Universidade Federal do Amazonas
Instituto de Computação
Programa de Pos-Graduação em Informática

Luis Miguel Rojas Aguilera

Metamorphic malware identification through Annotated Data Dependency Graphs' dataset indexing

Manaus, 2018

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

R741m

Rojas Aguilera, Luis Miguel

Metamorphic malware identification through Annotated Data Dependency Graphs’ dataset indexing / Luis Miguel Rojas Aguilera. 2018

106 f.: il. color; 31 cm.

Orientador: Eduardo James Pereira Souto

Orientador: Gilbert Martins Breves

Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. malware metamórfico. 2. grafos de dependência de dados. 3. indexação de bases de grafos. 4. comparação de grafos. 5. subgraph isomorphism. I. Souto, Eduardo James Pereira II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FOLHA DE APROVAÇÃO

"Metamorphic malware identification through Annotated Data Dependency Graphs´ datasets indexin"

LUIS MIGUEL ROJAS AGUILERA

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Eduardo James Pereira Souto - PRESIDENTE

Prof. Eduardo Luzeiro Feitosa - MEMBRO INTERNO

Profa. Eulanda Miranda dos Santos - MEMBRO INTERNO

Prof. André Ricardo Abed Grégio - MEMBRO EXTERNO

Manaus, 23 de Março de 2018

Luis Miguel Rojas Aguilera

Metamorphic malware identification through Annotated Data Dependency Graphs' dataset indexing

Dissertation submitted to Programa de Pós-Graduação em Informática of Instituto de Computação at Universidade Federal do Amazonas in partial fulfillment of the requirements for the Degree of Master's in Computer Sciences.

Advisors: PhD. Professor Eduardo James Pereira Souto and PhD. Professor Gilbert Breves Martins

Manaus, 2018

Rojas, Luis Miguel.
Metamorphic malware identification through Annotated Data Dependency Graphs' dataset indexing
88 pages
Dissertation (Master) - Instituto de Computação of Universidade Federal do Amazonas. Departamento de Informática.

1. code metamorphism
2. malware detection
3. graphs dataset indexing
4. data dependency graphs

Judging commission:

PhD. Professor
Eulanda Miranda dos Santos

PhD. Professor
André Ricardo Abed Grégio

PhD. Professor
Eduardo Luzeiro Feitosa

Acknowledgments

I would like to thank Professors Eduardo James Pereira Souto PhD and Gilbert Breves Martins PhD, for their guidance, encouragement and patience through out the project. I am thankful for the professors of the Postgraduate Program in Informatics (PPGI) at the Universidade Federal do Amazonas (UFAM) for their valuable training, advise and feedback. I also thank the Coordination for the Improvement of Higher Education Personnel (CAPES) for providing financial aid.

Finally, this project would not have been possible without the support of my family and, especially my loving wife, Yadini Perez Lopez.

Resumo

A mutação de código e o metamorfismo têm sido empregados com sucesso para a criação e proliferação de novas instâncias de *malware* a partir de códigos maliciosos existentes. Com estas técnicas é possível modificar a estrutura de um código sem alterar as funcionalidades originais para obter novas instâncias que não se encaixam nos padrões estruturais e de comportamento presentes em bases de conhecimento dos sistemas de identificação de *malware*, dificultando assim a detecção. Pesquisas anteriores que abordam a detecção de *malware* metamórfico podem ser agrupadas em: identificação por meio do *matching* de assinaturas de código e detecção baseada em modelos de classificação. O *matching* de assinaturas de código tem apresentado taxas de falsos positivos inferiores às apresentadas pelos modelos de classificação, uma vez que estas estruturas são resilientes aos efeitos do metamorfismo e permitem melhor discriminação entre as instâncias. Entretanto a complexidade temporal dos algoritmos de comparação impedem a aplicação desta técnica em sistemas de detecção reais. Por outro lado, a detecção baseada em modelos de classificação apresenta menor complexidade algorítmica, porém a capacidade de generalização dos modelos se vê afetada pela versatilidade de padrões que podem ser obtidos por médio da aplicação de técnicas de metamorfismo. Para superar estas limitações, este trabalho apresenta uma metodologia para a identificação de *malware* metamórfico através da comparação de grafos de dependência de dados anotados extraídos de *malwares* conhecidos e de instâncias suspeitas no momento da análise. Para lidar com a complexidade dos algoritmos de comparação, permitindo assim a utilização da metodologia em sistemas de detecção reais, as bases de grafos são indexadas empregando algoritmos de aprendizagem de máquina, resultando em modelos de classificação multiclasse que discriminam entre famílias de *malwares* a partir das características estruturais dos grafos. Resultados experimentais, utilizando um protótipo da metodologia proposta sobre uma base composta por 40,785 grafos extraídos de 4,530 instâncias de *malwares*, mostraram tempos de detecção inferiores aos 150 segundos para processar todas as instâncias e de criação dos modelos inferiores aos 10 minutos, bem como acurácia média superior à maioria de 56 ferramentas comerciais de detecção de *malware* avaliadas.

Palavras-chave: *malware* metamórfico, grafos de dependência de dados, indexação de bases de grafos, comparação de grafos

Abstract

Code mutation and metamorphism have been successfully employed to create and proliferate new malware instances from existing malicious code. With such techniques, it is possible to modify a code's structure without altering its original functions, so, new samples can be made that lack structural and behavioral patterns present in knowledge bases of malware identification systems, which hinders their detection. Previous research endeavors addressing metamorphic malware detection can be grouped into two categories: identification through code signature matching and detection based on models of classification. Matching code signatures presents lower false positive rates in comparison with models of classification, since such structures are resilient to the effects of metamorphism and allow better discrimination among instances, however, temporal complexity of matching algorithms prevents the application of such technique in real detection systems. On the other hand, detection based on classification models present less algorithmic complexity, however, a models' generalization capacity is affected by the versatility of patterns that can be obtained by applying techniques of metamorphism. In order to overcome such limitations, this work presents methods for metamorphic malware identification through matching annotated data dependency graphs, extracted from known malwares and suspicious instances in the moment of analysis. To deal with comparison algorithms' complexity, using these methods on real detection systems, the databases of graphs were indexed using machine learning algorithms, resulting in multi-class classification models that discriminated among malware families based on structural features of graphs. Experimental results, employing a prototype of the proposed methods from a database of 40,785 graphs extracted from 4,530 malware instances, presented detection times below 150 seconds for all instances, as well as higher average accuracy than 56 evaluated commercial malware detection systems.

Keywords: metamorphic malware, data dependency graph, graphs datasets indexing, graph matching

List of Figures

1.1 Quantity of malware samples found from the fourth quarter of 2015 to the third quarter of 2017 2

2.1 Sample of CFG extracted from assembly code in Table 2.1 11
2.2 Sample DDG extracted from code in 1 13
2.3 Example of an ADDG extracted from pseudocode 1 13
2.4 Example of equivalent instructions substitution 19
2.5 Representation of code reordering and dead code insertion. 21

3.1 Similarity scores between one iteration generated from NGVCK instances and benign files according to Baysa et. al approach . . . 27
3.2 Similarity scores between two iterations generated from NGVCK instances and benign files according to Baysa et. al approach . . . 28
3.3 Differences among metamorphic malware families in (Kim and Moon, 2010) 30
3.4 Similarity score among metamorphic variants of Evol malware family (Breves et.al,2015). 31
3.5 Similarity score among metamorphic variants of Polipmalware family (Breves et.al,2015). 31

4.1 Overview of the proposed approach for metamorphic malware detection. 36
4.2 Overview of feature vectors extraction phase. 37
4.3 Example of a CFG extracted from a function of program *ls* 39
4.4 Labeled DDG extracted from the CFG in Figure 4.3 39
4.5 Index induction and identification work-flows 44
4.6 Suspicions instance classification phase 47

5.1 Experimentation phases and steps. 49
5.2 Initial dataset splitting for training and testing. 53
5.3 Training-testing dataset configurations for model creation 53
5.4 Architecture for commercial tools virus detection rate lifting. . . . 58
5.5 Average disassembly time by file size ranges. 59
5.6 Feature space sizes for different datasets vs threshold employed for feature space reduction. 61
5.7 Accuracy score for Code Pervertor generated instances using only seeds for training. 63
5.8 Accuracy score for Code Pervertor's generated instances using seeds plus 20% of families for training. 63
5.9 Accuracy score of multi-class models trained with Seeds only and tested with Revert4 generated instances graphs' feature vectors. . 64
5.10 Accuracy score of multi-class models trained with Seeds plus 20% of Revert4 instances and tested with 80% of Revert4. 64
5.11 Training time with seeds plus 20% of each family of Code Pervertor generated instances. 65

5.12 Training time with seeds plus 20% of each family of Revert4 generated instances. 66
5.13 Times training with graphs extracted from seeds instances only. . 67
5.14 ROC Curves for OVR based models employing only seeds for training, using a threshold of feature space reduction of 0 and Code Pervertor instances for testing. 68
5.15 ROC Curves for OVR based models employing seeds and 20% of Code Pervertor for training , using a threshold of feature space reduction of 0 and the remaining 80% for testing. 68
5.16 ROC Curves for OVR based models employing only seeds for training, using threshold of feature space reduction of 0.8 and Code Pervertor for testing. 69
5.17 ROC Curves for OVR based models employing seeds and 20% of Code Pervertor for training, using threshold of feature space reduction of 0.8 and the remaining 80% for testing. 69
5.18 ROC Curves for OVR based models employing only seeds for training, using threshold of feature space reduction 0 and Rever4 generated instances for testing. 69
5.19 ROC Curves for OVR based models employing seeds and 20% of Revert4 generated instances for training, using threshold of feature space reduction 0 and remaining 80% for testing. 69
5.20 Accuracy of models trained with seeds only and tested with Code Pervertor generated instances. 70
5.21 Accuracy of models trained with seeds plus 20% of families of Code Pervertor generated instances and tested with the remaining 80%. 70

5.22 Accuracy of models trained with seeds only and tested with Revert4 generated instances. 71
5.23 Accuracy of models trained with seeds plus 20% of families of Revert4 generated instances and tested with the remaining 80%. . 71
5.24 Training times with graphs extracted from seeds instances only. . 72
5.25 Top 15 ranked models and antivirus for instances mutated with Revert4. 73
5.26 Top 15 ranked models and antivirus for instances mutated with Code Pervertor. 74

List of Tables

2.1 Assembly code sample 11

2.2 Example of hash signature extracted from code 15

2.3 Examples of techniques that can be applied to alter the code body without affecting its original functionalities 17

2.4 Examples of instruction reordering 18

2.5 Two generations of Win95/Regswap. 19

2.6 Metamorphic variants of the virus Win32/Evol. 20

3.1 Example of an IOM structure as employed by Canfora et. al . . . 23

3.2 Example of report extracted from notepad.exe as employed by Choudhary and Vidyarthi 24

3.3 Comparative table of state-of-the-art research on metamorphic malware detection 32

4.1 Example of mnemonics and their classes according to the x86asm[dot]net reference. 41

4.2 Examples of classifications given to assembly code instruction operands by mapping references in LLVM and Capstone. 41

4.3 Example of a feature vector extracted from the DDG in Figure 4.4. 43

4.4 Example of ground truth binarization applied. 46

4.5 Example of the use of OvR ensembles for classification. 46

5.1 Dataset distribution 50

5.2 Configuration of training algorithms used for experimentation . . 54

5.3 ADDG and CFG construction times for Seeds and Code Pervertor mutated instances. 59

5.4 Graphs' Average Construction Time vs CFG size for Reverter4 generated files. 60

Nomenclature

ADDG Annotated Data Dependency Graph

API Application Programming Interface

AUC Area Under the ROC Curve

CFG Control Flow Graph

CPU Central Processing Unit

DDG Data Dependency Graph

HMM Hidden Markov Models

InfoSec Information Security

IOM Instruction Occurrence Matrix

IoT Internet of Things

ISA Instruction Set Architecture

MAIL Malware Analysis Intermediate Language

MSDN Microsoft Documentation

NGVCK Next Generation Virus Construction Kit

ROC Receiver Operating Characteristics

RPME Real Permutation Engine

SMIT Symantec Malware Indexing Tree

SVM Support Vector Machines

List of Equations

5.1 Algorithms' and processes' execution time. 51

5.2 Models' accuracy score. 51

5.3 Machine learning algorithms' training time. 52

5.4 Classifiers' true positive rate 57

5.5 Classifiers' false positive rate 57

5.6 Time coefficient of variation by file size and CFG size ranges. 58

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Objective 5

1.3 Contributions 6

1.4 Document structure 6

2 Background **7**

2.1 Graph theory 7

2.2 Control Flow Graphs 9

2.3 Data Dependency Graphs 12

2.4 Malware 13

2.5 Traditional malware detection techniques 14

2.5.1 Anomaly-Based Detection 14

2.5.2 Signature-Based Detection 14

2.6 Code obfuscation techniques used to avoid detection 15

2.7 Metamorphic malware 16

2.7.1 Code morphing techniques used in metamorphic malware . 17

3 Literature Review **22**

3.1 Detection based on statistical models of classification 22

3.2 Identification using code signature matching 26
3.3 Final considerations 32
4 Methods proposal 35
4.1 Feature vectors extraction 36
4.1.1 Data Dependency Graph construction 37
4.1.2 Annotated Data Dependency Graph construction 40
4.1.3 Feature vectors construction 42
4.2 Index induction 43
4.2.1 Single-model multi-class classification 44
4.2.2 One-vs-Rest for multiple-classifiers multi-class classification 45
4.3 Suspicious instance classification 47
4.4 Chapter's final considerations 47
5 Experimentation 48
5.1 Experimental setup 48
5.1.1 Dataset construction 49
5.1.2 Dataset indexing 51
5.2 Experimental results 58
5.2.1 Dataset construction results 58
5.2.2 Dataset indexing results 60
5.2.3 Comparison with tier-one commercial antiviruses 73
6 Final thoughts and future works 76
Bibliography 78

Chapter 1

Introduction

In the past few decades, the Information Security (*InfoSec*) worldwide panorama has been characterized by the rising creation and spread of information system threats. The rapid rate which interconnectivity and online activity evolves, with around 2.5 quintillion bytes of data produced everyday [1], and the steady increase in the number of connected devices due to the upswing of technologies, as the Internet of Things (IoT)[2] and wearable devices [3], has increased vulnerabilities and the chances of attacks, as well as the challenge of keeping systems secure.

Security reports continually state worrying information about threats. According to Symantec's 2016 Internet Threat Report [4] in 2015 the number of zero-day vulnerabilities increased 125 percent from 2014. On average, a new zero-day vulnerability was found every week in 2015. There was more malware found from 2014 to 2015 than in the previous 10 years combined [5]. ENISA's 2018 Threat Landscape Report [6] stated that in 2017 anti-virus vendors like Avira [7] detected more than 4 million samples per day and more than 700 million samples in the first quarter of 2017. McAfee Labs 2017 Threat Report [8] points out a record of 57.6 million new unique malwares and a total of more than 750 million samples found only in the third quarter of 2017 (Figure 1.1).

The existence of such critical landscape is mostly politically [9] and financially motivated [10]. Cyber-crime is incentivized by a commercial industry with revenues of several million dollars a year[11]. Malicious users (hackers) seek undetectable attacks that stay hidden as long as possible, allowing hackers to perform illegal and profit rewarded activities like: *a)* stealing and selling login details [12]; *b)* pay-per-click fraud [13]; *c)* social media spam [14]; *d)* premium-rate SMS ; *e)* banking fraud [15] and *f)* gaining distributed computational power from thousands or millions of terminals (ex. *botnets*) [16].

To support these malicious activities, endpoint binary executable programs are employed, meant to be executed on a terminals' hosted operating system. Such programs are known as malicious software or malware, and according to

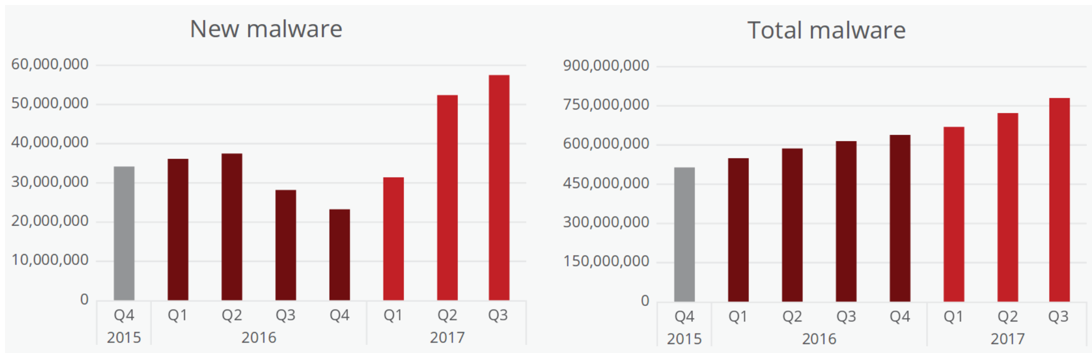

Figure 1.1: Quantity of malware samples found from the fourth quarter of 2015 to the third quarter of 2017
Source: McAfee Labs Threat Report 2017 [8]

their behavior and/or purpose can be non-exhaustively classified into [17]: *a)* computer virus; *b)* worm; *c)* trojan horse; *d)* spyware or *e)* ransomware. Some reports [18] [6] have stated that, in 2016 and 2017, malware was the top threat, surpassing other historically important threats like: *a)* web based attacks; *b)* phishing ; *c)* spam , *d)* identity theft and *e)* cyber-espionage.

In general, systems created for malware detection have presented better results by implementing signature matching [19]. That is, once a code is analyzed and labeled as malicious, a signature is created from its payload and integrated into a reference database, which is later used to detect instances of the same malware through comparisons. For example, one technique that is broadly used to construct such signatures consists of applying a checksum algorithm (MD5, SHA-256, SHA-512) to a malicious binary code's body to construct a hash that uniquely represents such instance. In a general signature-based detection system, such hashes are stored and later compared with the hashes of suspicious samples by exact or pattern matching, and are then labeled as malicious or not.

Although signatures have a low false positive rate, the number of malware samples covered by each signature is also low, typically one [20]. If a malware's source code is changed and recompiled, the signature of the obtained binary is likely to be different than the signature of the first original binary, which can also be obtained by compiling the same source code with different compilers given that each compiler employs different optimization methods.

When the signature of a suspicious instance is not found in reference databases, a common detection alternative is a set of heuristics. Some drawbacks of using heuristics are: *a)* they require a significant amount of resources and high processing capabilities; *b)* thwarting sandboxing analysis is possible with minor efforts [21]; *c)* given the increasingly use of protections against code stealing, included mostly in proprietary software, heuristics based on characteristics of the code or

its behavior are prone to fail and, in many cases, present false positive rates above 10% due to the difficulty of discriminating between malicious and legitimate traits based on its behavior [22] [23] [24].

In addition, to make malware even more difficult to detect, hackers have invented techniques that exploit the deficiencies presented by signature matching and heuristics. One such technique is code obfuscation, which generates new instances of malware with previously unseen patterns , that is, the probability that such new instance's signatures and patterns (structure and behavior) will be present in anti-virus databases are reduced.

Metamorphism is one technique employed for code obfuscation, which creates new instances of malware by changing the code of existing malware instances without altering its original functions. A set of operations can be applied to code to alter its structure without affecting its execution capabilities and original purpose. Several tools, like: *a)* Next Generation Virus Construction Kit (NGVCK) [1]; *b)* Code Pervertor and *c)* Revert4 [2]; automate such processes and generates thousands of different samples in a matter of minutes, increasing the diversity of a malware family and reducing the capacity of detectors to match new malware instances from the knowledge in databases, therefore, impacting detection.

This study addresses the challenges of malware identification resistant to code metamorphism. The main goal was to create a method that uses semantic signatures as structures that broadly represent instances of malware families to cope with the complexity of graph matching in datasets with thousands of instances. The methods in this work constitute a practical approach that can be implemented for large-scale endpoint malware detection.

1.1 Motivation

As Cohen stated [25], the creation of general definitive methods for malware detection is an undecidable problem. The exponential growth and sophistication of information and communication technologies has broadened the sensitivity complexity of systems. Therefore, it is easy for malicious organizations to find more targets and vulnerabilities to exploit. This scenario is confirmed by recent security reports, showing a significant increase in attacks and threats, as well as generation of new unique malware instances that have not been seen before [26] [8] [6].

Along with this panorama, malicious hackers and malware creators have emerged with advanced contrivances and intentions to disturb mechanisms employed by anti-malware suites. Many reports about information security [11]

[1] Publicly available at: http://vxheaven.org/vx.php?id=tn02
[2] Publicly avilable at: http://z0mbie.daemonlab.org

[26] [27] have addressed the impact of malware detection and system performance associated with the use of code obfuscation techniques used to change the structures and patterns present in viruses to avoid detection. Automatic code mutation, code permutation, and metamorphism are very effective techniques for code obfuscation that create several samples from previous instances, with high randomization in code structures and patterns. Such level of diversity, which can be obtained in an automated manner from a single malware at exponential rates, reduces the complexity of effective malware generation and makes the creation of accurate malicious pattern identification models more difficult.

Also, Symantec's 2017 Internet Security Threat Report [26] recognizes the inefficiency of current malware detection tools to cope with this critical landscape. There is an evident need for research and development of more accurate and better techniques for malware detection..

Concerning that issue, several studies have aimed to create malware identification and detection methods that are resilient to code morphing techniques and that have low impacts on time and resource consumption. Some usual approaches are: *a)* statistical modeling of morphing engine creation patterns [28] [29]; *b)* analysis of opcode instruction sequence occurrence distributions [30] [31]; *c)* statistical scanners made from the combination of feature ranking methods on opcodes n-grams sets [32] [33]; and *d)* analysis of intermediate semantic representations of codes such as pseudocodes [34], control flow graphs [35], system API call graphs [36] [37] and data dependency graphs [38] [39]. Usually such approaches are accompanied by classification models based on machine learning algorithms like: *a)* Decision Trees; *b)* Random Forest; *c)* Naive Bayes; *d)* Support Vector Machines (SVM) and *e)* Hidden Markov's Models.

However, such proposals do not completely solve the problem. Some limitations of such approaches are: *a)* necessary supervision of the creation and adjustment of models according to the malware family and/or creation kit of which instances are to be detected; *b)* high variance in the identification results; *c)* reduced capacity of generalization due to high variability in patterns and behaviors reached by metamorphism and the high appearance rate of new vulnerabilities and stealth techniques; and *d)* performance issues associated with complexity of comparison algorithms.

The use of Data Dependency Graphs stands out as a promissory approach since such semantical representations remain mostly invariant when the code is modified, without changing their original functions. Kim and Moon [39] examined how such code mutations confused anti-virus tools, however, remained ineffectual for Data Dependency Graphs (DDG). The graphs were similar even when the source code was significantly altered.

Given the resilience to metamorphism provided by DDGs, the adopted ap-

proach for metamorphic malware detection consists on partial matching of DDGs in suspicious instances with previously gathered DDGs of malwares, looking for pieces of such graphs that permits identify the presence of known malware patterns [39] [38]. The main drawback of this method is associated with performance issues since current algorithms for partial matching of graphs or subgraph isomorphisms are NP-Hard [40]. To address that problem, genetic algorithms and graph reductions techniques have been employed. However, such procedures are computationally expensive and, when executed on large bases with thousands of graphs, can be time consuming, therefore considered an impractical approach for real-time malware detection. Shrinkage operations applied to the structures, by omitting unnecessary parts to reduce execution time of the matching algorithm, results in the omission of representative information that could contribute to the accuracy of identification.

To address current issues of metamorphic malware detection, this study proposes a method for endpoint malware detection resilient to code metamorphism based on semantic-aware large-scale annotated data dependency graph matching. Through reverse engineering, a DDG is constructed for each function in the assembly code of the suspicious binary, such DDG is labeled with a tag that represent the semantic of the assembly instruction present in each node of the DDG. Then a vector is constructed with structural features of the graphs to conform a dataset of signatures used to create models that serve as indexes to retrieve instances presenting the same patterns on input suspicious binary, with practical performance.

