DETECTION OF MALICIOUS PORTABLE EXECUTABLES

BY

SYEDA MOMINA TABISH

REGISTRATION NUMBER: 2006-NUST-MS-PHD-IT-25

SUPERVISOR

DR FAUZAN MIRZA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

IN

DEPARTMENT OF COMPUTING

NUST School Of Electrical Engineering and Computer Science

National University of Sciences and Technology

H-12, Islamabad, Pakistan

April, 2010

MASTER OF SCIENCE THESIS

OF

SYEDA MOMINA TABISH

REGISTRATION NUMBER: 2006-NUST-MS-PHD-IT-25

APPROVED:

Thesis Committee:

Major Professor Dr. Fauzan Mirza

Dr. Muddassar Farooq

Dr. Syed Ali Khayam

Ali Sajjad

DEAN OF THE GRADUATE SCHOOL

NUST School of Electrical Engineering and Computer Sciences

National University of Sciences and Technology

H-12, Islamabad, Pakistan

April, 2010

ABSTRACT

Malicious portable executables (PE) pose a significant threat to Microsoft Windows operating systems. State-of-the-art antivirus software detect the malicious PE files using signature-based approaches or manually generated heuristics. However, the size of signature database, the signature matching overhead and the cost of manual heuristic generation cannot scale with an exponential increase in the number of malicious PE files. In this work we present a data mining approach to automatically extract distinguishing features and classify unseen malicious PE files. The distinguishing features are extracted using the structural information provided in the standard PE file format for executables, DLLs and object files used in Microsoft Windows operating systems. The eventual classification is performed using well-known data mining algorithms. Our executable classification methodology is twofold; firstly we classify benign and malicious executables and secondly we classify malicious executables as a function of their payload. We evaluated PE-Miner on two malware collections, VX Heavens dataset and Malfease dataset, that contain 11 thousand and 5 thousand malicious PE files respectively. The results of our experiments show that PE-Miner achieves more than 99% detection rate with less than 0.5% false alarm rate for distinguishing between the benign and malicious executables. Furthermore, it achieves an average detection rate of 90% with an average false alarm rate of less than 5% for categorizing the malicious executables as a function of their payload. It is important to emphasize that PE-Miner has low processing overheads and takes only 0.244 seconds on the average to scan a given PE file.

ACKNOWLEDGMENTS

First of all I would like to thank Allah Almighty for giving me the strength and courage to undertake this project. Without His will absolutely nothing is possible.

I am exceedingly grateful to my supervisor Dr. Fauzan Mirza for giving me courage and trusting me. I derived a lot of motivation, courage and inspiration by working with him. I am also thankful to him because I was provided with all the resources I needed to work on our project.

I would like to thank Dr. Muddassar Farooq, Dr. Syed Ali Khayam, and Mr. Ali Sajjad for their help in modeling and software implementation. Without their immense help and support this project would not have been possible. I learnt a lot from them while working on the thesis both technically and professionally. I am ever so grateful to them for providing me with thorough and articulate advices. I would also like to thank Next Generation Intelligent Networks Research Center (nexGIN RC) [1] and National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan, for supporting this work.

I would like to thank Mr. Zubair Shafiq for helping with the implementation of the proposed algorithm. I would further like to thank Mr. Shehzad Mughal and Mr. Muhammad Umer for useful discussions regarding design and implementation of the techniques proposed in this thesis.

I am also grateful to the Director General School of Electrical Engineering and Computer Sciences, Dr. Arshad Ali and the respected faculty members for providing me with knowledge and wisdom over the past few years. Without their efforts I would not have been able to undertake this thesis.

[1]This work is part of the Artificial Immune System Based Intrusion Detection System project at nexGIN RC. The IP rights of this work are held with Next Generation Intelligent Networks Research Center (nexGIN RC), FAST-NU, Islamabad, Pakistan

DEDICATION

I DEDICATE THIS THESIS TO MY PARENTS AND MY BEST FRIENDS (AYESHA, MUNAZZA, IRAM, ISBAH AND MEMO).

