] T. Hofmann, B. Schölkopf, and A. Smola, "Kernel methods in machine learning," *The Annals of Statistics*, vol. 36, 01 2007.

[18] S. V. N. Vishwanathan, K. M. Borgwardt, I. R. Kondor, and N. N. Schraudolph, "Graph kernels," *CoRR*, vol. abs/0807.0093, 2008.

[19] W. Imrich and S. Klavzar, *Product Graphs, Structure and Recognition*. 01 2000.