1.2 Objective

The objective of this study is to create and evaluate a method for malware detection that is resilient to metamorphism, based on the comparison of annotated data dependency graphs of malware instances using indexing models created with machine learning algorithms.

To accomplish this main objective we proposed the following specific objectives as auxiliary elements to the final goal:

- Define a schema for executable binary representation based on the use of annotated data dependency graphs and the classification of instructions in its nodes as a semantic signature employed for pattern matching.

- Define and evaluate a graph dataset indexing approach based on the use of machine learning algorithms that provides a mechanism to leverage information about databases of annotated data dependency graphs to identify

if an instance was generated from mutations applied to a sample of that database.

1.3 Contributions

To fulfill our objectives the following contributions were made:

- A method for semantic aware annotated data dependency graphs extraction from executable code based on: *a*)assembly code instructions classification and *b*)data flow analysis.
- A new heuristic for graph matching based on the use of machine learning algorithms and structural features in annotated data dependency graphs.

The following scientific publications were originated from this research:

- L. M. Rojas, E. Souto, and G. Breves, "Detecção de malware metamorfico baseada na indexação de grafos de dependencia de dados", Awarded Honorable Mention in XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais: SBSEG 2017: Anais. SBC, 2017, pp. 264–277.

1.4 Document structure

The rest of this document is organized as follows: Chapter 2 presents definitions and explanations about the domains of: *a)* malware detection; *b)* anti-malware evasion and metamorphic malware; *c)* graph theory and *d)* control and data flow graphs; which are provided to better understand this work's proposal. Chapter 3 presents a state-of-the-art study in the field of metamorphic malware detection, to comprehend current limitations and development, as well as contributions of this work, given the current panorama. Chapter 4 presents the methods for metamorphic malware detection, as well as details about the methods' components. Chapter 5 consists of the experiments carried out to evaluate the proposed methods, which determined the validity and applicability of such methods in real world detection systems. Finally, Chapter 6 provides conclusions and future research needed to understand the limitations and contributions of the proposed methods.

Chapter 2

Background

This chapter presents a set of definitions and explanations necessary to understand the components of the methods presented in this study. Given that the methods for malware detection in this research are based on the use of graphs, this chapter begins with a set of definitions about graph theory, which are maintained throughout this document. Afterwards, explanations of Control Flow and Data Dependency Graphs to state the characteristics of such structures employed in this work are provided. Thereupon, characteristics of the malware detection techniques used and a critical view of its advantages and limitations are presented. Furthermore, techniques used by malware creators to avoid current mechanisms of detection are introduced and explained, further focusing on metamorphism as the main issue addressed in this research.

2.1 Graph theory

This section provides a theoretical background about the field of graph theory. Initially, it provides a definition of Graph, stated by Harris et.al [41] in the book: *Combinatorics and graph theory*, since it fits the conception and notion of the term used by the authors of this study:

Definition 1 *Graphs*
A graph consists of two finite sets, V and E. Each element of V is called a vertex (plural vertices). The elements of E, called edges, are unordered pairs of vertices. For instance, the set V might be $\{a, b, c, d, e, f, g, h\}$, and E might be $\{\{a, d\}, \{a, e\}, \{b, c\}, \{b, e\}, \{b, g\}, \{c, f\}, \{d, f\}, \{d, g\}, \{g, h\}\}$. Together, V and E are a graph G.

A more accurate definition of graph employed herein is Attributed Graph, however, both terms are used synonymously throughout this work:

Definition 2 *Attributed Graph*
An attributed graph is a six-tuple $G = (V,E,\alpha,\beta,L_v,L_e)$, *where* V *denotes a finite set of nodes,* $E \subseteq V \times V$ *is a finite set of edges,* $\alpha : V \rightarrow L_v$ *is a node labeling function,* $\beta : E \rightarrow L_e$ *is an edge labeling function,* L_v *is a set of node labels, and* L_e *is a set of edge labels.*

Since all the graph structures used herein are considered directed graphs, defined by Allen as [42]:

Definition 3 *Directed Graphs*
A directed graph, G*, can be denoted by* $G = (B,E)$ *where* B *is the set of nodes (blocks)* $\{b1,b2,...,b_n\}$ *in the graph and* E *is the set of directed edges* $\{(b_i,b_j)\ ,\ (b_k,b_l),...\}$*. Each directed edge is represented by an ordered pair* $(b_i,\ b_j)$ *of nodes (not necessarily distinct) which indicate that a directed edge goes from node* b_i *to node* b_j*.*

Also it is introduced the concept of subgraph as follows:

Definition 4 *Subgraph*
A subgraph $G_s = (V_s,E_s,\alpha_s,\beta_s,L_v,L_e)$ *of a graph* G*,* $G_s \subseteq G$ *is a six-tuple, where* $V_s \subseteq V$*,* $E_s \subseteq E \cap (V_s \times V_s)$*,* $\alpha_s(v) = \alpha_s(v) \forall v \in V_s$ *and* $\beta_s(e) = \beta(e) \forall e \in E_s$

Graphs are used as a representation of control and data flow in executable programs' code. Matching such structures determines similarities between two different programs. In general, two graphs G and G' can be compared by determining a mapping function u, which associates nodes and edges of G with nodes and edges of G' and vice versa. Some of the more frequently used mapping functions are graph isomorphism and subgraph isomorphism.

Graph isomorphism is defined as:

Definition 5 *Graph Isomorphism*
A bijective function $u : V \rightarrow V'$ *is a graph isomorphism from a graph* $G = (V,E,\alpha,\beta,L_v,L_e)$ *to a graph* $G = (V',E',\alpha',\beta',L_v,L_e)$ *if:* $\alpha(v) = \alpha'(u(v))\ \forall\ v \in V$*, for any edge* $e = (v1,v2) \in E$ *there exists an edge* $e' = (u(v_1),u(v_2)) \in E'$ *such that* $\beta(e) = \beta'(e')$ *and for any* $e' = (v_1',v_2') \in E'$ *exists an edge* $e = (u^{-1}(v_1'), u^{-1}(v_2')) \in E$ *such that* $\beta(e) = \beta'(e')$*.*

Intuitively speaking, two graphs are isomorphic if they are equal. For most applications, this mapping is much too strict. In practical scenarios, more relaxing concepts are introduced, one of which is the subgraph isomorphism. Similarly, by mapping one graph to another, a graph can be mapped onto parts of another graph. Such mapping can be defined as a subgraph isomorphism [43] :

Definition 6 *Subgraph Isomorphism*
An injective function $u : V \rightarrow V'$ *is a subgraph isomorphism from* G *to* G' *if there exists a subgraph* $G_s \subseteq G'$ *such that* u *is a graph isomorphism from* G *to* G_s*.*

Other graph matching mapping functions include maximum common subgraph and graph edit distance, however, are not used herein.

Since are used for Control Flow Graphs and Data Dependency Graphs, hereby is introduced the definitions for successor and predecessor functions, as follows:

Definition 7 *Successor and Predecessor functions*
Given a directed graph $G = (B,E)$*, a successor function* $\Gamma^l{}_G$ *maps* G *into* G *such that* $\Gamma^l{}_G(b_i) = \{b_j|(b_i,b_j) \in E\}$*. In the same way a predecessor function* $\Gamma^{-l}{}_G$ *is defined as the inverse of the successor function such that* $\Gamma^{-l}{}_G(b_j) = \{b_i|(b_j,b_i) \in E\}$*.*

The sets provided by the successor and predecessor functions are called the immediate successors and predecessors, respectively. Both sets can be empty. Herein, successor and predecessor functions are used for analysis and construction of Control Flow and Data Dependency Graphs.

2.2 Control Flow Graphs

Any static global analysis of the expression and data relationships in a program require knowledge of its control flow [42]. A Control Flow Graph (CFG) represents the relations of control between the parts of a program. To express such relations, the program is divided into groups of indivisible units of code sentences, called basic blocks. With the aim of denoting a Control Flow Graph, the following set of definitions are provided [44].

Initially, the definition for the Control Flow Graph's basic blocks are presented as follows:

Definition 8 *Basic Block*
A basic block is a maximal sequence of instructions with a single entry and a single exit point. There is no instruction after the first instruction that is the target instruction of a jump instruction, and only the last instruction can jump to a different block.

Instructions that can start a basic block include the followings:

- The first code's instruction,

- Target of a branch or function call, and
- A fall through instruction, i.e. an instruction following a branch, a function call or a return instruction.

Instructions that can end a basic block includes :

- Conditional or unconditional branches,
- Function calls,
- Return instructions

In a CFG the relations of control between basic blocks are expressed through control flow edges:

Definition 9 *Control Flow Edge*
A control flow edge $e = (a, b)$ is a directed edge that expresses relation of control between blocks a and b. That is, block a contains an instruction that calls for the execution of block b.

Finally, the definition of Control Flow Graph is:

Definition 10 *Control Flow Graph*
A Control Flow Graph (CFG) is a directed graph $G = (V, E, \alpha, \beta, L_v, L_e)$, where V is the set of basic blocks, E is the set of control flow edges, L_v is the group of instructions' sets inside each basic block and $\alpha(v)\ \forall\ v \in V$ determines the set of instruction from L_v belonging into v.

Figure 2.1 shows a sample CFG constructed from the assembly code in Table 2.1, extracted from the entry point function of program *cat* present in any *linux* distribution.

Table 2.1: Assembly code sample

Address	Assembly code
0x00405268	mov rdx, rbx
0x0040526b	mov rsi, r12
0x0040526e	mov edi, r13d
0x00405276	test rax, rax
0x00405279	mov rbp, rax
0x0040527c	jns 0x4052a0
0x00405283	mov eax, dword [rax]
0x00405285	cmp eax, 4
0x00405288	je 0x405268
0x0040528a	cmp eax, 0x16
0x0040528d	jne 0x4052a0
0x0040528f	cmp rbx, 0x7fffe000
0x00405296	jbe 0x4052a0
0x00405298	mov ebx, 0x7fffe000
0x0040529d	jmp 0x405268
0x0040529f	nop
0x004052a0	add rsp, 8
0x004052a4	mov rax, rbp
0x004052a7	pop rbx
0x004052a8	pop rbp
0x004052a9	pop r12
0x004052ab	pop r13

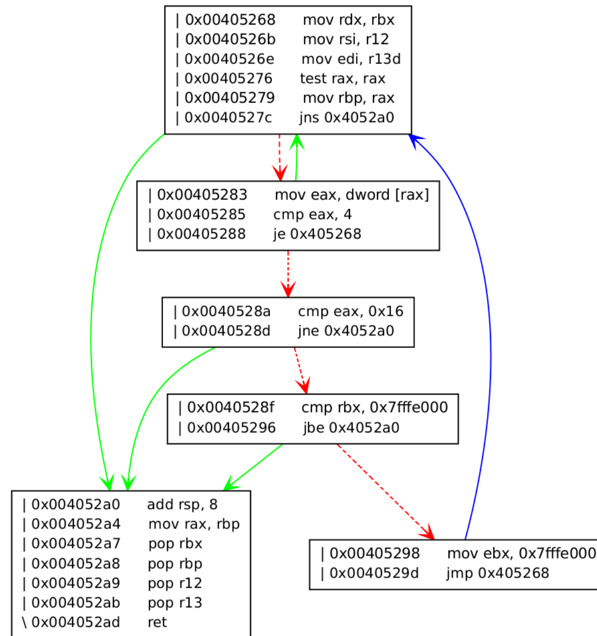

Figure 2.1: Sample of CFG extracted from assembly code in Table 2.1

Every block in the CFG ends up with a control instruction that changes execution flow (ex. jns, je, jmp). Blocks with a last instruction of conditional jump have two edges outgoing, with a different line pattern to denote possible outcomes of execution flow according to the condition asserted.

2.3 Data Dependency Graphs

A Dependency Graph of a program represents the dependence relations among its parts so that, if a program is divided into single units, represented by $U = \{u_1, u_2, ..., u_n\}$, there is a dependence between any pair of units (u_i, u_j) if one of them has no meaning in the program without the existence of the other [45].

A Data Dependency Graph (DDG) is a dependency graph that expresses relations between parts of code based on the use of data made by such parts. In a DDG there is a node for each instruction that consumes and/or modifies data and an outgoing edge from the node that consumes entering the node that modifies.

The edges used to express such relations are denominated Dependency Edges. In this work the definition of Dependency Edge is adapted from the definition by Kim and Moon [39], presented as follows:

Definition 11 *There is a dependency edge from vertex v_1 to vertex v_2 if there is a certain variable X such that X is used on v_1 while the value of X is assigned on v_2.*

A formal definition for Data Dependency Graph is provided as follows:

Definition 12 *A Data Dependency Graph (DDG) of a program is a directed graph $G = (V, E)$, where V represents the set of instructions that handles data and E is the set of dependency edges.*

DDGs have been used for: *a)* compiler optimizations [45], *b)* software plagiarism detection [46] and *c)* malware detection [38] [39]. In this work, Data Dependency Graphs are used to represent data flow and dependency of assembly code instructions based on its use of registers and memory.

The following definition for Annotated Data Dependency Graph (ADDG) is used:

Definition 13 *An Annotated Data Dependency Graph (ADDG) of a program is a $DDG = (V, E)$ in which every vertex $v_i \in V$, v_i is assigned a label.*

Figure 2.2 shows an example of a DDG induced from the pseudo-code expressed in Algorithm 1. Every vertex on that DDG represents a line of code in Algorithm 1, such correspondence is presented with a number on each vertex according to the line number such vertex represents. The edges connecting vertexes 4 with 2 and 4 with 3 exist because x_1 is declared in 2, x_2 is declared in 3 and both variables are used in 4, this same reasoning explains the edge connecting node 4 with node 5. Similarly, in Figure 2.3 an example of an ADDG extracted from Pseudo-code in 1 is presented.

Algorithm 1: Pseudocode of simple example function

```
1 function SampleFunction ();
2 x1 ← 1 ;
3 x2 ← 3 ;
4 x3 ← x1 + x2 ;
5 print(x3) ;
```

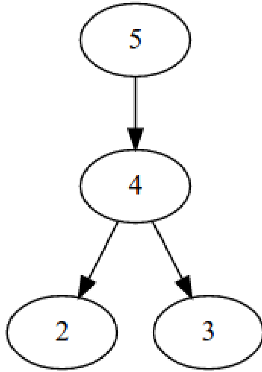

Figure 2.2: Sample DDG extracted from code in 1

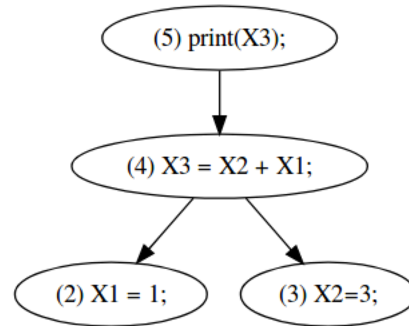

Figure 2.3: Example of an ADDG extracted from pseudocode 1

2.4 Malware

Malware is a collective noun which denotes programs that have a malicious intent – the neologism standing for mal-icious soft-ware [47]. According to Gary MacGrow and Greg Morriset's definition, malicious code is any code created with the intention to cause harm or subvert the functions of information systems [48].

A malware carries out activities such as setting up back doors for bots, up keyboard loggers, prepend on benign programs to steal personal information, consume system resources, and allow unauthorized access to compromised systems. Specifically, malware usually denotes hostile, intrusive, or simply annoying software that is programmed to gather sensitive information, gain access to private systems, or disrupt the legitimate computer operations in any other way. Also, malware comes in various forms such as virus, worm, Trojan horses, and other programs created with malicious intent [49].

2.5 Traditional malware detection techniques

Although anti-malware suites differ in the implementation of their identification techniques, they tend to incorporate the same detection mechanisms. These mechanisms can be classified as signature matching and anomaly-based detection. Furthermore, there are hybrids that incorporate the best of both approaches [50] [51].

2.5.1 Anomaly-Based Detection

Anomaly-based detection systems are designed to detect any kind of misusing that falls out of the ordinary activity of a computer system by monitoring activities and classifying them as either normal or anomalous [52]. It looks for indicators that position the input file as suspicious, such as: *a)* very big file sizes; *b)* large debug sections; *c)* entry-point code redirection; and *d)* suspicious kernel operations. This behavior is incorporated in detection suites through heuristics that express conditions that label an instance as malign. In general, heuristics are founded in analysis of: *a)* network traffic; *b)* system registers used; *c)* program logs; and *d)* structural characteristics of code body.

Some studies have addressed high rates of false positives presented by detection mechanisms based on heuristics [53] [31]. In many cases it is hard to distinguish between malicious or benign, based on the behavior and trails left by systems [31].

Some of the drawbacks presented by anomaly-based detection are:

a) Some heuristics are based on inspecting behavior and code body characteristics and look for patterns frequently seen in malware. However, such patterns start to be seen in legitimate software as code presents encrypted sections and other reverse engineering deceiving techniques to protect it against copy [23] and massive generation of network traffic (usually employed by real-time applications [54]).

b) Usually suspicious programs are executed on an emulated environment in order to protect the host system from malware's activities. However, some mechanisms can be integrated in malware to detect when it is being virtualized, which consequently makes the malware behave normally in such scenarios [55].

c) c) Complexity and computational costs of execution lead to slower performance and high resource consumption.

2.5.2 Signature-Based Detection

Signature-based detection consists of searching the same byte sequences, previously seen in malware, from the suspicious instance's code body under analysis.

Table 2.2: Example of hash signature extracted from code

Opcode	Assembly Code
C7060F000055	mov [esi], 0x5500000F
C746048BEC5151	mov[esi+0004],0x5151EC8B
Signature: C7060F000055C746048BEC5151	

Once a malicious file is detected by software security analysts or automatic tools, other copies of such instance, in which modifications have not been applied, can be recognized as well by comparing both instances' signatures. Although less proactive than desired, signature-based malware scanning is still the dominant approach to identify malware samples in the wild due to its extremely low false positive rate, that is, the probability of mistaking a good-ware program for a malware program is very low [56].

Most signatures used in existing signature-based malware scanners are hashes of malicious binaries. Although hash signatures have a low false positive rate, the number of malware samples covered by each hash signature is also low, typically one. As a result, databases of signatures employed by anti-viruses grow at the same rate that new instances appear in the wild [20]. Table 2.2 presents an example of a hash signature that would be used for scanners while analyzing files looking for signatures matchings.

Usually, antiviruses incorporate more sophisticated mechanisms to enhance the speed of signature matching, since with every new malware discovered the quantity of signatures in the bases increases. An example is the use of wildcard methods that perform comparisons skipping bytes or byte ranges, thus improving speed by reducing the quantity of comparisons made in one second [57].

2.6 Code obfuscation techniques used to avoid detection

Malware creators develops sophisticated code obfuscation and hiding mechanisms that permits replication of malware to new instances with original functionalities but with different signature and execution flow. Techniques to avoid signature based detection through modifications of the malware body have evolved to more sophisticated methods, as malware analysts have found efficient methods to deal with obfuscated code. A major concern is that obfuscation methods have been automated and incorporated in malware creation kits, which allows massive generation of new malware even by non-technical users [58]. Some ways to create obfuscated malware are:

a) **Packing** applied by compressing the malware with a specified packing method that should be unknown by the anti-virus. A packed instance must be unpacked in order to perform detection, however, unpacking a program is only possible with a specific unpacker. Most detection tools employ entropy analysis to address packed malware [59] [60].

b) **Encryption** consists of encrypting the malware payload for distribution and decrypts it on runtime. An encrypted virus generally contains two elements : the encrypted virus code and a small decryption engine that decrypts the virus payload at runtime. Since code is decrypted on execution, applying signature matching on the code in memory can be effective to detect encrypted malware with few limitations [61].

c) **Oligomorphism** also employs encryption, however, a set of encryption engines are used and not just one. The decryptor is changed in successive generations by modifying the body of the decryptor code. This way the obfuscation and hiding presents high variability. The decrypted code resides in memory at runtime and can be detected with a signature based approach [62].

d) **Polymorphism** is applied through encryption, but the cryptographic generator is modified on each run of the program using obfuscation techniques. This results in an immense quantity of possible variants of encryption engines. Polymorphic malware is also detected by signature matching when the decrypted code is loaded into memory [61].

e) **Metamorphism** consists of applying operations to malware's code body to modify it without compromising the original functions. Such operations can be: *a)* code blocks transposition, *b)* variable renaming, *c)* dead code insertion, *d)* code replacement, and others. These operations can be applied to the malware by an external permutation tool or included as a function so it auto-mutates on each replication. Empirical studies have demonstrated how difficult it is for commercial antiviruses to detect obfuscated code and, more specifically, metamorphic malware [39] [63] [38]. More details on this technique are presented in the next section.

2.7 Metamorphic malware

Malware creators have realized that using encryption to generate obfuscated variants that avoid signature based detection, presents inefficiencies. The creation of encryption schemes with high entropy, which are less likely to be detected, can be highly resource consuming. Virus writers used to waste weeks or months creating a new polymorphic virus that often did not have a chance to appear in the wild due to its bugs. While, a researcher might be able to detect such a virus in a few minutes or days. Most polymorphic viruses decrypt themselves to a single

constant virus body in memory, while metamorphic malware do not. Therefore, detection of the virus code in memory needs to be algorithmic because the virus body does not become constant, even in the memory [64].

Moreover, the same obfuscation performed on the encryption engines can be applied to the malware code body, which results in modifications of the hashes of the programs without losing the original functionalities. This way encryption was abandoned, and metamorphism emerged. The next subsection provides more details about techniques employed to create metamorphic malware, as well as more details about techniques used to avoid signature based detection by applying modifications to executable code's opcodes.

2.7.1 Code morphing techniques used in metamorphic malware

To create metamorphic malware, morphing techniques can be applied that modify the structure of code body at an instruction level or modify the control flow. Table 2.3 presents examples of techniques that can be applied for code mutation at a high level code representation.

Table 2.3: Examples of techniques that can be applied to alter the code body without affecting its original functionalities

(a) Original code	(b) Dead code insertion	(c) Variable renaming

dim n, p, i
n = 5
p = 1
for i = 1 to n do
p = p * i
end for
	dim n, p, i
	n = 5
	p = 1
	for i = 1 to n do
	if i >0 then
	p = p * i
	end if
	end for
		dim a, b, c
		a = 5
		b = 1
		for c = 1 to a do
		b = b * c
		end for

(d) Code replacement	(e) Statement reordering	

dim n, p, i
n = 5
p = n / 5
for i = 1 to n do
p = p * i
end for
	dim i, p
	p = 1
	dim n
	n = 5
	for i = 1 to n do
	p = i * p
	end for	

By combining the techniques presented in Table 2.3, highly variable codes can be generated. Some mutations frequently found in metamorphic malware

creations are:

a) **Instruction reordering** , which consists of transposing instructions that do not depend on the output of previous instructions. When instructions are reordered, signatures involving the instructions can be broken, while code execution is unaffected. [65]. This technique can also be achieved by applying commutative and/or associative operators as presented in Table 2.4.

To avoid affecting original execution order, control flow instructions are used (e.x *jump*) as shown in Figure 2.5.