TABLE OF CONTENTS

Page

Page

LIST OF ABBREVIATIONS

API	Application Programming Interface
AUC	Area Under the Curve
COTS	Commercial Off the Shelf
DLL	Dynamic Link Library
IBK	Instance Based Learner
J48	Decision Tree
JRip	Rule Based Learner
KM	Kolter Maloof
ML	Machine Learning
NB	Naive Bayes
PE	Portable Executable
ROC	Receiver Operating Curve
SVM	Support Vector Machines
WEKA	Wakaito Environment for Knowledge Acquisition

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A number of non-signature based malware detection techniques have been proposed recently. These techniques mostly use heuristic analysis, or behavior analysis or a combination of both to detect malware. Such techniques are being actively investigated because of their ability to detect zero-day malware without any a priori knowledge about them. Some of them have been even integrated into the existing Commercial Off the Shelf Anti Virus (COTS AV) products, but achieved only limited success [4], [5]. The most important shortcoming of these techniques is that they are not *realtime deployable*[1]. We, therefore, believe that the domain of *realtime deployable* non-signature based malware detection techniques is still open to novel research.

Non-signature based malware detection techniques are primarily criticized because of two inherent problems: (1) high *fp* rate, and (2) large processing overheads. Consequently, COTS AV products mostly utilize signature based detection schemes that provide low *fp* rate and have acceptable processing overheads. But it is a well-known fact that signature based malware detection schemes are unable to detect *zero-day* malware. We cite two reports to highlight the alarming rate at which new malware is proliferating. The first report is by Symantec that shows an increase of 468% in the number of malware from 2006 to 2007 [6]. The second report shows that the number of malware produced in 2007 alone was more than the total number of malware produced in the last 20 years [7]. These surveys suggest that signature based techniques cannot keep abreast with the security challenges

[1]We define a technique as *realtime deployable* if it has three properties: (1) a *tp* rate of approximately 1, (2) an *fp* rate of approximately 0, and (3) the file scanning time is comparable to existing COTS AV.

of the new millennium because not only the size of the signatures' database will exponentially increase but also the time of matching signatures. These bottlenecks are even more relevant on resource constrained smart phones and mobile devices [8]. We, therefore, envision that in near future signature based malware detection schemes will not be able to meet the criterion of *realtime deployable* as well.

We argue that a malware detection scheme which is *realtime deployable* should use an intelligent but simple static analysis technique. In this work we propose a framework, called *PE-Miner*, which uses novel *structural features* to efficiently detect malicious PE files. PE is a file format which is standardized by the Microsoft Windows operating systems for executables, dynamically linked libraries (DLL) and object files. We follow a threefold research methodology in our static analysis: (1) identify a set of structural features for PE files, which is computable in realtime, (2) use an efficient preprocessor for removing redundancy in the features' set, and (3) select an efficient data mining algorithm for final classification. Consequently, our proposed framework consists of three modules, the feature extraction module, the feature selection/preprocessing module, and the detection module inline with above-mentioned research methodology.

We have evaluated our proposed detection framework on two independently collected malware datasets with different statistics. The first malware dataset is the VX Heavens Virus collection consisting of more than ten thousand malicious PE files [9]. The second malware dataset is the Malfease dataset, which contains more than five thousand malicious PE files [10]. We also collected more than one thousand benign PE files from our lab, which we use in conjunction with both malware datasets in our study. The results of our experiments show that our PE-miner framework achieves more than 99% detection rate with less than 0.5% false alarm rate for distinguishing between the benign and malicious executables.

Further, our framework takes on average only 0.244 seconds to scan a given PE file. Therefore, we can conclude that our proposed PE-Miner is *realtime deployable*, and consequently it can be easily integrated into existing COTS AV products.

Remember that PE-Miner framework can also categorize the malicious executables as a function of their payload. This analysis is of great value for system administrators and malware forensic experts. PE-Miner achieves a detection rate of 90% with an average false alarm rate of less than 5% for categorizing the malicious executables as a function of their payload.

We have also compared PE-Miner with other promising malware detection schemes proposed by Perdisci et al [1], Schultz et al [2] and Kolter et al [3]. These techniques use some variation of n-gram analysis for malware detection. PE-Miner provides better detection accuracy[2] with significantly smaller processing overheads compared with these approaches. We believe that the superior performance of PE-Miner is attributable to a rich set of novel PE format specific structural features, which provides relevant information for better detection accuracy [12]. In comparison, n-gram based techniques are more suitable for classification of loosely structured data, therefore, they fail to exploit format specific structural information of a PE file. As a result, they provide lower detection rates and higher processing overheads as compared to PE-Miner.