Table 2.4: Examples of instruction reordering

a = 10;
b = 20;
x = a * b;
	can be changed to:		a = 10;
			b = 20;
			x = b * a

Original machine and assembly codes			
c7 45 f4 0a 00 00 00	movl	[rbp-0xc], 0xa ;	a = 10
c7 45 f8 14 00 00 00	movl	[rbp-0x8], 0x14 ;	b = 20
8b 45 f4	mov	eax, [rbp-0xc] ;	
0f af 45 f8	imul	eax, [rbp-0x8] ;	a * b
89 45 fc	mov	[rbp-0x4], eax ;	x = a * b
Mutated machine and assembly codes			
c7 45 f4 0a 00 00 00	movl	[rbp-0xc], 0xa ;	a = 10
c7 45 f8 14 00 00 00	movl	[rbp-0x8], 0x14 ;	b = 20
8b 45 f8	mov	eax, [rbp-0x8] ;	(reordered)
0f af 45 f4	imul	eax, [rbp-0xc] ;	b * a (reordered)
89 45 fc	mov	[rbp-0x4], eax ;	x = b * a

b) **Dead code insertion** is implemented by randomly inserting code that is either not executed or, if executed, has no effect on the results of the original program. Usually, this is implemented by the insertion of *NOP* instructions which does not affect the CPU state and some antiviruses are eventually trained to ignore flat instructions like those. Other examples include *MOV eax, eax* , *ADD eax, 0* and *SUB eax, 0*. Dead code insertion can be very effective to change statistical patterns of malware families [66] [44]. By applying dead code insertion, infinite variants can be obtained from a single instance. However, as an instruction's address is changed it can affect cross referencing, making code inexecutable or changing its original function. To avoid damaging cross references, dead code should be inserted in unused spaces [67].

c) **Register renaming** consists of using different registers for each new generated variant. Therefore, the signature changes and makes detection based on register-using patterns identification difficult. *Win95/Regswap* and *Win32/Evol* viruses implements metamorphosis via register usage exchange. Any part of the virus body use different registers from one variant to another but the original intention remains. Tables 2.5 and 2.6 exemplify the application of this technique

on the viruses Win95/Regswap and Win32/Evol both found in the wild in 1995 and 2000 respectively.

Table 2.5: Two generations of Win95/Regswap.

5A	pop	edx
BF04000000	mov	edi,0004h
8BF5	mov	esi,ebp
B80C000000	mov	eax,000Ch
81C288000000	add	edx,0088h
8B1A	mov	ebx,[edx]
899C8618110000	mov	[esi+eax*4+00001118],ebx
58	pop	eax
BB04000000	mov	ebx,0004h
8BD5	mov	edx,ebp
BF0C000000	mov	edi,000Ch
81C088000000	add	eax,0088h
8B30	mov	esi,[eax]
89B4BA18110000	mov	[edx+edi*4+00001118],esi

d) **Subroutine permutation** consists of reordering the subroutines of a program, since the order in which subroutines appear in code is not critical to its execution. The order of subroutines is different from one generation to another, which leads to *n!* different virus generations, where n is the number of subroutines. The *BadBoy DOS* virus family had eight subroutines, 8! = 40,320 different generations and *Win32/Ghost* (discovered in May 2000) had 10 functions, 10! = 3,628,800 combinations [64]. Subroutine permutation is employed in The Real Permutation Engine (RPME), which was used to generate the highly metaphoric families of malwares like Metaphor and Zmist [67] [68].

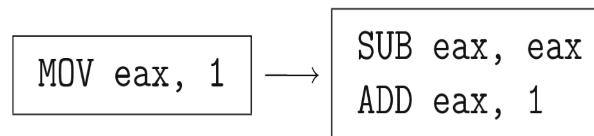

Figure 2.4: Example of equivalent instructions substitution

e) **Equivalent instruction replacement**: The instruction set for modern processors have numerous equivalent instructions (or groups of instructions). For example, *MOV eax, 0*, is equivalent to *SUB eax, eax* and *XOR eax, eax*. Figure 2.4 illustrates an example where a single instruction is equivalent to a sequence of instructions.

f) **Code reordering** consists of inserting a conditional or unconditional branch instruction after a block of instructions. Blocks defined by such branching instructions can then be permuted to change the control flow. Figure 2.5

Table 2.6: Metamorphic variants of the virus Win32/Evol.

a) Generation A:		
C7060F000055	mov	dword ptr [esi],5500000Fh
C746048BEC5151	mov	dword ptr [esi+0004],5151EC8Bh
b) Generation B:		
BF0F000055	mov	edi,5500000Fh
893E	mov	[esi],edi
5F	pop	edi
52	push	edx
B640	mov	dh,40
BA8BEC5151	mov	edx,5151EC8Bh
53	push	ebx
8BDA	mov	ebx,edx
895E04	mov	[esi+0004],ebx
c) Generation C:		
BB0F000055	mov	ebx,5500000Fh
891E	mov	[esi],ebx
5B	pop	ebx
51	push	ecx
B9CB00C05F	mov	ecx,5FC000CBh
81C1C0EB91F1	add	ecx,F191EBC0h ; ecx=5151EC8Bh
894E04	mov [esi+0004],ecx	

illustrates the 'spaghetti code' that can easily be generated by this approach. At this point, consecutive instructions are permutated and linked together by unconditional jumps. The reordering of instructions does not modify the order in which they are executed, but breaks signatures that rely on the adjacency of certain sets of instructions.

A more advanced technique for metamorphism proposes an evolutionary approach through genetic frameworks [69] [47]. To drive evolution towards the creation of malicious applications that are hard to detect, an individual is awarded a progressively higher fitness value if they satisfy a series of predefined requisites. During this process, it is possible to obtain non-valid programs, unable to compile or not executed correctly. Moreover, programs that are compiled and executed successfully could lose the malware characteristics, becoming harmless software applications.

Interpreted languages can have poly/metamorphic behavior but rely on a static compiled interpreter to execute, which usually applies many optimization strategies not possible to control from the high level code, hence a large portion of the 'run-time signature' is not mutable. Compiled low level languages are preferred over high level languages when it comes to metamorphic malware creation, since they offer enough computational flexibility. That said, herein, we only address metamorphic malware in the form of mutated endpoint executable

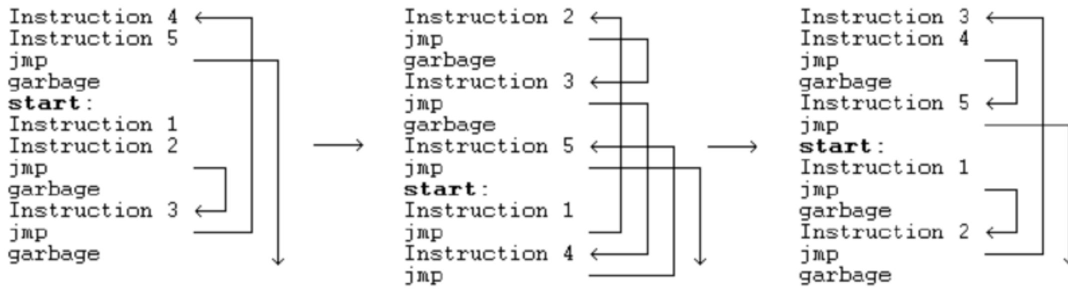

Figure 2.5: Representation of code reordering and dead code insertion.

binaries.

Currently, metamorphic malware detection is an issue for the InfoSec community, which is corroborated by the constant emergence of new studies on the topic and the potential threat acknowledged by recent annual security reports [37] [70] [26] [71]. The current knowledge about malware creation and malware detection techniques, the proliferation of malware creation kits accessible on the Internet [1], and the organization of cybercrime moved by financial motives, indicates that most metamorphic malware may exist without being detected.

[1]Tools publicly available on: http://vxheaven.org/ and http://z0mbie.daemonlab.org/

Chapter 3

Literature Review

Several strategies have been proposed to improve malware detection resilient to metamorphism. This chapter presents a revision of previous research efforts about metamorphic malware detection. The approaches discussed in this section are broadly grouped into: *a)* pattern recognition and *b)* semantic structures matching.

The first group consists of approaches that employ statistical models of classification, constructed by previously gathered knowledge about malware families and virus construction kits. These models decide if a suspicious instance can be classified into a set of defined classes of interest, such as malware or benign, or if instances were generated with a specific engine of metamorphism or construction kit. The second group consists of approaches that employ algorithms to compare semantic structures extracted from known malware and instances under analysis to decide if a suspicious instance is or was generated from a known malware, based on the similarity of its representative structures.

3.1 Detection based on statistical models of classification

The approaches presented in this section provide solutions to metamorphic malware identification based on statistical models that represent the patterns found in *a)* malign code families, *b)* benign code, *c)* metamorphic engines and *d)* virus creation kits. Typically, in such methods, machine learning algorithms [72] are employed to construct binary classifiers that discriminate into: *a)* malware and benign, *b)* does or does not belong to a specific family and *c)* was or was not generated with a specific virus construction or metamorphic engine.

Canfora et al. [32] proposes a detection technique that relies on the assumption that a common side effect metamorphic engines is the dissemination of

repeated instructions in the body of malwares, which permits elaborate identification models based on statistical patterns of instructions and opcode distributions in a malware's code body. Through comparative analysis, Canfora et al. concluded that the distribution of opcodes in trusted programs are different from the distribution of those same opcodes in malware programs. Based on such conclusion, the authors proposed a data structure that associates each opcode with the number of unique instructions that it contains, with at least 2 occurrences in the code, called an Instruction Occurrence Matrix (IOM). Such IOM structure, which is shown in Table 3.1, serves as a feature vector that characterizes a malware's code body based on the frequency of appearance of its instructions. IOMs are employed to train classifiers and, in tandem with such models, to classify a suspicious code as malware or benign. Five decision trees: J48, Ladtree, Nbtree, Random Forest, and Reptree are built using Weka [73], where the intermediate nodes contain the conditions applied to one of the attributes selected for classification and the end nodes represent the predicted class. The classification analysis showed that the method can correctly classify both malware and non-malware with an accuracy of 94% and a false positive rate of 3% for non-malware and generated malwares. The approach is equally effective for non-metamorphic malware detection, however, the accuracy of the method decreases when benign codes are added to the malware.

Table 3.1: Example of an IOM structure as employed by Canfora et. al

8086 Op-code	Number of unique instructions with more than one occurrence
AAA	0
AAD	5
AAM	3
AAS	2
ADC	0
ADD	15
...	...
INT	2
...	...

The use of Hidden Markov Models [74] for metamorphic malware detection is one of the first and most abided approaches in the topic [75] [76] [77] [78]. In such studies, opcode sequences in metamorphic malware families are used to construct models that express the underlying probabilities of specific opcodes that appear after other given opcode. Afterwards, such models are employed to determine similarities among previously seen and under analysis instances based

on whether they presents the same probabilistic patterns, using algorithms like: *a)* Forward algorithm [79], *b)* Viterbi Algorithm [80] and *c)* Baum-Welch Re-estimations [81]. Such models determine if an instance is part of a specific family of malware but are bound to the mutator or metamorphic engine that generated the instances. Studies have been carried out to create metamorphic engines to defeat Hidden Markov Models (HMM) based approaches, obtaining exact and promissory results. For example, Desai [82] and Tamboli et al [67] demonstrated such technique was ineffective when more than 20% of dead and benign like codes were inserted into the program.

Singh et al. [83] proposed an approach to classify binaries into benign or malware by applying Support Vector Machines (SVM) [84].In that work, they studied three similarity scoring techniques that can be used to discriminate between malwares and benign, namely: *a)* Hidden Markov Models, *b)* Simple Substitution Distance, and *c)* Opcode Graphs similarity. For each technique, they carefully analyzed the degree of malware mutation needed to break the score, using the area under the ROC curve (AUC) as a measure of success. The authors trained binary classification models, which discriminated between malware and benign, employing SVM to obtain a hyperplane of maximal separation for the previously mentioned scores. Through experimentation with instances created by the Next Generation Virus Construction Kit [1] (NGVCK) and metamorphic Malicia Project's [2] malware families, the authors concluded that more accuracy was obtained by combining the scores with SVM, rather than individually using each technique. Such method presented an average accuracy of 87% and a false positive rate of 4%.

Table 3.2: Example of report extracted from notepad.exe as employed by Choudhary and Vidyarthi

Process name	**PID**	**Operation**	**Path**	**Detail**
notepad.exe	616	CreateFile	C:/Windows/System32	Desired Access: Generic
notepad.exe	616	CloseFile	C:/Windows/System32/notepad.exe	
notepad.exe	616	LoadImage	C:/Windows/System32/notepad.exe	Image Base: 0xc0000
notepad.exe	616	LoadImage	C:/Windows/System32/ntdll.dll	Image Base: 0x77240000
notepad.exe	616	CreateFile	C:/Windows/Prefetch	Access: Read
notepad.exe	616	QueryStandardInfo	C:/Windows/Prefetch	AllocationSize: 45,056

Choudhary and Vidyarthi [70] studied the behavior of metamorphic malware instances by execution in virtual environments and by analyzing the reports generated with a text mining-based technique. In Table 3.2 an example of the reports obtained is presented. Such reports contain information regarding the effects of

[1]Publicly available on: http://vxheaven.org/vx.php?id=tn02
[2]Publicly available on: http://malicia-project.com

the program execution on the host system and the state of program during execution, such as: *a)* operations performed by the main and children processes, *b)* location where operations are performed and *c)* parameters passed during such operations. Then, the Information Gain [85] was calculated for each word present in reports, based on their frequency of occurrence in benign and malware. Furthermore, by using Information Gain as a measure of word representativity for malware or benign, a model for binary classification was trained using Support Vector Machines. Experimentation demonstrated effectiveness in separating malware from benign files with an accuracy of 97.8%. Using 10-fold cross validation, 188 (91 malwares + 97 benign) samples were classified with four instances misclassified as benign. However, this method is ineffective for techniques like anti-debugging, anti-vm, and anti-emulation, which avoid dynamic malware detection by providing malware instances that are able to detect if they are being analyzed and avoid executing their malicious activity in such conditions [86] [55].

Vinod et al. [49] proposed a statistical method for malware detection that does not rely on similarity matching of signatures. For analysis, a program is disassembled using IDA-Pro [87] and the code obtained is parsed to extract the frequency of occurrence of mnemonics' n-grams sets for n values 1 to 5. Also, the instructions' opcodes frequency of occurrence are calculated. With the gathered information, feature vectors are constructed. Redundant features are eliminated using: *a)* Class-wise Document Frequency (CDF) [88], *b)* Scatter Criterion [89] and *d)* Principal Component Analysis (PCA) [90]. With the resulting features, malware or benign binary classifiers are trained by: *a)* K-NN's IBk [91], *b)* J48 Decision Tree [92], *c)* AdaBoost [93], *d)* Random Forest [94] and *e)* Sequential Minimal Optimization (SMO) [95]. For identification of benign files, the best results were obtained using IBk with a 0.98 true negative rate and 0.02 false negative rate. For malware identification, the best model presented a 0.96 true positive rate and a 0.09 false positive rate.

Eskandari and Hashemi [96] studied semantic patterns in API Call Graphs extracted from metamorphic malware. In such graphs, nodes exists in each part of the malware's code, which makes a system's API call and directed edges represent the sequence where it occurs. From each malware in a set, an API call graph was constructed and was later reduced to a feature vector. With the obtained vectors, classification algorithms were employed to train models that labeled an instance into benign or malware. Experimental outcomes presented accuracy scores below 85% for all the models of classification obtained.

3.2 Identification using code signature matching

This section presents approaches of metamorphic malware identification using similarity-matching of structures extracted from a code's body. In such approaches, structures that represent the semantics and functionalities of assembly codes are extracted from malwares. Those structures are selected due to their resilience to the effects of metamorphism, since metamorphic instances are generated by mutations applied to existent malwares that do not alter their original functions.

Baysa, Low, and Stamp [29] proposed a solution for metamorphic malware identification based on matching of files with a similarity-matching algorithm called string edit distance [97], which was applied to code sequence segments of files under analysis and known malwares. Their solution focused on code sorting and analysis of data areas, characterizing them by the peculiarity of their byte sequences, as well as their lengths, which was later used to compare two files. This solution started by splitting each file into sequences of segments with different entropy levels, using wavelet [98] and entropy analysis [99], then aligned the sequences of the two files by calculating the edit distance between the segments, obtaining the similarity between two files. Such technique was tested on metamorphic variants created with the following virus creation kits and engines of metamorphism: *a)* G2, *b)* NGVCK and *c)* MWORM. For each family under analysis, similarity scores were higher among instances of the same family and between malware and benign instances. For each family under analysis, similarity scores are lifted among instances of the same family and between malware and benign instances. For G2 and MWORM, the difference between similarity scores among malwares and benign instances was evident. However, as seen in Figures 3.1 and 3.2, it was not possible to select a threshold that completely separated NGVCK generated instances from benign ones without incurring false positives or false negatives. Experimentation did not compare instances of different families, so it was not possible to confirm if the method was able to distinguish between different families from the results.

Methods based on matching structures in malware families rely on the assumption that the selected structure is resilient to the effect of metamorphism. One approach that seeks metamorphic-resilient structures is code normalization [100] [101].

Intermediate languages are widely used for code normalization. Alam, Hospool, and Traore [34] studied problems with previous intermediate languages and developed a new language called MAIL (Malware Analysis Intermediate Language). As described by the authors, MAIL was designed as a small, simple, and extensive language that represents structural and behavioral information of an assembly program by translating instructions into a representation grouped according to

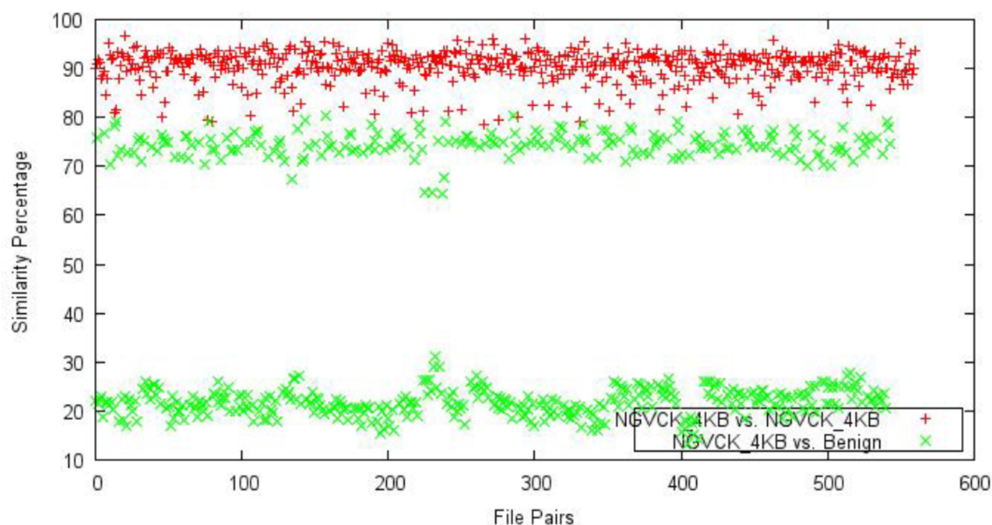

Figure 3.1: Similarity scores between one iteration generated from NGVCK instances and benign files according to Baysa et. al approach

its semantic. To perform detection, a binary program is first disassembled and translated to a MAIL program. The MAIL program is then annotated with patterns that represent the logic of each line of code. A Control Flow Graph (CFG) is later built from the annotated program, which becomes part of the program's signature and is matched against a database of known malware samples. The CFG is divided into smaller CFGs, each per function found in the code. A program that contains part of the control flow from a training malware sample is classified as a malware if a percentage (compared to some predefined threshold) of the CFGs involved in a malware signature match the signature of a stored sample of known malware. For successful matching, all the statements in the matching blocks must have the same patterns. Experimental evaluation using an existing dataset yields a malware detection rate of 93.92% and false positive rate of 3.02%.

A programs' semantics can be effectively represented by graphs in which the portions of code and its relations are modeled by nodes and edges. Those graphs are usually called dependency graphs since there structures represent dependencies between parts of code. One advantage of this kind of signature is that it is more resilient to metamorphism. Since obfuscation techniques must be applied without eliminating original functioning, dependency graphs represent the original intention and function of the code. Therefore, a detection system, using dependency graphs as signature, can classify files by comparing the dependency graph of that file with stored graphs, previously identified as malware.

Xin Hu et al. [102] proposed a malware database management system called

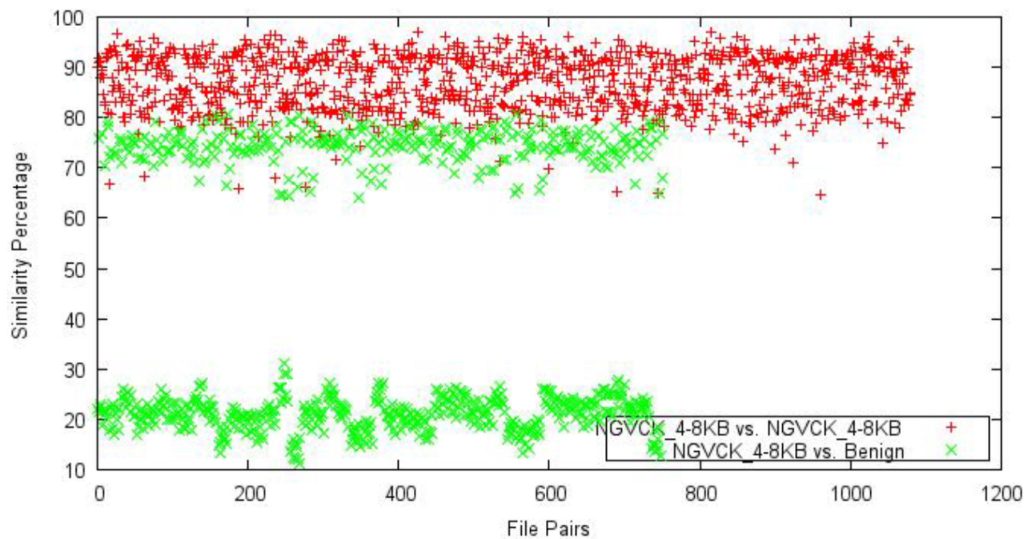

Figure 3.2: Similarity scores between two iterations generated from NGVCK instances and benign files according to Baysa et. al approach

SMIT (Symantec Malware Indexing Tree), in which detection is based on matching malware's function-call graphs. Since each malware program is represented as a graph, the problem of searching for the most similar malware program in a database to a given malware sample is cast into a K Nearest Neighbors [103] search problem in a graph database. To speed up such search, the authors developed a method to compute graph similarity that exploits the structural and instruction-level information in the underlying malware programs, and a multiresolution indexing scheme that uses feature vectors for early pruning. Xin Hu et al. described effective pruning power and scalability of the nearest neighbor search mechanism proposed. Experimentation results showed database query execution up to 400 seconds for less than 50 graphs distances computations. The authors recognized that, since SMIT analyzes malware samples at the level of individual instructions and function calls, it may be susceptible to advanced obfuscation techniques such as *a)* instruction reordering, *b)* equivalent instruction substitution and import table modification (to hide the symbolic names of imported functions).

Lee et al. [104] proposed a method for malware detection resilient to metamorphism based on the comparison of known malwares of suspicious instances using API call graphs extracted from the instances. In such graph, there is a node for each system's API call made by the program and edges connect such calls according to the sequence in which such calls are made. Afterwards, the nodes are classified into groups depending on the type of operation that is performed, which are: *a)*open; *b)*close;*c)* write and *d)* read. Also, the object involved in the opera-

tion is included in the node's label, which is selected from 32 objects present in the Microsoft Documentation reference (MSDN), including: *a)* socket, to represent the interface object involved in a network operation and *b)* registry, to represent an object involved in a read/write operation. Then, a similarity index is calculated by computing the intersection and union of the graphs constructed and by dividing the number of unions by the number of edge intersections. Experiments tested how uniquely the proposed graph structure represented the semantics by comparing graphs extracted from benign and malicious files. If comparison outputs a similarity score above 0.9, then the graphs were considered equal. One out of 30,000 possible pairs of malware and benign instances in the experimental dataset had a similarity greater than 0.9. The system tested generated variants of 10 malwares by applying code insertion, code reordering, and code replacement for 10% to 100% of the sample codes, obtaining an accuracy score of 91% and a false positive score of 6%. Such method presented performance issues associated with extensive graph comparison operations of big datasets.

Runwal et al. [105] presented an approach for metamorphism identification based on executable file differentiation. The authors developed a similarity measure based on extracted opcode sequences that can be used to compare executable files. The method proposed employs weighted directed graphs that express the probability an opcode appears after another in a family of malwares. Such graphs are normalized by adjacency matrices made from all the possible opcodes in an 8,086 assembly code, and the probability of each opcode occurring after the others in each graph. Then, the problem is reduced to compare such matrices with that of an input graph to determine if an instance was generated from another instance in the dataset. Experiments about benign, malware, and partially morphed of each kind showed coefficients of similarities with an average over 0.5 between instances of the same family. Such approach was inefficient for code integration where malwares were introduced into the body of known benign instances, as well as for removing uncommon opcodes from the malware.

Kim and Moon [39] studied the effects of metamorphism in data dependency graphs and proposed an approach based on graph matching for malware identification. Regarding effects of metamorphism on the Data Dependency Graphs, through experimentation, the authors concluded: *a)* format alteration and variable renaming do not change anything, even in the control flow graph, *b)* statement reordering actually changes the order of vertices in the dependency graph while the structure remains the same, *c)* more complex techniques such as statement replacement, control replacement, and junk code insertion can add one or more vertices into the dependency graph, however, new vertices do not harm the structure of the previous dependency graph because it should keep the same functionality as before and d)Spaghetti code alters the control flow of a graph

and adds vertices for unconditional jump into the dependency graph. Likewise, authors implemented previous experiences from Liu et. al [106], which concluded that DDGs are robust to the following five disguises in software plagiarism: format alteration, variable renaming, statement reordering, control replacement, and junk code insertion.

Kim and Moon labeled an instance a metamorphic variant by comparing maximum subgraphs isomorphism [107]. Since this process involves high computational complexity, heuristics based on the use of genetic algorithms are employed to reduce comparison time. This method was tested against metamorphic variants of malware scripts found in the wild. The dataset also included generated instances that applied techniques of metamorphism. When compared to a set of commercial antiviruses, the approach by Kim and Moon outperformed by detecting 16 of the 18 variants tested. Kim and Moon's method correctly identified 100% of the malware families highly metamorphic variants: *Hello*, *Neves Rabit* and *Small*.

Figure 3.3 shows the scores of differences among metamorphic malware instances in the same family, calculated by Kim and Moon. The figure shows that two instances presented a similarity of 50% and 80% respectively, while all the other instances presented similarity of 100%.

Virus	Hello	Neves	Rabbit	Internal	Small
Hello	**0**	28.13	22.22	40.91	40
Hello.v1	**0**	28.13	22.22	36.36	40
Hello.v2	**0**	28.13	11.11	36.36	40
Hello.v3	**0**	31.25	22.22	31.82	40
Hello.v4	**0**	34.38	22.22	36.36	40
Neves.a	28.13	**0**	22.22	36.36	40
Neves.b	63.16	**0**	22.22	31.82	40
Neves.c	28.13	**0**	22.22	36.36	40
Neves.d	31.58	**0**	33.33	27.27	40
Rabbit.a	22.22	22.22	**0**	155.56	60
Rabbit.b	20	40	**0**	130	60
Internal.a	55.3	40.6	**22.2**	27.3	40
Internal.b	20	22	77.78	**0**	40
Internal.c	40.91	36.36	155.56	**0**	40
Internal.f	21	22	77.78	**0**	40
Internal.g	44.7	40.6	44.4	50	**40**
Small.a	40	40	60	40	**0**
Small.b	28.57	42.86	57.14	35.71	**0**

Figure 3.3: Differences among metamorphic malware families in (Kim and Moon, 2010)

Breves et al. [38] proposed an approach based on the use of Data Dependency Graphs, with emphasis on a technique that reduces the complexity without compromising the accuracy of graph matching. Initially, vertexes were classified into:

a) load, where a variable is created and starts a dependency chain, b) processor, a variable that transform the data, and c) decision, a variable that modifies the flow of the program based on some condition (an assembly code found as CMP statements that are placed before a jump instruction). Graphs were pruned considering the most relevant vertices only from those classified as decision. To identify the most relevant decision vertices, a four step process was performed, as follows: *a)* calculate the shortest relative distance between each decision vertex, *b)* construct a derived virtual graph, consisting only of decision vertices, with edges representing the relative distance between each vertex of the decision, *c)* calculate the maximum clique derived from this virtual graph and *d)* reduce the dependency graph by removing any vertices and edges that are not associated with the vertices of decision present in the maximum clique of the derived virtual graph.

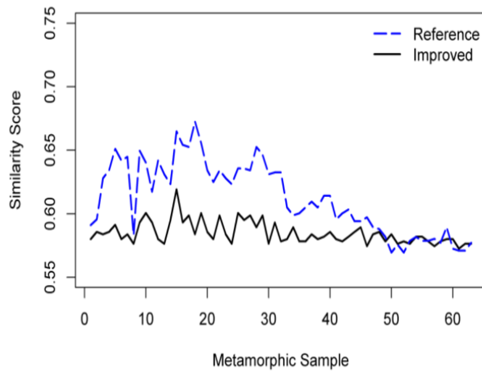

Figure 3.4: Similarity score among metamorphic variants of Evol malware family (Breves et.al,2015).

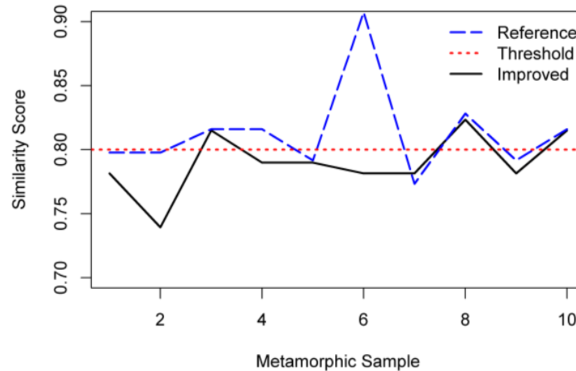

Figure 3.5: Similarity score among metamorphic variants of Polipmalware family (Breves et.al,2015).

Breves' method was applied to 63 samples of Evol.a malware and 10 samples of Polip malware. All the samples were then subjected to comparison with metamorphic versions using either the graph generated by the reference reduction process or the version reduced by the use of virtual clique. As a result, the number of vertices reduced on average was 23.52%. Figures 3.4 and 3.5 present scores of similarities among metamorphic variants of malware families Evol.a and Polip respectively, as presented by Breves et.al. In such figures, the average similarity among samples of the family Polip is around 80% and 60% among instances of the Evol family. In all cases, the DDGs extracted from samples in the same family remained similar for over 50% of its structure.

3.3 Final considerations

The previous sections include a set of state-of-the-art studies about metamorphic malware detection. Such studies were found by applying several search criteria related to the topic in the following scientific research databases : IEEE Xplore, Elsevier, ACM Press and Springer. Finally, studies that elaborated about the topic and presented relevant results were selected. Table 3.3 summarizes the selected studies, presenting the detection techniques used, best accuracy score, best false positive rates, and size of employed dataset. The following are some deficiencies identified in the selected works:

The high variability in patterns and behaviors reached by metamorphism and the high rate of appearing of new vulnerabilities and stealth techniques is a factor that affects the capacity of generalization of detection models. Works that perform detection by separating into malware and benign are prone to low accuracy when several characteristics of benign code are inserted into malicious variants [82]. In general, approaches that perform detections by classifying into malware families have more discrimination capacity.

Approaches based on comparisons of semantic representations of code body are more resilient to metamorphism mutations [39], however, still deal with performance issues associated to matching algorithms[107] [108]. Graph matching in growing bases of millions of signatures is time consuming, as suspicious instances have to be compared with each instance in the database, in a brute force fashion. This only worsens when considering the rate at which databases grow, with thousands of new malware instances found every day.

Table 3.3: Comparative table of state-of-the-art research on metamorphic malware detection

Work reference	**Technique**	**Detection accuracy (Best result)**	**False positive rate (Best result)**	**Data set size (Benign/Malware)**
(CANFORA; IANNACCONE; VISAGGIO, 2013)	Code instructions mining	94%	3%	500/500
(SINGH et al., 2015)	Support Vector Machines	88%	4%	40/950

(CHOUDHARY; VIDYARTHI, 2015)	Dynamic analysis, text mining	97.8%	1%	97/91
(VINOD et al., 2014)	Statistical analysis, mnemonic n-gram	96%	9%	4050/2430
(BAYSA; LOW; STAMP, 2013)	Statistical analysis, structural entropy	100	0	16/100
(ESKANDARI; HASHEMI, 2012)	API Call graph mining	85%	48%	2140/2305
(ALAM; HORSPOOL; TRAORE, 2013)	Code normalization, intermediate language, subgraph matching	93.92%	3.02%	1137/250
(PAYANDEH, 2014)	Hidden Markov Models	70%	0%	0/200
(LEE; JEONG; LEE, 2010)	System API Call Graph Matching	91%	1.3%	300/100
(RUNWAL; LOW; STAMP, 2012)	Opcode graph similarity	96%	6%	41/225
(ALAM et al., 2015)	Code normalization, annotated control flow graph matching	93.92%	3.02%	2330/1020

(KIM; MOON, 2010)	Data dependency graphs matching	100%	0%	0/18
(MARTINS BREVES; FREITAS; SOUTO, 2014)	Data dependency graphs matching	100%	0%	0/73
(ROJAS; MARTINS BREVES; SOUTO, 2017) This work	Data dependency graphs' database indexing	97.7%	3.3%	0/4530

Matching semantic structures provides lower false positive rates since higher discrimination and uniqueness between instances can be obtained, however, performance issues remain. On the other hand, pattern recognition is more practical due to lower algorithmic complexity, though, the chances a malware instance will be labeled as benign and vice-versa are higher.

In order to leverage advantages and overcome limitations found in previous studies , we present a hybrid approach for semantic structures matching and pattern recognition. This approach is based on the studies of Kim and Moon and Breves et.al. In these works, and herein, Data Dependency Graphs are used because of their resilience to metamorphism, as studied by Kim and Moon [39] and Liu et. al [106]. However, in our proposed methods,, to reduce the effects of metamorphism and create a structure that better represents the graphs, we use information about the semantic of programs by labeling DDG's nodes with a tag from assembly instructions classification, which allowed us to introduce the concept of Annotated Data Dependency Graphs (ADDG) as stated in Definition 13. In the previous studies, shrinking techniques were proposed to reduce the quantity of nodes compared during graph matching. Herein, such pre-processing is not applied directly to the graphs, but to feature vectors extracted from the graphs in a process of information gain optimization. Furthermore, different then in previous proposals, in this work, comparisons are applied not by exact matching of instances but based on pattern matching employing models of classification trained with machine learning algorithms. Such models are optimized to discriminate instances from each other in the same dataset by selecting the most prominent and representative features of the graphs, reducing complexity of comparison and enhancing matching accuracy. The next chapter presents all the insights of the proposed methods.

Chapter 4

Methods proposal

Herein, we propose a method for malware detection that is resilient to code metamorphism through Data Dependency Graphs (DDG) matching. Such graphs were selected given their resilience to the effects of metamorphism and code mutation, as seen in previous studies [46] [39] [38].

However, such studies have shown that DDGs are not fully immune to metamorphism. Kim and Moon [39] stated that complex techniques such as statement replacement, control replacement, and junk code insertion can add one or more vertices into a dependency graph, even though new vertices do not harm the structure of the previous dependency graph since the same functionality should remain. Furthermore, spaghetti codes alter the control flow of graphs and add vertices for unconditional jump to dependency graphs. Subroutine inlining and outlining is the most complex technique and can modify up to 50% of a dependency graph [39] [46]. In addition, generated ADDGs may not be accurate due to deficiencies inherent to binary reverse engineering.

Given the previously mentioned drawbacks, which can affect the accuracy of constructing such graphs, DDGs matching through graph isomorphism (exact matching) is prone to false negatives. Since most of the graphs are not modified, subgraph isomorphism is best suited, notwithstanding, subgraph isomorphism is NP-Hard [40]. For that reason, herein, DDGs matching is performed through pattern recognition instead of exact graph matching, using classification models trained with machine learning algorithms.

Such models are recognized as an index since they recover a previously identified malware from a reference base. A feature vector extracted from an ADDG represents a query to such index. The inner structure of the classification model discriminates among malware families according to structural characteristics of the Annotated Data Dependency Graphs (ADDG) that conforms the instances in such families. Such models are obtained through a process of fitting that optimizes such discrimination. This all identifies which malware a binary is related

to by submitting the feature vector of an ADDG extracted from such instance, without comparing the ADDG with all the malwares in the base.

Therefore, we propose a method for metamorphic malware identification comprised of two phases: a) index induction and b) suspicious instance classification. The first phase constructs an index structure that permits perform matching of an input instance to a base of graphs extracted from known malware. The second phase identifies if an input suspicious instance was generated from one of the instances from malwares base. Figure 4.1 presents a diagram representing the proposed method. The next sections provide details about the method's components.

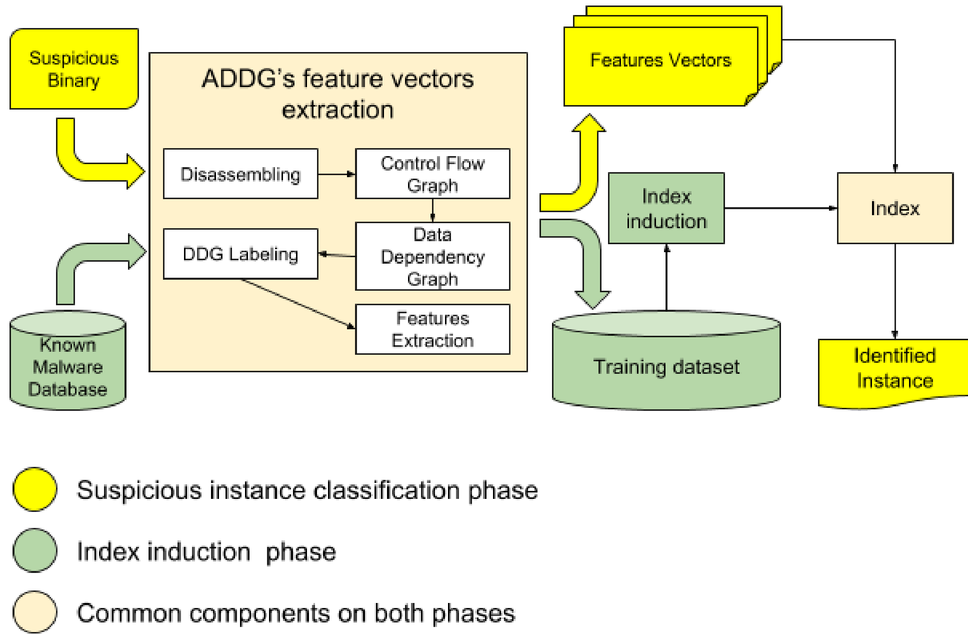

Figure 4.1: Overview of the proposed approach for metamorphic malware detection.

4.1 Feature vectors extraction

For both phases of the proposed methods, we implemented a static process of feature vectors extraction. Such vectors represent structural characteristics of ADDGs extracted from the assembly code in previously identified malwares, during index induction, and in suspicious instances during index querying. In order to construct such feature vectors, the following steps were sequentially executed for each input binary instance: *a)* binary disassembling, *b)* Control Flow Graph construction, *c)* Data Dependency Graph construction, *d)* Annotated Data Dependency Graph construction and e) ADDG's structural features extraction.

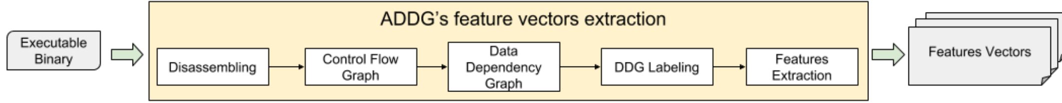

Figure 4.2: Overview of feature vectors extraction phase.

Figure 4.2 shows the steps of feature extraction proposed in this research. Initially, the object code (in which the input binary program was expressed) was reverse engineered to transform the opcodes to instructions in assembly code. For each function in the obtained assembly code representation, a Control Flow Graph (CFG) was constructed that expressed the order of execution of code blocks inside the function. The obtained CFGs were then transformed into Data Dependency Graphs (DDG) using data flow analysis, which mapped data dependencies among instructions throughout all blocks of the CFG. Then, each node of the DDGs was given a label that expressed the semantic of the instruction contained in such node, based on the instruction's mnemonic and operands, resulting in an ADDG for each DDG. From the obtained ADDGs, structural features were extracted to conform the feature vectors employed during index induction and querying.

4.1.1 Data Dependency Graph construction

The first step for feature vector extraction was construction of the Data Dependency Graphs based on executable programs. During index induction, DDGs were extracted from known malware samples and during the process of analysis from the input suspicious instance. This section explains the process of binary code's DDG extraction used.

Given an executable binary, the object code was expressed in a higher level format that represented the semantics and functioning of the program, best suited for data and control flow analysis. Initially, a process of reverse engineering was applied to binaries, to recover a representation in assembly code. Afterwards, a CFG was constructed through a process of control flow analysis for each function in the assembly code obtained. Such steps can be performed by reverse engineering frameworks like: a) Radare2 [109], b) IDA-Pro [87] and c) LLVM [110].

From each CFG, a DDG was constructed as a result of data flow analysis. Such process aimed to identify the points in a code where a variable was modified and/or consumed and connected those points to develop a graph structure with the characteristics of a DDG, as detailed in Section 2.3.

Algorithm 2: Worker-list approach for Data-flow analysis and DDG construction

```
1  function BuildGDD (CFG);
   Input  : Control Flow Graph of binary function
   Output: Data Dependency Graph
2  Out(s) ← ∅ for s in CFG ;
3  V ← ∅ ;
4  E ← ∅ ;
5  W ← ∅ ;
6  W.Push(CFG.entryBB)
7  while W ≠ ∅ do
8      s ← W.Pop() ;
9      In(s) ← ∩_{s'∈preds(s)} Out(s') ;
10     temp ← In(s) ;
11     for Instruction i in s do
12         for Variable v read by i do
13             if v exists in temp then
14                 if i not in V then
15                     V.Add(i) ;
16                 end
17                 if temp[v] not in V then
18                     V.Add(temp[v]) ;
19                 end
20                 E.Add(Vertex(i, temp[v])) ;
21             end
22         end
23         for Variable v modified by i do
24             temp[v] ← i ;
25             if i not in V then
26                 V.Add(i) ;
27             end
28         end
29     end
30     if Out(s) ≠ temp then
31         W.Push(Successors(s))
32     end
33 end
34 DDG ← Graph(V, E) ;
35 return DDG ;
```

We propose an algorithm for data flow analysis based on *the work-list approach* [111]. In such algorithm, presented in *Algorithm 2*, a CFG is run through

iteratively, block by block, to identify the instructions that perform read/write operations and determine chains of data-dependency.

On each iteration of the algorithm, a DDG is constructed by applying updates to its structure based on the states of data-dependency chains.

Data dependency chains are expressed in two mapping functions, namely *In()* and *Out()*. Given a variable x, function *In()* provides a reference to the instruction that last modified x, considering all the predecessors of the basic block under analysis for each iteration of the algorithm. Function *Out()* maintains a map of variables of the instructions that modify it only for the instructions inside each block. A map *Out()* is saved for each block and updated with new relationships of data-dependency found in each iteration of the algorithm.

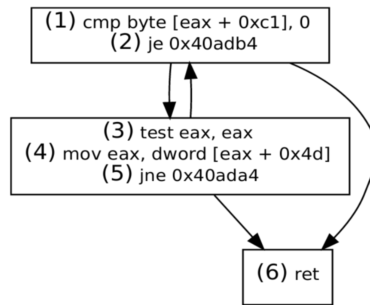

Figure 4.3: Example of a CFG extracted from a function of program *ls*

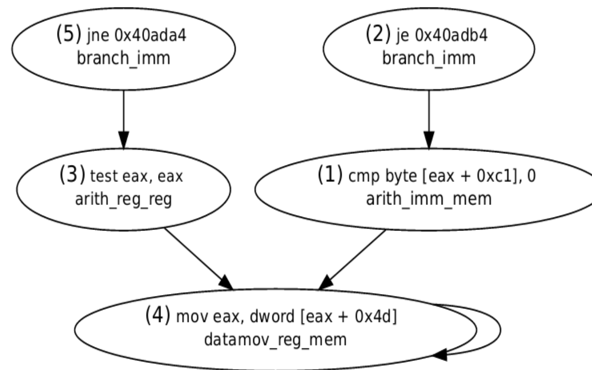

Figure 4.4: Labeled DDG extracted from the CFG in Figure 4.3

The basic blocks processed are collected into a queue. On each iteration, a block is taken from the queue, thus the algorithm stops when the queue is empty. When the algorithm starts execution, the queue only contains the basic block referring to the analyzed assembly code's entry point. After each iteration, the queue is populated with the successors of the block processed on that iteration.

If the state of the last computed *Out()* of the analyzed block differs from the one computed in the current iteration, no population in the queue is done. As the processing queue is populated according to the blocks connectivity, isolated blocks end up being taken out of analysis.

On each iteration, *In()* is initially computed by intersecting the *Out()* of the immediate predecessors of the block being processed. Then, a new *Out()* starts to be computed for the current block and is initialized with the state of *In()*. From this point on, if each instruction in the block handles data (read/write operation) involving a variable or flag, one of the followings operations are performed:

i. If the variable is being manipulated for the first time, a new vertex is added to the DDG and *Out()* is updated with a map relating to the variable and address of the instruction.

ii. If the variable is already mapped in *Out()* and its value is not changed by current instruction, one of the following actions is taken:

ii.i) If the instruction that modifies the variable value already has a vertex in the DDG a new node is created for the current instruction;

ii.ii) Otherwise, two nodes are created in the DDG, one for current instruction and the other for the instruction that changes the variable value. In either case, a new edge is added to the graph, outgoing from the node of the instruction that modifies towards consuming node.

iii. If the instruction alter the contents of the variable, operations described in *ii.* are executed, and the map of the variable in *Out()* is updated to contain only the current instruction.

Figures 4.3 and 4.4 show examples of a DDG and the CFG used to construct the DDG. As shown in Figure 4.4, there are outgoing edges from the vertices that represent instructions 3 and 1 and entering into the vertex for instruction 4, given that 4 defines the value of the register *eax* and this same variable is consumed by 3 and 1.

Some metamorphic engines implement trash instruction insertion by transforming one instruction into many equal ones, mostly in arithmetic operations as sums, or by inserting several *nop* instructions (i.e instructions that perform no action). To make the process less susceptible to such obfuscation, equal successive instructions are handled only once.

4.1.2 Annotated Data Dependency Graph construction

At the same time a DDG is being constructed, its nodes are labeled to conform an ADDG. ADDGs constitute an enhanced program representation since they also include semantic and functionality of instructions. ADDGs bring information about program's semantics into the identification process, which provides

classification models with better discrimination capacity, thus, enhance identification accuracy. Furthermore, this process aims at reducing the effects of single instruction substitution by providing a label for each node in the DDG that represents all possible node's equivalent instructions. This section elaborates details about the ADDG construction process.

Given a node v of a DDG, v is labeled with a string followed by the union of substrings, each representing a classification of the parts that conforms the instruction in v (i.e mnemonic and operands). Such label follows the naming pattern $I = I_m_I_o$ for any given instruction I, in which I_m is a class for the mnemonic in I and I_o is the union of classes assigned to each element in the operands present in I separated by an underline character.

Table 4.1: Example of mnemonics and their classes according to the x86asm[dot]net reference.

Classes	**Mnemonics**
branch	jnl, jnae, call, jnp, ret, loopne, jno, jb, jnb, js, alter, jpe, ja, jmpe, retn
stack	push, popa, pushad, pushf, pushal, pop
datamov	setne, movd, setge, movlps, movbe, cmovo, sete, fistp
control	fdisi, wait, fstcw, hint, nop,fneni nop, fwait, ud2

The classes employed for I_m (mnemonic in the instruction) are taken from the map mnemonic-class proposed in the *x86[dot]net* reference [112]. This reference maps 709 mnemonics into 60 class labels according to their semantics, for example, mnemonics that: *a)* perform control flow operations creating a branch in the program execution flow (*jne, ret, jnp etc.*) are labeled branch, *b)* perform operations with the program's memory stack (*push, pop, pushal*) are labeled stack and *c)* perform operations that modify the pace of program execution (*wait, nop*) are labeled *control*. Regardless of the label assigned, such classes group mnemonics according to their meaning. Table 4.1 presents examples of mappings proposed by such reference.