The remaining thesis is organized as follows. In Chapter 2, we discuss the architecture of our proposed framework. In Chapter 3, we briefly explain the work that has been done in malware detection and in Chapter 4, we discuss the techniques closely related with our proposed approach. In Chapter 5 we present description and statistics of the executable datasets used in our study. Chapter 6

[2]Throughout this text, the terms *detection accuracy* and *Area Under ROC Curve (AUC)* are used interchangeably. The AUC ($0 \leq \text{AUC} \leq 1$) is used as a yardstick to determine the detection accuracy. Higher values of AUC mean high *tp* rate and low *fp* rate [11]. At AUC = 1, *tp* rate = 1 and *fp* rate = 0.

contains the discussion on the results of our experiments, evaluation of our scheme on cross datasets, and describe the limitations of PE-Miner and potential solutions. We finally conclude the thesis in Chapter 7, with an outlook of our future work.

CHAPTER 2

PROPOSED FRAMEWORK

In this chapter we discuss our proposed PE-Miner framework. We set the following strict requirements on our PE-Miner framework to ensure that our research is enacted with a product development cycle that has a short time-to-market:

- It must be a pure non-signature based framework with an ability to detect zero-day malicious PE files. Moreover, it should have the nice-to-have feature of categorizing malware as a function of their payload.

- It must be *realtime deployable*. To this end, we say that it should have more than 99% *tp* rate and less than 1% *fp* rate. We argue that it is still a challenge for non-signature based techniques to achieve these true and false positive rates. Moreover, its time to scan a PE file must be comparable to those of existing COTS AV products.

- Its design must be modular that allows for the plug-n-play design philosophy. This feature will be useful in customizing the detection framework to specific requirements, such as porting it to the file formats used by other operating systems

We have evolved the final modular architecture of our PE-Miner framework in a systematic questioned oriented engineering fashion. In our research, we raised following relevant questions, analyzed their potential solutions and finally selected the best one through extensive empirical studies.

1. Which PE format specific features can be statically extracted from PE files to distinguish between benign and malicious files? Moreover, are the format

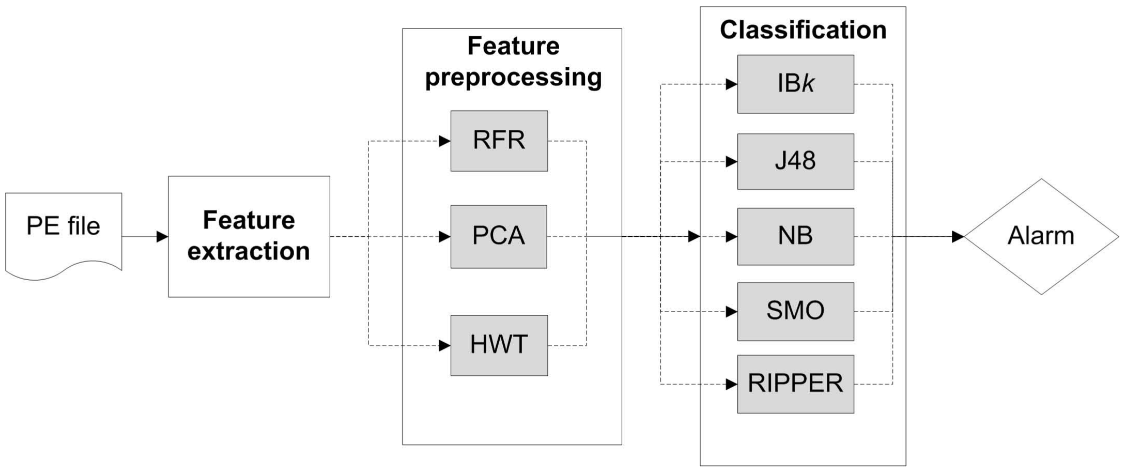

Figure 1. The architecture of our PE-Miner framework

specific features better than the existing n-grams or string-based features in terms of detection accuracy and efficiency?

2. Do we need to deploy preprocessors on the features' set? If yes then which preprocessors are best suited for the raw features' set?