Table 4.2: Examples of classifications given to assembly code instruction operands by mapping references in LLVM and Capstone.

Operands	**Classification**
0x40ada4, 0x40adb4 (immediate value of fixed address)	imm
eax, ebx, ecx (registers)	reg
byte[eax+0xc1], dword[eax+0x4d] (calculated values from data in memory)	mem

To create I_o (label for operands) the mapping of class-operand is that used by disassemblers LLVM [110] and Capstone [113]. In such mapping, reference operands are classified according to their meaning, for example: *a)* fixed code addresses are labeled *imm* (*immediate values*); *b)* registers are classified as reg and *c)* calculated values from memory data are tagged with *mem*. Table 4.2 presents examples of such classifications, more examples can be found in the source code of such frameworks that are publicly available [1] [2].

Figure 4.4 shows an example of the application of the proposed labeling method. For the instruction in line 3, mnemonic *test* is translated to class *arith* and operands eax, eax to class *reg_reg*, in the corresponding label.

4.1.3 Feature vectors construction

After ADDG construction, feature vectors extraction are executed during the phases of index induction and suspicious instance analysis for identification, . Such feature vectors are employed in the process of model training and as input to identification models during suspicious instance's analysis. This section provides explanations about the process of ADDG's feature vector extraction.

For each ADDG extracted from each function of assembly code, a feature vector was constructed. Such process was executed as the DDG was traversed for labeling. These feature vectors expressed structural characteristics of the ADDG and also included information on the operations expressed by the instruction of each node in the ADDG.

Given a feature vector f extracted from an ADDG, for any given characteristic c_w of f, c_w is tagged following the naming convention expressed by the pattern $c_w = F_e_F_r$. In this pattern, F_r is reserved to include a label from the set of labels used to annotate DDGs and F_e represents prefixes derived from the graph structure, being: a) n for the number of occurrences of F_r in a graph's nodes, b) i for quantity of edges entering into vertices containing F_r and c) o to express the quantity of edges originated in nodes containing F_r.

Table 4.3 presents an example of a feature vector extracted from the DDG in Figure 4.4. Since the graph contains two vertices with the label *branch_imm*, the entry for the feature *n_branch_imm* has the value of 2. Also *i_arith_reg_reg* is valued 1 given that there is one edge entering vertices with *arith_reg_reg* from the DDG in Figure 4.4.

Since ADDGs only contained nodes related to instructions that perform read and write operations, feature labels that were not related to data handling operations received the value zero for all the feature vectors. Given the low variability

[1] Unofficial Automated Mirror of LLVM, Online: https://github.com/llvm-mirror
[2] Capstone Github Repository, Online: https://github.com/aquynh/capstone

Table 4.3: Example of a feature vector extracted from the DDG in Figure 4.4.

n_branch_imm	i_branch_imm	o_branch_imm	n_arith_reg_reg	i_arith_reg_reg	...
2	0	2	1	1	...

in values of such features throughout the dataset, such features presented the lowest entropy and highest information gain, which gave them a better score in the process of feature selection. In general, such behavior was undesired since it affected generalization and reduced the ability to correctly classify new instances.

Since the quantity of features in the feature vectors used herein were higher than 50% of most training datasets, feature vectors were considered of high-dimensionality. This made the available data sparse, which is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality. Also, organizing and searching data often relies on detecting areas where objects form groups with similar properties. However, for high dimensional data, all objects appear to be sparse and dissimilar in many ways, which prevents common data organization strategies from being efficient. In machine learning and data mining this phenomenon is known as the curse of dimensionality [114].

In order to reduce bias towards features with higher information gain [85], feature selection is recommended before models creation to decrease the model's training and prediction execution times, as well as to avoid the curse of dimensionality and enhance accuracy of identification.

4.2 Index induction

The problem with identifying instances with the same semantic pattern as suspicious input binary, based on its ADDG, is formulated as a classification task. Given the dataset of feature vectors extracted from the ADDGs in known malware samples, a classification model was created with a class for each malware instance represented in the dataset. Such step was denoted as index induction since the model induced was employed as an index structure. For classification, the models received a feature vector as input to obtain an identifier of the malware as output, the instance that ADDG belonged to was related. The previously explained consists of index induction and identification work-flows, which are presented in Figure 4.5.

In order to create such models, we adopted a method of supervised classifiers training [72]. Given a training data set of the form $\{(x_i, y_i)\}$ for i in $\{1...N\}$,

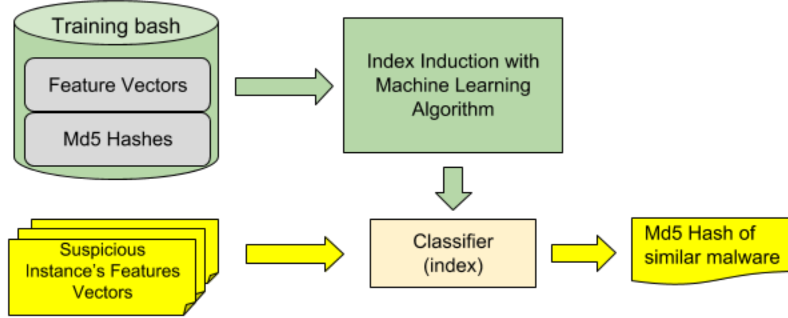

Figure 4.5: Index induction and identification work-flows

the dataset with N feature vectors, such that $x_i \in \mathbb{R}^d$ is the feature vector of the i-th example and y_i is its label (i.e class, identifier of a malware sample the ith ADDG belongs to). The objective is to apply a learning algorithm to fit the function $g : X \rightarrow Y$ where X is the input space and Y is the output space.

There are two classical approaches for dealing with multiple class data sets (i.e more than two classes): a single classifier that discriminates among all classes, or alternatively, by dividing the problem into multiple classifiers [115]. The first approach is also called multinomial, polytomous, or single-label classification and can be formally expressed as finding a classifier g that discriminates among K classes employing a scoring function $f : X \times Y \rightarrow \mathbb{R}$ that co-relates the probability of a feature vector x_i to belong to class y_k for $k \in \{1...K\}$, such that g is defined as returning the y value that gives the highest score of probability $g(x) = \arg\max_x f(x,y)$. One-vs-Rest and One-vs-All are examples of strategies employed to reduce multi-class classification to multiple classifiers [115].

In the proposed methods, both approaches for multi-class classification fit the component *Index induction*. In this work, we explore single classifiers for multi-class classification, as well as One-vs-Rest strategy for multi-class to multiple classifiers reduction.

4.2.1 Single-model multi-class classification

For this strategy, the goal was to construct models that gave the feature vectors of ADDGs extracted from a given binary as outputs for an identifier of the seed instance from which such binary was generated.

Such models were intended to identify and separate patterns of the functioning of each family expressed in the ADDGs. Therefore, a supervised training process was executed, where machine learning algorithms were given the previously constructed feature vectors as input and the md5 hash of the seed instance from which the binary input graph was extracted from was the expected output.

Such process is known as training.

For models to present higher accuracy on classification when training with seed instance information only, structural patterns for each family should be caught from the information of one member. Since ADDGs within the same family also belong to the same identity patterns, models that better over-fit the data are prone to present higher accuracy. However, it is possible that instances from different families while training, present similar patterns and end up being grouped in the same cluster of the classifier inner structure. As the purpose is to discover which family an instance belongs to, this is convenient since an instances' membership is not known beforehand.

4.2.2 One-vs-Rest for multiple-classifiers multi-class classification

One-Vs-Rest (OVR) is the most commonly used strategy for multi-class to multiple subproblems reduction [115]. This consists of fitting one classifier per class. For each classifier, the class is fitted against all the other classes. Algorithm 3 details training following the One Vs Rest approach.

Algorithm 3: Algorithm for training models following the One-vs-Rest approach

```
1 function TrainOvR (L,X,Y);
  Input  :
  A machine learning algorithm L ;
  Set X of size K made of the feature vectors; Set of targets Y of size K
  where Y_{i∈{1...K}} is the expected output of sample X_i
  Output:
  A list of classifiers f_k for k ∈ {1...K}
2 for k in {1...K} do
3 |   Construct a new set of targets z of size K in which z_i = 1 if y_i = k and
  |     z_i = 0 otherwise ;
4 |   Apply L to X,z to obtain f_k
5 end
```

Ground truth employed in the multi-class approach needs to be binarized to fulfill the requirements of OvR, in which not only classifiers are to be trained to detect if an instance belongs to a class but also if it does not belongs. Given a dataset D of size N with K classes, binarization consists on replacing the ground truth with a set of vectors V in which $V_{i \in 1,...N}$ has shape 1 by K.

Table 4.4: Example of ground truth binarization applied.

Original Ground Truth	Binarized Ground Truth			
	Label Vector V_i			
	j=1 (Class A)	j=2 (Class B)	j=3 (Class C)	j=4 (Class D)
$Y_1 = \text{A}$	1	0	0	0
$Y_2 = \text{B}$	0	1	0	0
$Y_3 = \text{C}$	0	0	1	0
$Y_4 = \text{A}$	1	0	0	0
$Y_5 = \text{C}$	0	0	1	0
$Y_6 = \text{D}$	0	0	0	1

For all vectors in V, every position $j \in 1, ...K$ in V_i represents a unique class. V_i receives 1 in j and 0 for all other positions if j represents the same class as that of expected output in the original dataset D. In Algorithm 3 binarization occurs in line number 3. In Table 4.4 an example of a binarized dataset is presented.

Classification means applying all classifiers to an unseen sample x and predicting the label k for which the corresponding classifier reports the highest confidence score $\hat{y} = \arg\max_{k \in \{1...K\}} f_k(x)$ where K represents the quantity of classes in the dataset as well as the quantity of binary classifiers. At that stage, each classifier in the OvR ensemble is applied to a feature vector of one ADDG extracted from the instance under analysis.

Table 4.5: Example of the use of OvR ensembles for classification.

	Probability for each class i			
	i=1	i=2	...	i=K
$p_{i=1}$	.85	.05	...	.27
$p_{i=2}$	.33	.76	...	.12
...	...	...	...	...
$p_{i=K}$	.48	.58	...	.95
$max(p_i)$	.85	.76	...	.95
Class selected is K with the highest probability(.95)				

Each classifier outputs a set p_i of size K containing at each position $i \in 1..K$ the probability given by the classifier of the input feature vector belonging to class i. All such sets p_i are summarized to one by selecting the highest score of probability in each position of p_i. Then, when the resulting set is selected, the correct class is represented by the set position with the highest score. Table 4.5 presents an example of the use of OvR ensembles for classification.

4.3 Suspicious instance classification

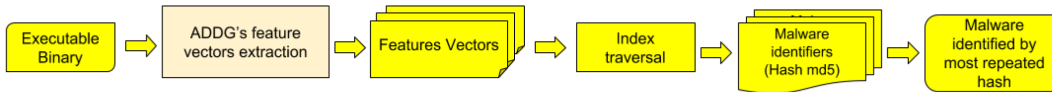

Figure 4.6: Suspicions instance classification phase

The models created in the phase of index induction were later employed to identify which malware family an input instance belonged to. For that, all the steps involved in the construction of ADDG's feature vectors were executed on the input instance, which resulted in a set of feature vectors. Such feature vectors were submitted to an index which outputs a set of md5 hashes, identifiers of binaries in the reference base. To determine which instance generated the input, the most repeated hash in the set of returned hashes was selected. The diagram in Figure 4.6 presents the steps involved in this phase.

4.4 Chapter's final considerations

This chapter presents the methods proposed for metamorphic malware identification based on DDGs matching. Each of the methods' components, as well as the intermediate structures and algorithms employed, are described separately in each section. Next, we present some aspects to be considered in future metamorphic malware identification studies.

The proposed methods depended on reverse engineering, as well as control and data flow analysis. The accuracy of such processes can be affected by the quality of the tool used for such tasks. Techniques like obfuscation, packing, and anti-analysis can be applied to affect the output of such tools. In this work, such problems are not addressed, and no specific tools are recommended, although we do site some tools used in previous works.

The assembly instructions classification proposed may not obtain equivalent instructions grouping and further evaluation and adjustment of such component could enhance identification accuracy. Also, since the mapping proposed focuses on Instruction Set Architecture (ISA), different mappings should be used according to the architecture of the suspicious binary analyzed.

The algorithms employed for index induction and feature selection, as well as the structural features considered, can also affect the accuracy of the models created following the proposed methods. In any case, model tuning, according to the dataset used, is recommended.

Chapter 5

Experimentation

In order to evaluate the proposed methods, we implemented a prototype following the definitions provided in Chapter 4. A set of mutated malware samples were submitted to the implemented prototype to evaluate its performance in terms of the components' execution times and classification accuracy. The next section presents a description of the experiments applied and the results obtained from those experiments.

5.1 Experimental setup

The experimental setup was comprised of all the components in the proposed methods., Experimentation was divided into two main blocks, according to the order the methods were executed, namely: feature vectors construction and dataset indexing. Figure 5.1 presents an overview of the experimental setup employed in this work.

Initially, a set of malware samples collected in the wild were submitted to two code mutators to generate a set of mutated metamorphic instances. The obtained metamorphic instances, as well as the malware samples collected in the wild, were submitted to reverse engineering to obtain a set of ADDGs from each malware instance. Afterwards, a process of feature vectors extraction was performed on the obtained ADDGs.

The set of feature vectors was divided into two datasets, one for training and one for testing. The training dataset was employed to fit classification models, following the single classifier and One-vs-Rest approaches for multi-class classification. The testing dataset was used to evaluate the accuracy of the obtained models.

Data preprocessing and malware identification was conducted in a virtual environment with 4 processing units of 2.1 Ghz each, 4GB of RAM, and 20GB

of swapping memory. Model training was executed using a PC with 62 processing units with 2.1 Ghz clock-rate and 128GB of RAM. The prototype was implemented in the version 3.5 of the Python programming language. The following subsections provide further explanation about the steps of the experimental setup.

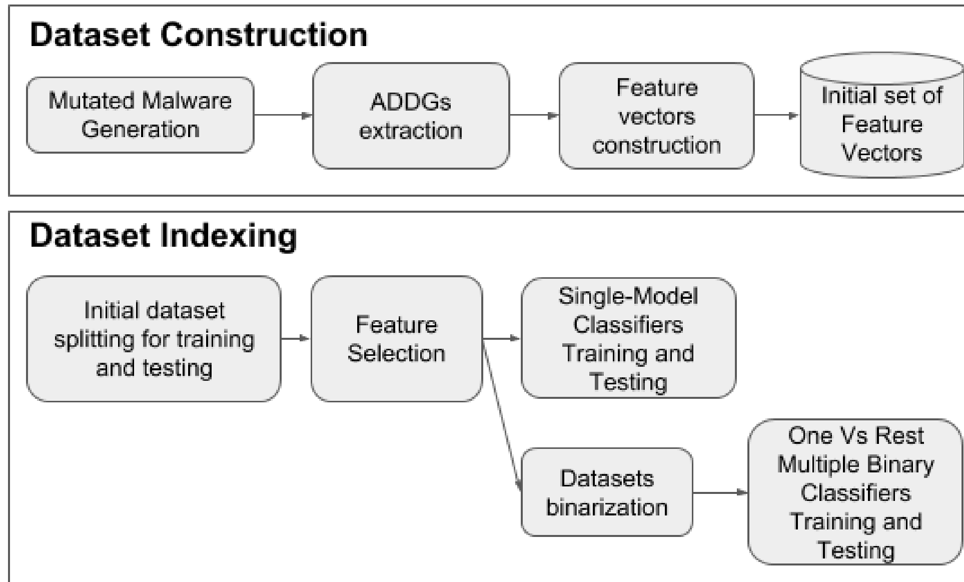

Figure 5.1: Experimentation phases and steps.

5.1.1 Dataset construction

The dataset used in this work consisted of the feature vectors extracted from ADDGs of malware instances found in the wild, as well as instances mutated with code metamorphism techniques. Initially a set of seed malware instances (i.e binaries samples used to generate metamorphic instances through mutations) were obtained in Malshare[dot]com [1], a public database of malware. Afterwards, ADDGs were extracted from seeds and mutated instances. Structural features were extracted from the obtained ADDGs to conform to the set of feature vectors employed for experimentation.

[1]Public repository of malware of the Malshare Project, Online: http://www.malshare.com

Mutated instances generation

To generate mutated instances we employed the code mutators Revert4 (R4) and Code Pervertor (CP) created by the Russian hacker Zombie[2], which can be obtained at vxheaven[dot]org. The code permutation techniques included in such tools are: *a)* Single and batch instructions substitution, *b)* Instruction reordering, *c)* Trash-code insertion and *d)* Variable renaming. Some executions for mutations were not completed due to errors that arose during the process. However, we ensured that for each seed instance at least three mutated instances for each mutator were included in the dataset. Table 5.1 presents the quantity of malware instances grouped by mutators, as well as the quantity of seed instances employed.

Table 5.1: Dataset distribution

	Malware samples	**Avg. File Size / Standard Deviation**	**Graphs extracted**	**Avg. Graphs per file/Standard Deviation**
Seeds	291	216Kb / 258	5825	20/36
Code pervertor	3574	365Kb / 395	13975	4/14
Revert4	665	371Kb/ 357	14720	22/30

In order to construct the experimental dataset of vectors for all the functions in the binaries of the malware instances batch, graphs and their feature vectors were extracted as a result of executing the implemented prototype. Table 5.1 includes the number of graphs that were extracted, and the average number of graphs extracted by binary, as well as the standard deviation of that measure.

ADDGs construction and feature extraction

As previously stated, the proposal presented for malware identification started with binary disassembling, CFG construction, and ADDG construction. Herein, the initial two tasks were performed through Radare [3], a framework for binary analysis that, among other steps, performs: *a)* entry point and symbol identification through flags analysis, *b)* cross reference identification, *c)* function structures identification, *d)* computed references identification through code emulation and *e)* consecutive function analysis [116]. After, an implementation of the Algorithm

[2]See official website on: http://z0mbie.daemonlab.org/
[3]Radare2 Github Repository, Online: https://github.com/radare/radare2

2 for data flow analysis was applied to the assembly code obtained from reverse engineering, resulting in an ADDG from each function in the binary.

It is of interest in this research to lift parameters and their values that evaluate suitability of the proposed approach so they can be used as a practical detection tool. To analyze how well the proposed methods performed when applied to real life malware binaries, we measured execution times for: *a)* disassembling, *b)* CFG construction and *c)* ADDG construction. For each file in the set of malware instances from the wild, as well as those generated by means of mutations, the initial steps of the method, until ADDG construction, were executed and for each step we calculated the time of execution as the difference between ending and starting timestamps (number of seconds since January 1st, 1970 in UTC), as follows:

$$Execution\ time = Ending\ timestamp - Starting\ timestamp \tag{5.1}$$

Also, we analyzed the influence of metamorphism on reverse engineering by comparing graph sizes and times of execution between seeds and mutated instances. Since mutation introduces complexity in the structures extracted from code, higher execution times and graph sizes are expected.

Since DDG labeling and feature vector construction were done while constructing the ADDG and such operations had complexity $O(1)$, execution times for those processes were not measured separately, but as part of ADDG construction.

5.1.2 Dataset indexing

Once we extracted the dataset of feature vectors, models of classification were trained to be used in the new suspicious files analysis phase. As previously stated, such step was performed following two approaches: *a)* single-model multi-class classifiers and *b)* ensemble of binary classifiers following One-vs-Rest.

For each classifier from both approaches of multi-class classification, we calculated the accuracy of identification as:

$$accuracy(y,y') = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(y'_i = y_i) \tag{5.2}$$

In equation 5.2, n represents the quantity of graphs in the testing dataset and y refers to the md5 hash of the seed instance from which the instance of the graph was generated.

Also, as part of the prototype implementation's performance assessment, we gathered the time for training as the sum of all the time taken to process each

feature vector. Such metric was computed using the equation 5.3, in which t is the quantity of graphs in the training dataset.

$$Training\ time = \sum_{i=0}^{t_{samples}-1} Training\ time\ for\ i \tag{5.3}$$

Dataset splitting for training and testing

A preliminary experiment was performed to evaluate how models were able to classify mutated instances trained only with information of the seeds. Such approach is considered advantageous, since it is independent from mutated instances' data, which can be hard to lift and generalize given the high variability and diversity of techniques that can be included in engines and virus generation kits.

Since mutator patterns are not previously known, and most metamorphic engines are expected not to be public, ideally, models should present higher accuracy when trained with seed instances only. However, most techniques are well known and can be implemented to generate mutated instances that can be used for training.

Likewise, in a second experiment, models were trained with seeds plus 20% of instances from each family. Herein, we were interested in observing the influence on identification accuracy by including information about the patterns present in mutation engines. We expected increased accuracy, as well as increased training time and dataset size.

The training dataset's feature spaces were reduced based on the features' variance throughout each dataset. Features presenting variance below a given threshold $t \in \{0, 0.2, 0.4, 0.6, 0.8\}$ were excluded. For example, when employing $t = 0$, features that had the same value in all feature vectors of the dataset were removed. Choosing higher values of t meant reducing tolerance to repetitions. For each threshold of variance, a new reduced training dataset was obtained.

Figure 5.2 presents how the initial dataset was split to conform training and testing batches. Blocks inside the training batch were further divided according to the variance threshold employed for feature selection, however, is not shown in the figure. After splitting the initial dataset and applying feature selection on the datasets in the training batch, we obtained 15 datasets for training and 4 datasets for testing.

Classification models creation

To obtain models of classification, a set of machine learning algorithms can be used. Herein, we selected: *a)* Adaboost [117], *b)* Decision Tree (CART) [118],

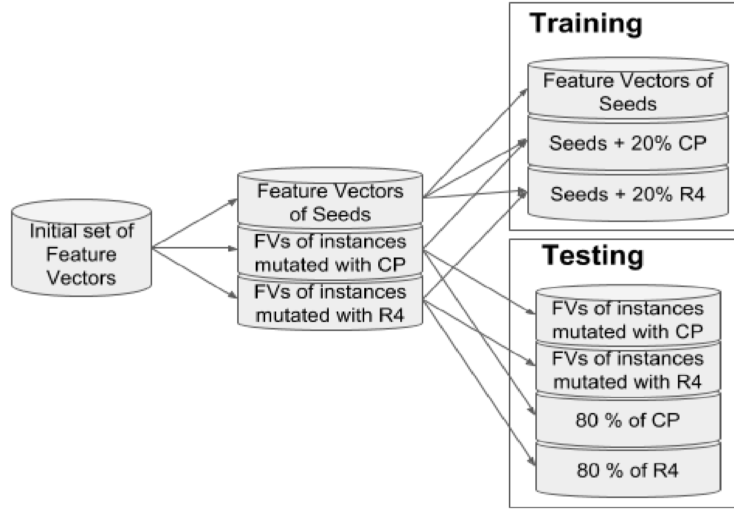

Figure 5.2: Initial dataset splitting for training and testing.

c) KNN [119] with $k \in \{3, 20, 100\}$, *d)* Multilayer Perceptron [120], *e)* Naive Bayes and *f)* Random Forest with 10, 100 and 1000 trees. Such selection was based on the most common algorithms from previous works in the field. In order to execute such algorithms, we employed its implementations from the Python framework for machine learning Sklearn [121] [122].

For each dataset in the training batch and for each machine learning training algorithm used, we created a new model of identification. Figure 5.3 presents the dataset configurations used for model creation.

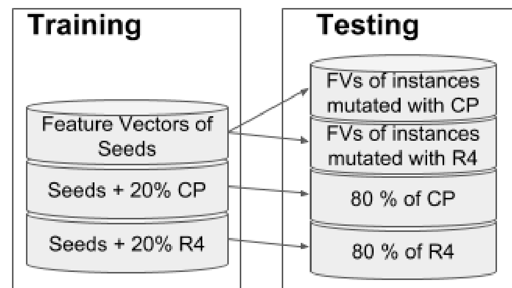

Figure 5.3: Training-testing dataset configurations for model creation

Such configurations can be described as: *a)* training with seeds only and testing with Code Pervertor's generated instances; *b)* training with seeds only and testing with Revert4's mutated instances; *c)* training with seeds plus 20% of Code Pervertor's generated instances and testing with the remaining 80%; *d)* training with seeds plus 20% of Revert4's generated instances and testing with

the remaining 80%. Since we employed 15 datasets for training and 10 machine learning algorithms, afterwards, 150 single models of multi-class classification were obtained.

Table 5.2 provides the configuration parameters used to execute the mentioned learning algorithms. Such configurations were provided to aid other researchers in implementing the proposed methods. However, further studies on such algorithms are necessary to understand the descriptions in the table.

Table 5.2: Configuration of training algorithms used for experimentation

Algorithm	**Parameter name**	**Parameter Description**	**Value**
AdaBoost	Base estimator	The base estimator from which the boosted ensemble is built	Decision trees
	Quantity of estimators	The maximum number of estimators at which boosting is terminated.	50
	Learning rate	Learning rate shrinks the contribution of each classifier by the value this parameter. There is a trade-off between the learning rate and quantity of estimators.	1
Decision Tree (CART)	Criterion	The function to measure the quality of a split	Information gain
	Minimum samples split	The minimum number of samples required to split an internal node	2

	Minimum samples leaf	The minimum number of samples required to be at a leaf node	1
KNN	Number of neighbors	Number of neighbors to use for datapoints grouping	3, 20, 100
	Grouping algorithm	Algorithm used to compute the nearest neighbors	Ball Tree
Multi Layer Perceptron	Hidden Layers	Quantity of full-connected hidden layers	1
	Size of layers	Number of neurons on each hidden layer	100
	Activation function	Activation function for the hidden layer	Rectified Linear Unit
	Iterations	Quantity of iterations performed by the back propagation fitting algorithm	100
Naive Bayes	Algorithm	Naive Bayes approach	Multinomial Naive Bayes
Random Forests	Quantity of estimators	Number of trees in the forest	10,100, 100
	Criterion	The function to measure the quality of a split	Gini
	Minimum samples split	The minimum number of samples required to split an internal node	2
	Minimum samples leaf	The minimum number of samples required to be at a leaf node	1

One-vs-Rest classifiers

For each configuration of training presented in the previous section, we also created an ensemble of multiple binary classifiers following the One-vs-Rest strategy (OvR). For each ensemble, all the classifiers were created with a single training algorithm also called estimator. The same algorithms presented in the previous subsection were employed as estimators. To execute such algorithms, we employed the algorithms' implementations from the Python framework for machine learning Sklearn [121] [122].

The output of each classifier in the OvR ensembles were characterized into one of four possible outcomes: *a)* a true positive, input expresses that the instance belongs to a class and it truly does; *b)* a true negative, the outcome expresses that the instance does not belong to a class and it truly does not; *c)* a false negative, instance is classified as belonging to a class and does not; and *e)* a false negative, input is classified as not belonging to the class and it actually does.

The trained OvR ensembles were evaluated to determine their performance regarding training execution time, as well as prediction accuracy. It is important to know how well the classifiers inside each ensemble discriminated instances into the classes such classifiers were trained for. To address that, we computed the metrics Receiver Operating Characteristics (ROC) and Area Under the Curve (AUC) for each ensemble.

Receiver Operating Characteristics curve (ROC) consists of a two dimensional graph that represents the true positive rate (TPR), also known as sensitivity, on the y axis, against the false positive rate (FPR), also stated as $1 - specificity$, on the x axis. Such metrics are extracted from the model under different probability thresholds, allowing classification quality assessment over all possible operating points. ROC analysis is directly and naturally related to cost/benefit analysis of diagnostic decision making [123].

The area under the ROC curve (AUC) measures the capacity of a classifier to discriminate the classes it was trained for, under different thresholds of probability. As this score approaches one, the better the classifier discriminated each class or the classifier presented higher accuracy [123].

To plot the ROC curves for each OvR ensemble, all probabilities in the superset $P = p_{i,j,x} \forall i \in \{1...K\}, j \in \{1...K\}, x \in \{1..N\}$ of each classifier $c \in \{1...K\}$ in the ensemble e are considered, with K as the quantity of classes and N the quantity of test cases in the testing dataset. For each classifier, the $TPR_{i,t}$ and $FPR_{i,t}$ for each value of t_i $i \in unique(p_{i,j,x})$ is computed as:

$$TPR_{i,t} = \frac{\sum_{j=1}^{K}\sum_{x=1}^{N} TP_{j,x}}{\sum_{j=1}^{K}\sum_{x=1}^{N} TP_{j,x} + \sum_{j=1}^{K}\sum_{x=1}^{N} FN_{j,x}} \tag{5.4}$$

$$FPR_{i,t} = \frac{\sum_{j=1}^{K}\sum_{x=1}^{N} FP_{j,x}}{\sum_{j=1}^{K}\sum_{x=1}^{N} FP_{j,x} + \sum_{j=1}^{K}\sum_{x=1}^{N} TN_{j,x}} \tag{5.5}$$

Such values of t_i are used as a threshold to decide if the outputted $p_{i,j,x}$ is considered for each $g_{i,j,x}$ in the ground truth of the testing dataset, as follows:
a) a TP if $p_{i,j,x} > t_i$ and the expected output $g_{i,j,x} = 1$;
b) a FP if $p_{i,j,x} > t_i$ and the expected output $g_{i,j,x} = 0$;
c) a FN if $p_{i,j,x} < t_i$ and the expected output $g_{i,j,x} = 1$ and
d) a TN if $p_{i,j,x} < t_i$ and the expected output $g_{i,j,x} = 0$.
Al the computed values of $FPR_{i,t}$ and $TPR_{i,t}$ are averaged to obtain the final set of points $o_t = (TPR_t, FPR_t) \forall t \in unique(p_{i,j,x})$ to be plotted in the ROC curve. This approach is known as micro-average ROC plotting [123]. Intuitively speaking, it assesses how well classifiers classify their classes and at what cost (false positives).

Comparison with tier-one commercial antiviruses

Even though metamorphic malware detection is still an issue and threatens information security, it is not a new problem, with several investigations focused on solutions. More advanced techniques in the scientific community are expected to be included in commercial anti-malware tools.

Herein, we studied the capacity of current tier-one commercial malware detections tools to detect instances with metamorphism. Also, we studied how the classification models obtained in this work compare to antiviruses accuracy scores. For that, the same binaries employed to train and test the models generated in this work were submitted to a set of 56 anti-malware tools, among which, were: McAfee [4], Avira [5], Avast[6] and Kavspersky [7].

To perform such operation, we used the platform VirusTotal [8], which allowed

[4] McAfee Antivirus, Online: https://www.mcafee.com/us/index.html
[5] Avira Anivirus, Online: https://www.avira.com
[6] Avast Antivirus, Online: https://www.avast.com
[7] Kaspersky Labs, Online: https://www.kaspersky.com
[8] VirusTotal-Free online virus, malware and URL scanner. Online: https://www.virustotal.com

us to automatically submit several binary instances to the analysis mechanism of 67 well known anti-malware suites. Seed instances were sent before mutation, to ensure tools were provided with each family pattern before metamorphic instances were submitted, so the conditions were similar to the models trained and tested using the methods proposed in this research. Figure 5.4 presents the architecture executed to lift virus detection rate using commercial malware identification tools.

Figure 5.4: Architecture for commercial tools virus detection rate lifting.

5.2 Experimental results

The following subsections present the results of the experiment used to assess performance indicators of the proposed method.

5.2.1 Dataset construction results

Figure 5.5 presents the average times of disassembling by file size, separated according to the mutator employed. On top of each bar it is placed its height as well as the standard deviation with a thin line. Outliers detection is performed employing the clusterization-based algorithm DBSCAN [124] implemented in the python library Scikit Learn [121]. The quantity of outliers for each group of files (according the mutator employed) did not exceed 0.5 % of the group. File sizes are presented in ranges, selected so the time value on each range presents a coefficient of variation below 0.1, this value is calculated as:

$$Coefficient\ of\ variation = \frac{Standard\ deviation(\sigma)}{Mean(\mu)} \tag{5.6}$$

For most of the file ranges in Figure 5.5, disassembling mutated instances took longer than disassembling seeds. This happened because such modifications, applied by Code Pervertor and Revert4, were also intended to dwarf reverse engineering by making the virus more complex. Such complexity was attained by: *a)* trash code insertion; *b)* instructions and data offset randomization and *c)* spaghetti code, which hindered cross references identification. For most of files the disassembling time was lower than 1.5 seconds.

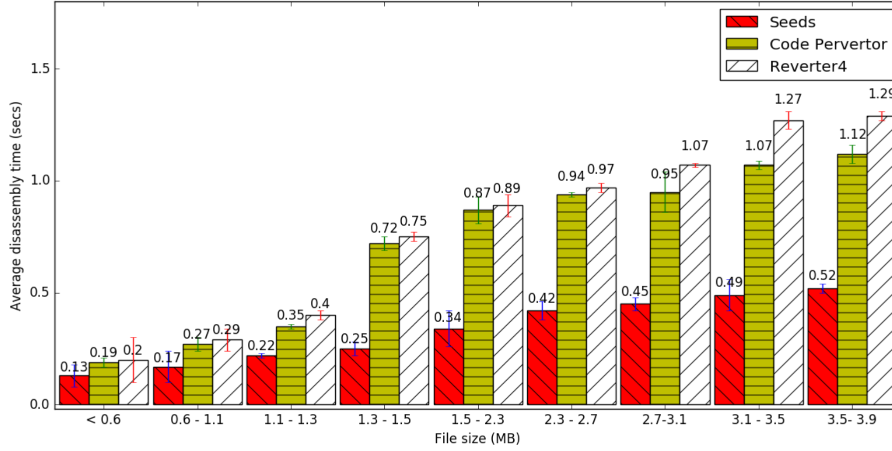

Figure 5.5: Average disassembly time by file size ranges.

Table 5.3: ADDG and CFG construction times for Seeds and Code Pervertor mutated instances.

CFG Size range	**CFG construction time**		**ADDG construction time**	
	Seeds	**CP Generated**	**Seeds**	**CP Generated**
<15	0.15	0.12	0.17	0.13
15 - 37	0.17	0.14	0.2	0.21
37 - 52	0.25	0.21	0.33	0.38
52 - 65	0.37	0.32	0.36	0.4
241 - 253	0.34	0.36	0.87	0.82
253 - 257	0.42	0.41	0.94	0.97
337 - 342	1.25	1.28	1.55	1.65
342 - 350	1.53	1.61	2.52	2.72

Table 5.3 presents times for ADDG and CFG construction vs size of CFG, for Seeds and Code Pervertor generated instances. The ranges presented in the first column of the table represent CFG sizes and were selected so the time value on each range presents a coefficient of variation below 0.1 employing equation 5.6.

Times for ADDG and CFG construction increased with higher CFG sizes. As seen in Algorithm 2 for data flow analysis, DDG construction time depends on the size of the CFG that is inputed to such algorithm, which justifies the steady increase in ADDG construction time alongside the increase of CFG size.

Since ADDG and CFG construction times are directly related to the size of CFG (see Algorithm 2) and the transformations applied by CP does not affect control flow, and by consequence CFG neither, graph's construction time for Seeds and Code Pervertor (CP) generated instances remained similar. There-

fore, the same CFG size ranges were seen for CFGs in Seeds and CP generated instances.

Table 5.4: Graphs' Average Construction Time vs CFG size for Reverter4 generated files.

CFG Size range	CFG construction time	ADDG construction time
<182	0.7	0.17
182 - 307	2.14	3.22
307 - 360	1.33	1.4
360 - 381	1.22	0.68
381 - 620	1.16	0.41
620 - 732	2.44	0.73
732 - 915	1.59	0.65
915 - 1090	1.72	1.71
1090 - 1113	1.87	1.52
1113 - 1220	1.97	1.67
1220 - 1268	1.82	1.21

Table 5.4 presents times for ADDG and CFG construction vs size of CFG, for Revert4 generated instances. The ranges presented in the first column of the table represent CFG sizes and were selected so the time value of each range presents a coefficient of variation below 0.1, using equation 5.6.

ADDGs and CFGs, extracted from Revert4 generated instances, presented more nodes than Seeds and CP generated instances. This happened because R4 applied techniques like junk code insertion, code integration, and spaghetti code that affected control flow by inserting new basic blocks, as well as CFG and DDG. This also affected processing time, since more basic blocks were processed by the disassembler to construct a CFG and by the data flow analysis algorithm for ADDG construction.

Such insertions of basic blocks in CFGs are applied by R4 in a random fashion, affecting the relationship between the size of the CFG and the data dependencies observed for Seeds and CP generated instances (Table 5.3). Since part of the inserted code did not fit in the original code's data dependency chains, newly inserted basic blocks were not processed by the algorithm for ADDG construction.

5.2.2 Dataset indexing results

This section presents the results for the process of dataset indexing. Corresponding our experimental setup , this section presents results and analysis about the:

a) effect of feature selection on features space size; *b)* times of training and testing for single-model multi-class classifiers and One Vs Rest multi-class classifiers; and *c)* accuracy score of models following both proposed approaches.

Feature space reduction

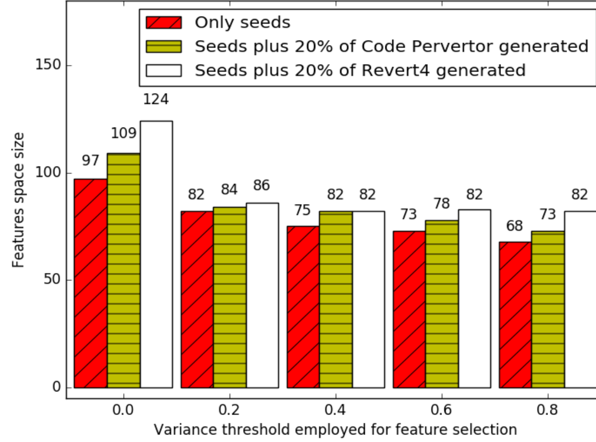

Figure 5.6: Feature space sizes for different datasets vs threshold employed for feature space reduction.

Figure 5.6 presents the impact of feature selection on feature space size, considering: *a)* only seeds; *b)* seeds plus 20% of each family of Code Pervertor generated instances; and *c)* seeds plus 20% of each family of Revert4 generated instances. Each group of columns represents a different threshold t of variance employed.

Before we performed feature selection, all datasets presented 10,080 features, however, as seen in Figure 5.6 this number dropped by more than nine thousands when reduction with the initial variance threshold (value of 0) was applied.

Feature vectors initially presents all possible feature labels, accounting 10080, made of the combinations of the tags employed to classify each part of the instruction on each DDG node and a prefix with three possible values that represents the cardinality of the node. Such process outputs several feature labels that end up being not related with any node in the DDG. Most instructions that are not related with read write operations are eliminated in the process of data flow analysis and does not make it to the ADDG, so does not its related labels. Since such labels are not present in any ADDG its value on the feature vector is always zero for all instances in the dataset, also the variance is zero. Is for that reason that the initial reduction prunes more than nine thousands unused features.

For thresholds of variance above 0.2 in all datasets, feature space sizes reduction eliminated less than ten features from one threshold to the other. For seeds plus 20% of Revert4 generated instances, no features were pruned for values of t above 0.4. Such low variation indicates that it might not be necessary to apply reductions to a value of t above 0.2 if the objective is to enhance performance and accuracy of prediction models. A direct relation exists between features space size with execution time of training algorithms, model's prediction accuracy, and model's prediction execution time, so a variation in such metrics is expected to be in the same proportion.

More features were pruned from the dataset containing only seed instances due to the extra code or trash code inserted by mutators. Code Pervertor inserts randomly selected non-functioning sentences in the spaces between instructions. Revert4 inserts randomly selected groups of instructions with no restriction of space or functioning, without affecting the original code's functionalities. Such mutations resulted in codes with a higher diversity of instructions for mutated instances than for seeds, which lead to less features with zero in all instances for the datasets of mutations than for seeds, and thus, more features pruned for seeds. Since Revert4 extensively used trash code insertions, less features were pruned from its dataset compared to the others.

Classification models were trained for each of the datasets obtained for feature selection from all thresholds. In the following subsections we present the effect of feature reduction on training and testing execution time and accuracy of identification.

Single model multi-class classifiers

This section presents the results of experiments using single-model multi-class classifiers. Training and prediction times, as well as the accuracy presented by the obtained classifiers, are shown and analyzed.

Figure 5.7 plots the accuracy score of each classifier trained, using only information of seed instances and tested in Code Pervertor generated instances. The horizontal axis represents the values of variance threshold t employed for feature space reduction. All scores presented in the figure are below 0.4 and above 0.2, accounting for an undesired result, since more than half of instances went undetected. In this case, classifiers did not learn the graphs' structural patterns well enough.

Figure 5.8 presents accuracy of models trained using seeds and Code Pervertor mutated instances. Including mutated instances for training brought enhancement on classifier's accuracy, reaching the highest value of 87% for Random Forest with 1000 trees. In this case, models are not only provided with structural features of the graphs in each family of malwares, but with information regarding

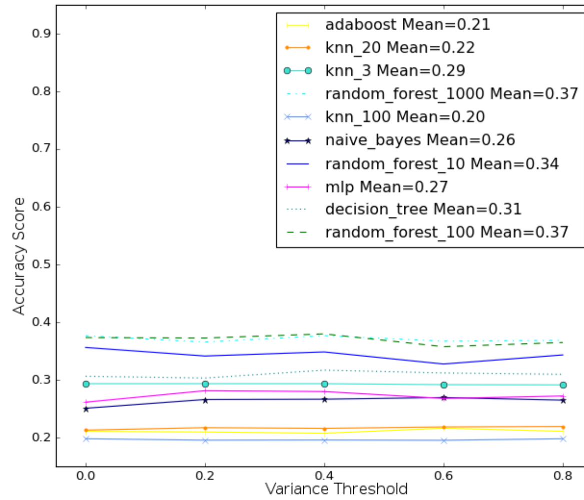

Figure 5.7: Accuracy score for Code Pervertor generated instances using only seeds for training.

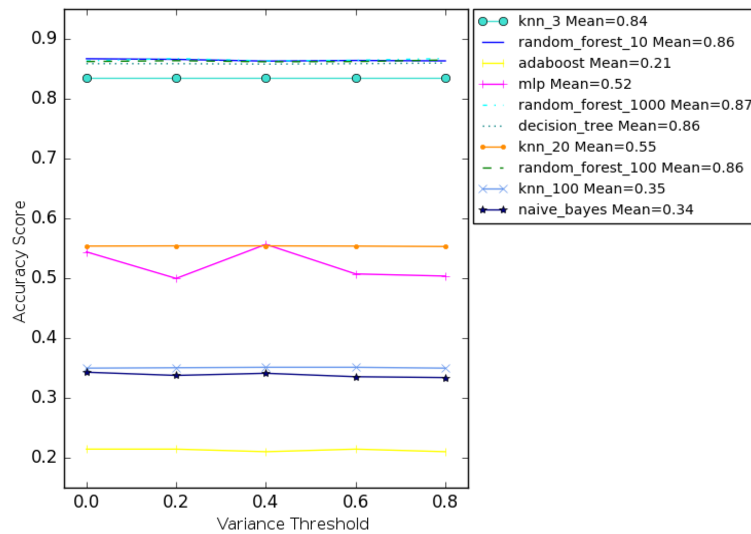

Figure 5.8: Accuracy score for Code Pervertor's generated instances using seeds plus 20% of families for training.

the changes performed by the mutator employed to generate instances in the family.

Code Pervertor applies mutations on the instruction level only, replacing one instruction with a maximum of two others with equivalent functioning. During the process of ADDG construction, the program instructions are transformed into a label with the aim of reducing the effects of single instruction substitution since the label substituting the instruction is expected to be the same for equivalent

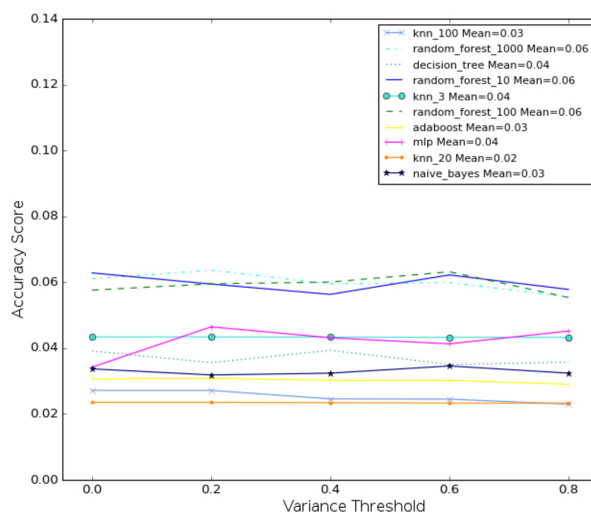

Figure 5.9: Accuracy score of multi-class models trained with Seeds only and tested with Revert4 generated instances graphs' feature vectors.

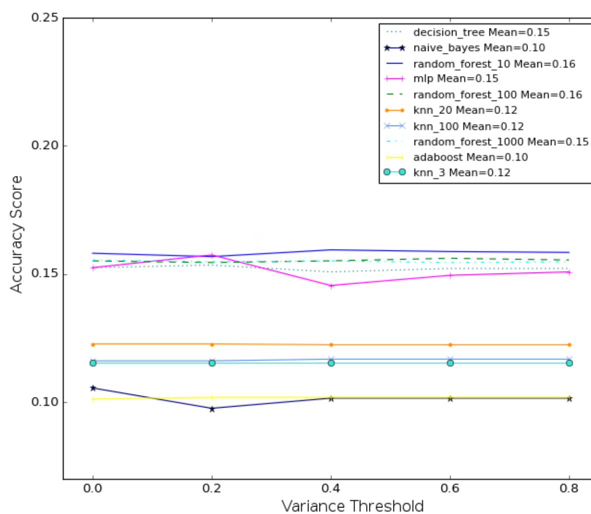

Figure 5.10: Accuracy score of multi-class models trained with Seeds plus 20% of Revert4 instances and tested with 80% of Revert4.

instructions. However, when Code Pervertor transforms one instruction into two, such labeling is ineffectual. This justifies the accuracy increment when considering mutated instances for training, since such traits are provided to the model.

The impact feature space reduction had on accuracy was almost insignificant, i.e accuracy remained mostly invariant for all thresholds in most models, with and without mutated instances for training. However, accuracy scores varied

most with variance threshold t when only seeds were used for training. Likewise, more variation was perceived on feature space reduction for the dataset of seeds than for the other datasets (Figure 5.6).

Each sample in the datasets represent an ADDG of a single function in a malware instance (average 20 functions per malware, see Table 5.1). No structural relation between binaries existed in the seed dataset, and neither among functions belonging to the same binary. In that case, the underlying distribution generating the data can be thought as stochastic and the data samples not have identical distribution. In the case of mutated malware samples, can be established a relationship between instances based on the seed instance from which the mutation was obtained, as well as the mutator employed, with groups of samples that follow the same distribution as the other instances inside the group but differ from distributions in other groups. Such aggregations can contain graphs of the same function from different mutated samples with a common seed.

Due to the stochastic distribution of the datasets used, in which, samples are not continually distributed and do not have an identical distribution, non-parametric models (i.e models that makes no assumption on the population distribution or sample size) like KNN, Random Forest and Decision Trees presented the highest scores of accuracy and parametric models like Adaboost, Naive Bayes and MLP presented the lowest [125]. When considering mutated instances for training, accuracy of parametric models enhanced since there were more samples with identical distribution (i.e similar graphs of the same function in mutated instances with a common seed). For Adaboost, any enhancement was perceived when including mutated samples in the training dataset.

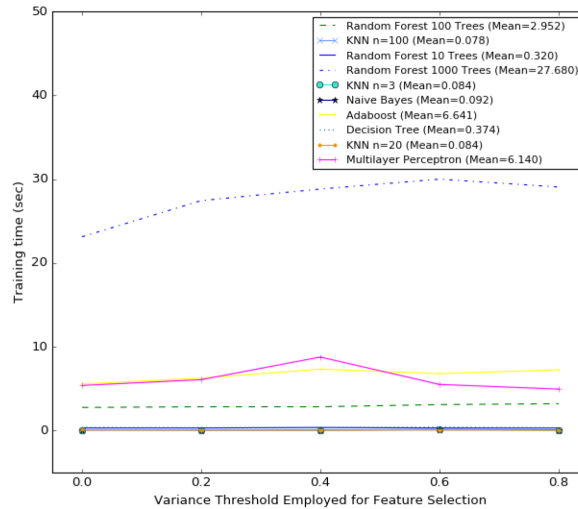

Figure 5.11: Training time with seeds plus 20% of each family of Code Pervertor generated instances.

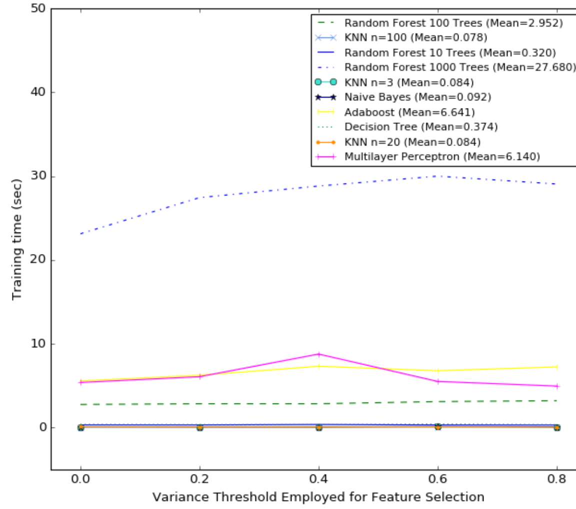

Figure 5.12: Training time with seeds plus 20% of each family of Revert4 generated instances.

Figure 5.9 presents the accuracy of models trained only with seeds and tested with Revert4 generated instances. Figure 5.10 presents scores for when R4 generated instances were considered for training along with seeds. As with the previous dataset configuration, which considered Code Pervertor generated instances, prediction accuracy was enhanced by including mutated instances for training. However, for both training configurations (i.e. only considering seeds and considering seeds plus Revert4 mutated instances) the obtained models presented an accuracy no higher than 20%, which accounted for an undesired result since more than three quarters of samples passed undetected.

The models obtained using the single-model multiclass approach presented more difficulty identifying instances mutated with Revert4. This is mostly due to the high level of metamorphism introduced by this mutator that provided more modifications to graphs than Code Pervertor. Accuracy was also lower for models trained with R4 mutated instances than for models trained with Code Pervertor generated instances. As with the models created employing Code Pervertor generated instances, non-parametric models presented higher accuracy scores. Models were not capable of catching the underlying distribution of graphs for the same function in different samples of the same family, when Revert4 mutated instances were included for training.

Even though considering Revert4 and Code Pervertor mutated instances for training enhanced accuracy, this cannot be seen as a positive result. Since we did not assess variations among mutated instances of the same family and we are not sure if mutators modified all code's functions (which graphs are compared for detection), it is possible the occurrence of graphs in the dataset that are

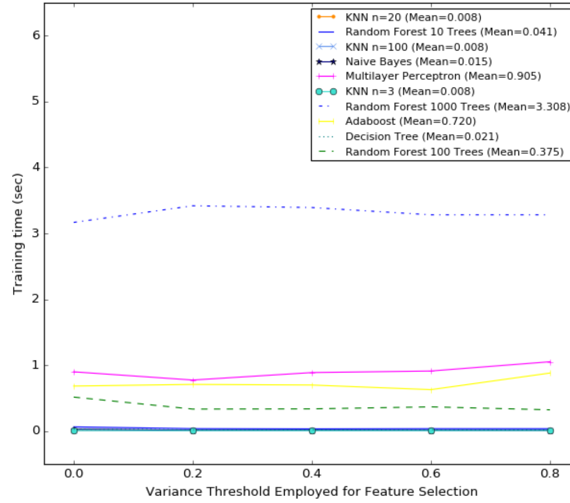

Figure 5.13: Times training with graphs extracted from seeds instances only.

completely equal. Therefore, the enhanced accuracy with the introduction of mutated instances for training and testing could be a result of biased classifiers. Further studies are needed to assess such issue.

Figure 5.13 presents training times using only graphs extracted from seed instances. Even though models created employing Decision Trees, Random Forests with 10 and 100 trees, and Random Forests with 1,000 trees (RM1000) ranked among the most accurate in tests with Revert4 and Code Pervertor datasets, RM1000 took 3 more seconds for training than the others. This also happened when training with 20% of each families of Code Pervertor and Revert4 generated instances, as seen in Figures 5.8, 5.10, 5.11 and 5.12.

No pattern suggests that reducing features space size also reduced training time. For most algorithms, there was no steady linear decrease nor increase of training time in consonance with reduction of feature space. However, in Figure 5.6 feature spaces decreased linearly with higher thresholds.

For training configurations that considered mutated instances, a rather erratic behavior was observed in the variation of training time for MLP, along with the feature space reduction variance threshold used, in which training time did not increase or decrease with decreased feature space size. This was due to the functioning of MLP, in which the delay of training converging to a solution depends on the initial set of weights assigned to each neuron of the net.

When considering all times and accuracy scores for each model and machine learning algorithm, the KNN with $K = 3$ stands out as a good option for tree-based algorithms. In any case, time exceeded one second and accuracy scores were among the highest. However, Random Forests with 10 trees presented the

best trade-off since they did not take longer than a second to train all dataset configurations, and the prediction accuracy was as good or better than all the other algorithms.

One-Vs-Rest Classifiers

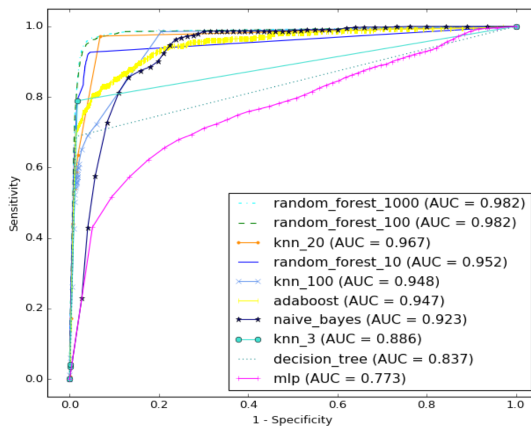

Figure 5.14: ROC Curves for OVR based models employing only seeds for training, using a threshold of feature space reduction of 0 and Code Pervertor instances for testing.

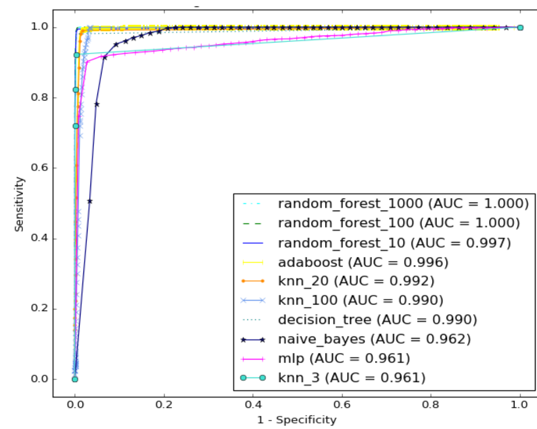

Figure 5.15: ROC Curves for OVR based models employing seeds and 20% of Code Pervertor for training , using a threshold of feature space reduction of 0 and the remaining 80% for testing.

Figures 5.14 and 5.15 presents the ROC curves for models employing only seeds and those trained with seeds plus 20% of each family of Code Pervertor generated instances respectively. As with the multi-class approach, such models trained employing mutated instances presented better performance (i.e higher accuracy and lower training time). Random Forest with 1000 and 100 trees presented AUC score of 1m which means that all submitted instances to the detection mechanism were correctly identified. Most classifiers presented AUC scores above 0.9 even for models trained only with seed instances.

Compared to outcomes in Figures 5.16 and 5.17, in which was employed threshold of variance of 0.8 for features' space reduction, it is not evident how reducing feature space affected AUC score, since for some models, there is an improvement and other presented worst results, however differences does not exceed 0.5. On the other hand as shown in Figure 5.24, in which are presented times of training for all models, employing only seeds, exists a tendency to time decrease along with the reduction of the quantity of features employed for training. It indicates that in this case reducing the feature space was favorable since reduced training time without reducing accuracy. ROC curves for other values of

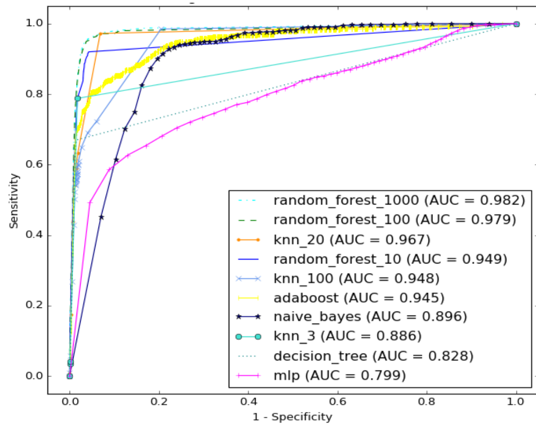

Figure 5.16: ROC Curves for OVR based models employing only seeds for training, using threshold of feature space reduction of 0.8 and Code Pervertor for testing.

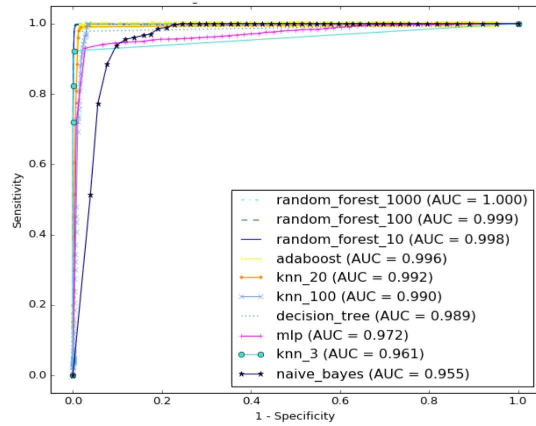

Figure 5.17: ROC Curves for OVR based models employing seeds and 20% of Code Pervertor for training, using threshold of feature space reduction of 0.8 and the remaining 80% for testing.

variance threshold employed for feature space reduction are not presented since no variation on the AUC score was experienced.

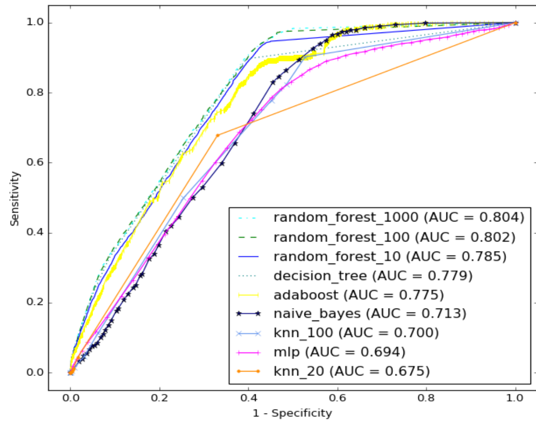

Figure 5.18: ROC Curves for OVR based models employing only seeds for training, using threshold of feature space reduction 0 and Rever4 generated instances for testing.

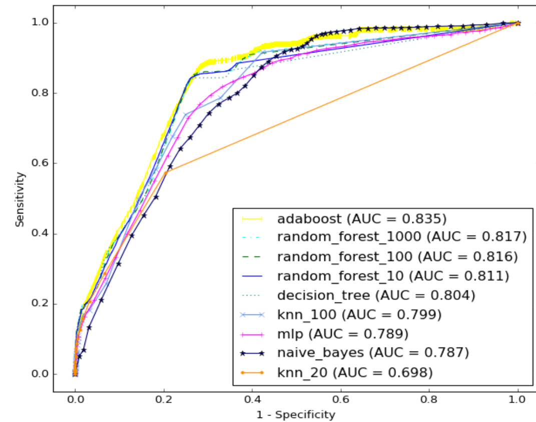

Figure 5.19: ROC Curves for OVR based models employing seeds and 20% of Revert4 generated instances for training, using threshold of feature space reduction 0 and remaining 80% for testing.

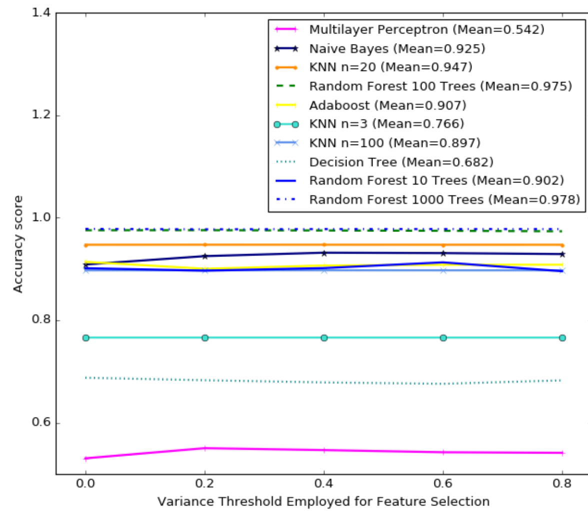

Figure 5.20: Accuracy of models trained with seeds only and tested with Code Pervertor generated instances.

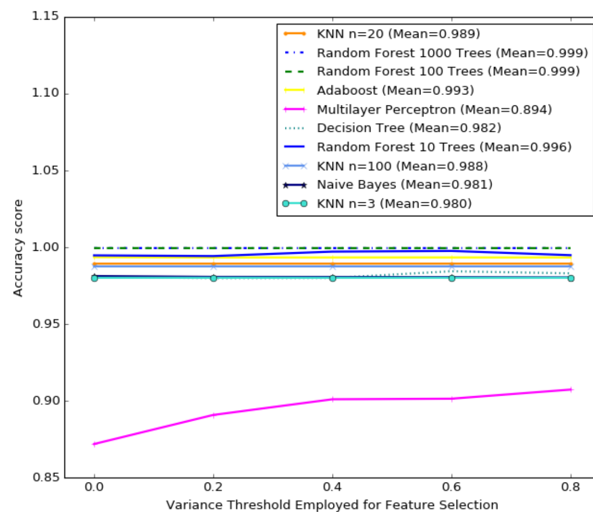

Figure 5.21: Accuracy of models trained with seeds plus 20% of families of Code Pervertor generated instances and tested with the remaining 80%.

Figures 5.18 and 5.19 present the ROC curves and AUC scores for models trained following the One vs Rest approach, using only seeds (Figure 5.18) and seeds plus 20% of each family of Revert4 generated instances. Models trained following the OvR approach presented better sample identification score than models trained with the single-model classifiers approach. For most points on the ROC curve, the true positive rate was higher than the false positive rate, which means that when considering the outcomes' average for all graphs in a

binary to predict its seed instance, chances are higher that classification is a true positive. However, just like models trained following the single-model classifier approach, classifiers in the OvR approach also presented more difficulty to detect Revert4 generated instances than for Code Pervertor generated instances.

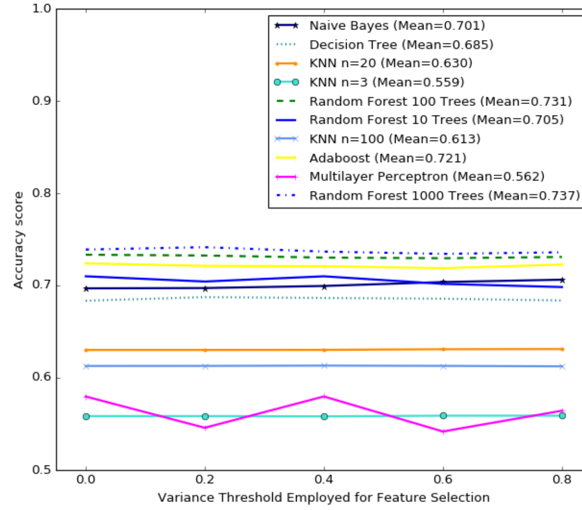

Figure 5.22: Accuracy of models trained with seeds only and tested with Revert4 generated instances.

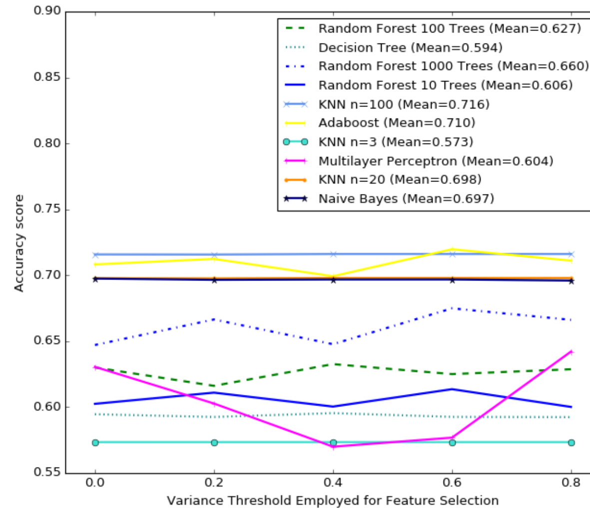

Figure 5.23: Accuracy of models trained with seeds plus 20% of families of Revert4 generated instances and tested with the remaining 80%.

Figures 5.20, 5.21, 5.22 and 5.23 present the accuracy scores of models trained following the OvR approach, for all the dataset's training and testing configurations. As with the single-model classifier approach, non-parametric training

algorithms also presented the highest accuracy scores. Even though the ground truth was modified into binarized vectors to comply with the OvR requirements, the independent variables of the training dataset was not modified and did not present an identical distribution, which explains why the non-parametric models presented better accuracy than parametric models.

Adaboost takes the place of KNN with $k = 3$ among the first in the single model multi-class approach. These models created with Adaboost were collocated among the first presenting similar results to models created with non-parametric algorithms. Similarly, when considering time of training, Adaboost is a better option since it presented a lower execution time than non-parametric algorithms.

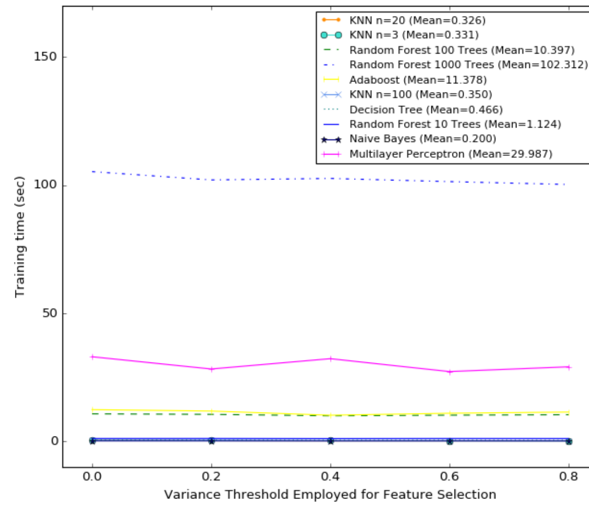

Figure 5.24: Training times with graphs extracted from seeds instances only.

Figure 5.24 presents training times using seeds only, for all the values of threshold employed for features' space reduction. Even though models trained with Random Forests of 1000 trees presented the highest accuracy scores , training time was also the highest, which is a disadvantage. Models created following the OvR approach presented higher accuracy for all models compared to models trained through the single classifier multi-class approach. Furthermore, times were higher, however, for all models except Random Forests with 1000 trees the time difference did not exceed 12 seconds, which can still be considered practical. For Random Forests with 1000 trees, training took 97 seconds longer when using only seeds in the OvR approach, as compared to the training time for the single classifier multi-class approach with the same algorithm.

In general, Adaboost presented lower training times than the algorithms used to create models with similar accuracy, for all configurations in the One vs Rest approach. On the other hand, Multilayer Perceptron presented some of the high-

est training times and the accuracy of its respective models were among the lowest.

Models following the OvR approach presented higher accuracy scores than models of the single classifier multi-class approach, however, training times were higher. Even though accuracy scores were higher with the OvR approach, the main drawback compared to multi-class was training time. While the algorithms took less than 15 seconds to train with the multi-class approach, primarily occurring between 0 and 5 seconds, the OvR approach experienced times above 500 an to 600 seconds. Such behavior is expected, since the OvR one classifier had to be trained for each class, and binarization had to be performed.

5.2.3 Comparison with tier-one commercial antiviruses

The analysis in this subsection shows the capacity of current tier-one commercial antivirus' for metamorphic malware identification based on their detection rate score. Such scores are compared to the accuracy scores presented by models from the proposed methodology, solely considering models trained with seed instances only. This analysis provides a comparison between our proposed methods and current state-of-art detection approaches, as such tools are created and maintained by leading representatives of malware detection worldwide.

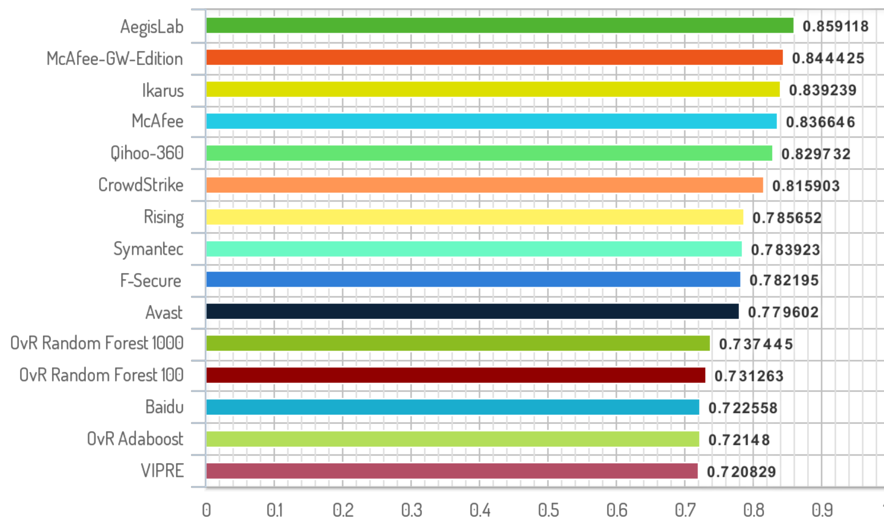

Figure 5.25: Top 15 ranked models and antivirus for instances mutated with Revert4.

Figures 5.25 and 5.26 present the detection rate scores of the top 15 ranked between models created in this work and antivirus, for Revert4 and Code Pervertor's generated instances respectively. Most detectors presented better results

detecting the Code Pervertor mutated instances. Revert4 applied more complex mutation techniques and provoked a higher deviation in generated mutated instances from the structural patterns of its seeds when compared to Code Pervertor. Instances generated with Revert4 hindered detection so that most antiviruses' accuracy were below 80%, even though seeds were already known by all tools when this experiment was executed. The best accuracy obtained for Revert4 generated mutations was 85%, which means that 99 instances passed undetected.

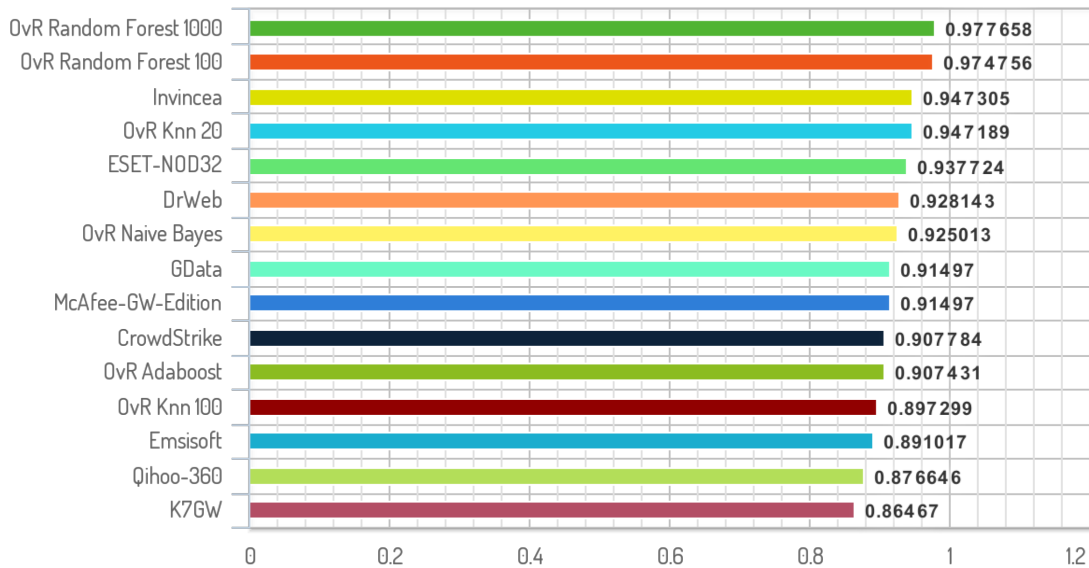

Figure 5.26: Top 15 ranked models and antivirus for instances mutated with Code Pervertor.

Most antiviruses presented a score above 90% detecting instances generated with Code Pervertor. The highest score among the antivirus was from Invincea, which scored 94%. For the other antiviruses, their scores decreased by 1% or more along with their ranking position. Two-hundred and fourteen instances went undetected by the highest scoring antivirus.

For Code Pervertor generated instances detection, 6 of the models trained herein ranked among the top 15. Random Forests with 1000 and 100 trees ranked first with 97%, which was 3% higher than the best ranked antivirus. Accuracy score enhanced 0.003% for Random Forest based models by adding trees to the model. Training times were 89% higher for models created with 1000 trees, however, accuracy remained similar. That said, if Random Forest with 1000 trees is not considered in the comparison, Random Forest based models would still rank first, 3% higher than the highest ranked antivirus. Therefore, Random Forests with 100 trees can be considered a better option.

Even though 3 of the generated models ranked among the top 15 most accurate for detection of Revert4 generated instances, none of the detection models trained in this work overcame 73% detecting Revert4 instances. On the other hand, the models trained in this research presented higher scores than the best ranked antiviruses for detecting Code Pervertor instances. The difference in accuracy among mutators presented by models was mostly due to the complexity of structural patterns added by Revert4 compared to Code Pervertor, which only applies simple and multiple instructions substitution. Such difference can also be seen in antivirus' detection accuracy. The quantity of instances that remained undetected by the tier-one antivirus tools showed that code metamorphism for malware detection hindering is still an unsolved issue.

Chapter 6

Final thoughts and future works

Herein, we proposed a method for metamorphic malware identification based on indexing annotated data dependency graph databases extracted from binary codes. A prototype implementation of the proposed methods was tested against mutated malware samples and presented detection accuracy within the top 15 when compared to 56 current commercial anti-malware tools. Furthermore, the execution times presented by each of the steps in these methods can be considered practical.

Malware instances from the wild were used to test the efficiency of our proposed methods. Mostly, found instances' size did not exceed 4 megabytes, which was expected to occur since malware creators likely prefer smaller sized samples that are easier to distribute. Tools used for reverse engineering presented execution times of less than5 seconds for processing most instances. Construction time of graphs (i.e CFGs and ADDGs) remained below 2 seconds for most samples. Subjectively, such times can be considered practical and demonstrate the feasibility of using reverse engineering for malware analysis.

We also proposed a scheme for executable binary representation based on annotated data dependency graphs, which has not previously been seen. Using ADDGs brings binary executable analysis to malware identification information regarding the semantics associated with instructions in the assembly code of binaries.

Applying feature space reduction techniques eliminated more than 95% of the initial set of features, reduced dataset size, lowered execution time of model's training, and enhanced accuracy of models. These techniques also demonstrated that most labels in the set of possibilities are not employed. In general, labels representing instructions that did not perform operations of read and write received values of zero for all features vectors, which was expected since ADDGs are intended to represent data flow patterns only.

Models that presented the highest accuracy of detection followed the multiple

classifier One vs Rest approach for multi-class classification, employing Random Forests. Likewise, Adaboost and KNN ranked among the most accurate and presented low training execution times. Models created with the Random Forests with 100 trees algorithm presented similar accuracy as models created with Random Forest with 1000 trees. Furthermore, training times were about 90% lower, which makes this algorithm the best configuration of the employed models given the favorable trade-off between training time and accuracy score.

Models' training was executed in an environment with high computing capacity given its 128 GB of RAM and 61 processing units. Such step does not need to be executed in the same environment where detection is to be applied. Malware identification can be performed in a wide variety of equipments, with different power capacities, from IoT devices to super-computers. After a model is trained (preferably in an environment with enough resources) it can be serialized and distributed for detection, like current endpoint malware scanners in which signature bases are constructed and later distributed to endpoint systems to be employed in the moment of detection.

Training times for index induction significantly increased with the addition of instances to the dataset. Datasets in the wild present billions of malware instances, which is much higher than the datasets used in this work. Currently, exponential growth of data is not limited to information security and is seen in many other areas. How much more time is needed to train classification models with bigger datasets depends on the time complexity of the algorithm used and the processing capabilities of the procedures executed. In order to address performance enhancement, future research should focus on strategies based on parallelization, on-line training, and knowledge transfer.

Even though metamorphism is not a recent issue, with several studies addressing its detection, current commercial anti-malware suites have difficulties detecting mutated instances, as shown in our results . The detection of such tools decreases as complexity in mutations increase, and even with simple modifications, in less than 10% of a program, its maliciousness can be hidden from those tools. Such decrease in accuracy also occurred in the detection models obtained in this work.

Accuracy scores obtained in this work were competitive with current commercial anti-malware suites, however, several instances were misclassified. In order to enhance detection accuracy, future studies should address: *a)* effects of metamorphism on ADDGs; *b)* labeling strategies using more accurate opcode instruction classifications to enhance normalization; *c)* comparison of feature space reduction methods; *d)* comparison of models created with other machine learning algorithms not presented herein.

Bibliography

[1] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of Big Data challenges and analytical methods," *Journal of Business Research*, vol. 70, pp. 263–286, 2017.

[2] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International Journal of Communication Systems*, vol. 25, no. 9, p. 1101, 2012.

[3] L. Piwek, D. A. Ellis, S. Andrews, and A. Joinson, "The rise of consumer health wearables: promises and barriers," *Journal PLoS Medicine*, vol. 13, no. 2, p. e1001953, 2016.

[4] P. Wood, B. Nahorney, K. Chandrasekar, S. Wallace, and K. Haley, "ISTR April 2016," tech. rep., 2016.

[5] K. Wood, Paul and Nahorney, Benjamin and Chandrasekar, Kavitha and Wallace, Scott and Haley, "2015 Internet Security Threat Report," Tech. Rep. April, 2015.

[6] ENISA, "Enisa threat landscape report 2017, top 15 top cyber-threats and trends," tech. rep., European Union Agency For Network and Information Security, 2018.

[7] "2017 Avira Threat Landscape," tech. rep., 2017. Accessed: 2017-12-03.

[8] N. Minihane, F. Moreno, E. Peterson, and R. Samani, "McAfee Labs Threat Report 2017," Tech. Rep. December, 2017.

[9] E. Lange-Ionatamishvili, S. Svetoka, and K. Geers, "Strategic communications and social media in the russia ukraine conflict," *Cyber War in Perspective: Russian Aggression against Ukraine, Tallinn: NATO CCD COE Publications*, 2015.

[10] M. C. Libicki, D. Senty, and J. Pollak, *Hackers Wanted: An Examination of the Cybersecurity Labor Market.* RAND Corporation, 2014.

[11] Cisco, "Cisco 2015 Annual Security Report," tech. rep., 2015.

[12] B. Kumar and S. Yadav, "Storageless credentials and secure login," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, p. 55, ACM, 2016.

[13] B. Kitts, J. Y. Zhang, G. Wu, W. Brandi, J. Beasley, K. Morrill, J. Ettedgui, S. Siddhartha, H. Yuan, F. Gao, *et al.*, "Click fraud detection: adversarial pattern recognition over 5 years at microsoft," in *Real World Data Mining Applications*, pp. 181–201, Springer, 2015.

[14] C. Cao and J. Caverlee, "Detecting spam urls in social media via behavioral analysis," in *European Conference on Information Retrieval*, pp. 703–714, Springer, 2015.

[15] M. Carminati, R. Caron, F. Maggi, I. Epifani, and S. Zanero, "Banksealer: A decision support system for online banking fraud analysis and investigation," *Computers & Security*, vol. 53, pp. 175–186, 2015.

[16] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in ddos attacks: trends and challenges," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2242–2270, 2015.

[17] J. Aycock, *Computer viruses and malware*, vol. 22. Springer Science & Business Media, 2006.

[18] Ernst and Young (EY), "Cybersecurity regained: preparing to face cyber attacks," tech. rep., 2017.

[19] O. Sukwong, H. Kim, and J. Hoe, "Commercial antivirus software effectiveness: an empirical study," *Computer*, vol. 44, no. 3, pp. 63–70, 2011.

[20] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh, *Recent Advances in Intrusion Detection*, vol. 5758 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 10 2009.

[21] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, *et al.*, "Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 165–187, Springer, 2016.

[22] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pp. 3–9, IEEE, 2015.

[23] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, p. 4, 2016.

[24] C. W. Dilshan Keragala, "Detecting Malware and Sandbox Evasion Techniques," tech. rep., January 2016.

[25] F. Cohen, "Computer Viruses : theory and experiments.," *Comput. Secur.*, vol. 6, no. 1, pp. 22–35, 1987.

[26] K. Chandrasekar, G. Cleary, O. Cox, and L. Hon, "Symantec 2017 internet security threat report," tech. rep., 2017.

[27] J. Antonakos, A. Davidi, C. De La Fuente, and D. Sachin, "2017 trustwave global security report," tech. rep., 2017.

[28] S. Attaluri, "Detecting metamorphic viruses using profile hidden markov models," *Diss. San Jose State University*, 2007.

[29] D. Baysa, R. M. Low, and M. Stamp, "Structural entropy and metamorphic malware," *Journal of Computer Virology and Hacking Techniques*, vol. 9, pp. 179–192, 4 2013.

[30] B. B. Rad, M. Masrom, and S. Ibrahim, "Opcodes histogram for classifying metamorphic portable executables malware," in *2012 International Conference on E-Learning and E-Technologies in Education, ICEEE 2012*, pp. 209–213, IEEE, sep 2012.

[31] G. Shanmugam, R. M. Low, and M. Stamp, "Simple substitution distance and metamorphic detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, pp. 159–170, 3 2013.

[32] G. Canfora, A. N. Iannaccone, and C. A. Visaggio, "Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics," *Journal of Computer Virology and Hacking Techniques*, vol. 10, pp. 11–27, 9 2013.

[33] J. Kuriakose and P. Vinod, "Ranked linear discriminant analysis features for metamorphic malware detection," in *2014 IEEE International Advance Computing Conference (IACC)*, pp. 112–117, IEEE, 2 2014.

[34] S. Alam, R. N. Horspool, and I. Traore, "MAIL: Malware Analysis Intermediate Language," in *Proceedings of the 6th International Conference on*

Security of Information and Networks - SIN '13, (New York, New York, USA), pp. 233–240, ACM Press, 2013.

[35] S. Alam, I. Traore, and I. Sogukpinar, "Annotated Control Flow Graph for Metamorphic Malware Detection," *The Computer Journal*, vol. 58, pp. 2608–2621, 10 2015.

[36] H. R. Ranjbar, M. Sadeghzadeh, and A. Keshavarz, "A novel data mining method for malware detection," *Journal of Theoretical and Applied Information Technology*, vol. 70, no. 1, pp. 43–51, 2014.

[37] S. Alam, I. Sogukpinar, I. Traore, and R. Nigel Horspool, "Sliding window and control flow weight for metamorphic malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 11, pp. 75–88, May 2015.

[38] G. Martins Breves, R. D. Freitas, and E. Souto, "Virtual Structures and Heterogeneous Nodes in Dependency Graphs for Detecting Metamorphic Malware," 2014.

[39] K. Kim and B.-R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, p. 1211, 2010.

[40] J. Choi, Y. Yoon, and B.-R. Moon, "An efficient genetic algorithm for subgraph isomorphism," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pp. 361–368, ACM, 2012.

[41] J. Harris, J. Hirst, and M. Mossinghoff, *Combinatorics and Graph Theory*. Undergraduate Texts in Mathematics, Springer New York, 2008.

[42] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, pp. 1–19, ACM, 1970.

[43] C. Irniger, "Graph Matching–Filtering Databases of Graphs Using Machine Learning Techniques," Master's thesis, Institut für Informatik und angewandte Mathematik, Universität Bern, 2005.

[44] S. Alam, R. Horspool, I. Traore, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," *Computers & Security*, vol. 48, pp. 212–233, 2 2015.

[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[46] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*, (New York, New York, USA), p. 872, ACM Press, 2006.

[47] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda, "Towards automated malware creation: Code generation and code integration," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, (New York, NY, USA), pp. 157–160, ACM, 2014.

[48] G. McGraw and G. Morrisett, "Attacking Malicious Code: A Report to the Infosec Research Council," *IEEE Software*, vol. 17, no. 5, pp. 33–41, 2000.

[49] P. Vinod, V. Laxmi, M. S. Gaur, and G. Chauhan, "Detecting malicious files using non-signature-based methods," *International Journal of Information and Computer Security*, vol. 6, no. 3, pp. 199–240, 2014.

[50] E. Skoudis and L. Zeltser, *Malware: Fighting Malicious Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[51] D. Shinder, "The Pros and Cons of Behavioral Based, Signature Based and Whitelist Based Security." http://www.windowsecurity.com/articles-tutorials/misc{_}network{_}security/Pros-Cons-Behavioral-Signature-Whitelist-Security.html, 2018.

[52] I. A. Saeed, A. Selamat, A. M. A. Abuagoub, and S. B. Abdulaziz, "A Survey on Malware and Malware Detection Systems," *analysis*, vol. 3, no. 10, pp. 13–17, 2013.

[53] I. E. Kamarudin, S. A. M. Sharif, and T. Herawan, "On analysis and effectiveness of signature based in detecting metamorphic virus," *International Journal of Security and its Applications*, vol. 7, no. 4, pp. 375–386, 2013.

[54] S. G. Perlman and R. van der Laan, "System for streaming databases serving real-time applications used through streaming interactive video," July 18 2017. US Patent 9,707,481.

[55] Y. Oyama, "Trends of anti-analysis operations of malwares observed in api call logs," *Journal of Computer Virology and Hacking Techniques*, pp. 1–17, 2017.

[56] 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.

[57] E. A. Daoud, I. Jebril, and B. Zaqaibeh, “Computer virus strategies and detection methods,” *Int. J. Open Problems Compt.*, 2008.

[58] J. Jones, “The state of Web exploit kits,” *BlackHat Las Vegas*, 2012.

[59] P. O’Kane, S. Sezer, and K. McLaughlin, “Obfuscation: The hidden malware,” *Security & Privacy, IEEE*, vol. 9, no. 5, pp. 41–47, 2011.

[60] B. B. Rad, M. Masrom, and S. Ibrahim, “Opcodes histogram for classifying metamorphic portable executables malware,” in *2012 International Conference on E-Learning and E-Technologies in Education (ICEEE)*, pp. 209–213, IEEE, 9 2012.

[61] Q. Zhang, *Polymorphic and metamorphic malware detection.* North Carolina State University, 2008.

[62] 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, 11 2010.

[63] J.-M. Borello, “Are current antivirus programs able to detect complex metamorphic malware? An empirical evaluation,” 2009.

[64] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *Journal in Computer Virology*, vol. 2, pp. 211–229, 11 2006.

[65] A. H. Toderici and M. Stamp, “Chi-squared distance and metamorphic virus detection,” *Journal of Computer Virology and Hacking Techniques*, vol. 9, pp. 1–14, 9 2012.

[66] B. B. Rad, M. Masrom, and S. Ibrahim, “Camouflage in malware: from encryption to metamorphism,” 2012.

[67] T. Tamboli, T. H. Austin, and M. Stamp, “Metamorphic code generation from LLVM bytecode,” *Journal of Computer Virology and Hacking Techniques*, vol. 10, pp. 177–187, 11 2013.

[68] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *Journal in Computer Virology*, vol. 2, pp. 211–229, nov 2006.

[69] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq, “Evolvable malware,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 1569–1576, ACM, 2009.

[70] S. Choudhary and M. D. Vidyarthi, "A Simple Method for Detection of Metamorphic Malware using Dynamic Analysis and Text Mining," *Procedia Computer Science*, vol. 54, pp. 265–270, 2015.

[71] J. Kuriakose and P. Vinod, "Unknown metamorphic malware detection: Modelling with fewer relevant features and robust feature selection techniques," *IAENG International Journal of Computer Science*, vol. 42, no. 2, pp. 139–151, 2015.

[72] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," 2007.

[73] Z. Markov and I. Russell, "An introduction to the WEKA data mining system," *ACM SIGCSE Bulletin*, vol. 38, no. 3, pp. 367–368, 2006.

[74] L. Rabiner and B. Juang, "An introduction to hidden markov models," *ieee assp magazine*, vol. 3, no. 1, pp. 4–16, 1986.

[75] S. G. Dastidar, S. Mandal, F. A. Barbhuiya, and S. Nandi, "Detecting metamorphic virus using Hidden Markov Model and genetic algorithm," 2012.

[76] S. Attaluri, S. McGhee, and M. Stamp, "Profile hidden Markov models and metamorphic virus detection," *Journal in Computer Virology*, vol. 5, pp. 151–169, 2008.

[77] K. Xin, G. Li, Z. Qin, and Q. Zhang, "Malware Detection in Smartphone Using Hidden Markov Model," *International Conference on Multimedia Information Networking and Security*, pp. 857–860, 2012.

[78] A. H. Toderici and M. Stamp, "Chi-squared distance and metamorphic virus detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 1, pp. 1–14, 2013.

[79] P. Zhang and C. G. Cassandras, "An improved forward algorithm for optimal control of a class of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 47, no. 10, pp. 1735–1739, 2002.

[80] G. D. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[81] Y. Zhang, D. Zhao, and J. Liu, "The application of baum-welch algorithm in multistep attack," *The Scientific World Journal*, vol. 2014, 2014.

[82] P. Desai, "A highly metamorphic virus generator," *Intelligence*, vol. 1, no. 4, pp. 402–427, 2011.

[83] T. Singh, F. Di Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "Support vector machines and malware detection," *Journal of Computer Virology and Hacking Techniques*, 2015.

[84] S. Suthaharan, "Support vector machine," in *Machine learning models and algorithms for big data classification*, pp. 207–235, Springer, 2016.

[85] J. T. Kent, "Information gain and a general measure of correlation," *Biometrika*, vol. 70, no. 1, pp. 163–173, 1983.

[86] C. V. Liţă, D. Cosovan, and D. Gavriluţ, "Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in upa packers," *Journal of Computer Virology and Hacking Techniques*, pp. 1–20.

[87] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. San Francisco, CA, USA: No Starch Press, 2008.

[88] H. Parvin, B. Minaei, H. Karshenas, and A. Beigi, "A new n-gram feature extraction-selection method for malicious code," in *International Conference on Adaptive and Natural Computing Algorithms*, pp. 98–107, Springer, 2011.

[89] K. Fukunaga and S. Ando, "The optimum nonlinear features for a scatter criterion in discriminant analysis," *IEEE Transactions on Information Theory*, vol. 23, no. 4, pp. 453–459, 1977.

[90] R. Bro and A. K. Smilde, "Principal component analysis," *Analytical Methods*, vol. 6, no. 9, pp. 2812–2831, 2014.

[91] D. Aha and D. Kibler, "Instance-based learning algorithms," *Machine Learning*, vol. 6, pp. 37–66, 1991.

[92] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[93] M. Collins, R. E. Schapire, and Y. Singer, "Logistic regression, adaboost and bregman distances," *Machine Learning Journal*, vol. 48, no. 1-3, pp. 253–285, 2002.

[94] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[95] Z.-Q. Zeng, H.-B. Yu, H.-R. Xu, Y.-Q. Xie, and J. Gao, "Fast training support vector machines using parallel sequential minimal optimization," in *Intelligent System and Knowledge Engineering, 2008. ISKE 2008. 3rd International Conference on*, vol. 1, pp. 997–1001, IEEE, 2008.

[96] M. Eskandari and S. Hashemi, "A graph mining approach for detecting unknown malwares," *Journal of Visual Languages & Computing*, vol. 23, no. 3, pp. 154–162, 2012.

[97] G. Sidorov, H. Gómez-Adorno, I. Markov, D. Pinto, and N. Loya, "Computing text similarity using tree edit distance," in *Fuzzy Information Processing Society (NAFIPS) held jointly with 2015 5th World Conference on Soft Computing (WConSC), 2015 Annual Conference of the North American*, pp. 1–4, IEEE, 2015.

[98] A. Rényi, "On measures of entropy and information," tech. rep., Hungarian Academy of Sciences, Budapest Hungary, 1961.

[99] C. Torrence and G. P. Compo, "A practical guide to wavelet analysis," *Bulletin of the American Meteorological society*, vol. 79, no. 1, pp. 61–78, 1998.

[100] D. Bruschi, L. Martignoni, and M. Monga, "Code Normalization for Self-Mutating Malware," *IEEE Security and Privacy Magazine*, vol. 5, pp. 46–54, 3 2007.

[101] Q. Zhang and D. S. Reeves, "MetaAware: Identifying Metamorphic Malware," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 411–420, IEEE, 12 2007.

[102] 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.

[103] R. Paredes and E. Chávez, "Using the k-nearest neighbor graph for proximity searching in metric spaces," in *International Symposium on String Processing and Information Retrieval*, pp. 127–138, Springer, 2005.

[104] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, (New York, New York, USA), p. 1970, ACM Press, 2010.

[105] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology*, vol. 8, pp. 37–52, 4 2012.

[106] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), pp. 872–881, ACM, 2006.

[107] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism algorithms for the matching of chemical structures," *Journal of computer-aided molecular design*, vol. 16, no. 7, pp. 521–533, 2002.

[108] D. Yuan, P. Mitra, and C. L. Giles, "Mining and indexing graphs for supergraph search," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 829–840, 2013.

[109] Radare2, "Radare2 github repository." https://github.com/radare/radare2, 2017.

[110] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[111] K. D. Cooper, T. J. Harvey, and K. Kennedy, "Iterative data-flow analysis, revisited," tech. rep., 2004.

[112] K. Lejska, "X86 opcode and instruction reference." http://ref.x86asm.net/index.html, 2017. Accessed: 2017-12-30.

[113] C. Nguyen Anh Quynh, "Capstone: next generation disassembly framework." http://www.capstone-engine.org/BHUSA2014-capstone.pdf, 2014.

[114] E. Keogh and A. Mueen, "Curse of dimensionality," in *Encyclopedia of Machine Learning and Data Mining*, pp. 314–315, Springer, 2017.

[115] N. Sánchez-Maroño, A. Alonso-Betanzos, P. García-González, and V. Bolón-Canedo, "Multiclass classifiers vs multiple binary classifiers using filters for feature selection," in *Neural networks (ijcnn), the 2010 international joint conference on*, pp. 1–8, IEEE, 2010.

[116] E. Eilam, *Reversing: Secrets of Reverse Engineering*. New York, NY, USA: John Wiley & Sons, Inc., 2005.

[117] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[118] W.-Y. Loh, "Classification and regression trees," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[119] N. Bhatia *et al.*, "Survey of nearest neighbor techniques," *arXiv preprint arXiv:1007.0085*, 2010.

[120] G. E. Hinton, "Connectionist learning procedures," in *Machine Learning, Volume III*, pp. 555–610, Elsevier, 1990.

[121] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[122] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[123] D. J. Hand and R. J. Till, "A simple generalisation of the area under the roc curve for multiple class classification problems," *Mach. Learn.*, vol. 45, pp. 171–186, Oct. 2001.

[124] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pp. 226–231, AAAI Press, 1996.

[125] M. Dundar, B. Krishnapuram, J. Bi, and R. B. Rao, "Learning classifiers when the training data is not iid," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, (San Francisco, CA, USA), pp. 756–761, Morgan Kaufmann Publishers Inc., 2007.

[126] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, no. 2, pp. 137–144, 2015.