3. Which are the best back-end classification algorithms in terms of detection accuracy and processing overheads.

Our PE-Miner framework consists of three main modules inline with the above-mentioned vision: (1) feature extraction, (2) feature preprocessing, and (3) classification (see Figure 1). We now discuss each module separately.

2.1 Feature Extraction

Let us revisit the PE file format [13] before we start discussing the structural features used in our features' set. A PE file consists of a PE file header, a section table (section headers) followed by the sections' data. The PE file header consists of a MS DOS stub, a PE file signature, a COFF (Common Object File Format) header and an optional header. It contains important information about a file such as the number of sections, the size of the stack and the heap, etc. The section

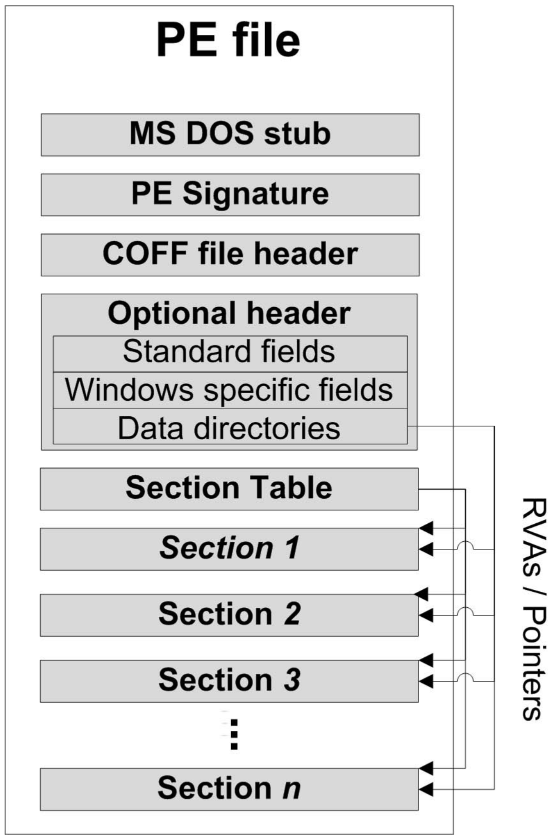

Figure 2. The PE file format

table contains important information about the sections that follow it, such as their name, offset and size. These sections contain the actual data such as code, initialized data, exports, imports and resources [13], [14].

Figure 2 shows an overview of the PE file format [13], [14]. It is important to note that the section table contains Relative Virtual Addresses (RVAs) and the pointers to the start of every section. On the other hand, the data directories in optional header contain references to various tables (such as import, export, resource, etc.) present in different sections. These references, if appropriately analyzed, can provide useful information.

We believe that this structural information about a PE file should be leveraged to extract features that have the potential to achieve our primary and secondary objectives. Using this principle, we statically extract a set of large number of fea-

Table 1. List of the features extracted from PE files

Feature Description	Type	Quantity
DLLs referred	binary	73
COFF file header	integer	7
Optional header – standard fields	integer	9
Optional header – Windows specific fields	integer	22
Optional header – data directories	integer	30
.text section – header fields	integer	9
.data section – header fields	integer	9
.rsrc section – header fields	integer	9
Resource directory table & resources	integer	21
Total		189

tures from a given PE file[1]. These features are summarized in Table 1[2]. In the discussion below, we first intuitively argue about the features that have the potential to distinguish between benign and malicious files. We then show interesting observations derived from the executable datasets used in our empirical studies.

DLLs referred. The list of DLLs referred in an executable effectively provides an overview of its functionality. For example, if an executable calls `WINSOCK.DLL` or `WSOCK.DLL` then it is expected to perform network related activities. However, there can be exceptions to this assumption as well. In [2], Schultz et al have used the conjunction of DLL names, with a similar functionality, as binary features. The results of their experiments show that this feature helps to attain reasonable detection accuracy. However, our pilot experimental studies have revealed that using them as individual binary features can reveal more information, and hence can be more helpful in detecting malicious PE files. In this study, we have used 73 core functionality DLLs as features. Table 2 shows the mean feature values for the two DLLs. Interestingly, `WSOCK32.DLL` and `WININET.DLL` are used by a majority of backdoors, nukers, flooders, hacktools, worms and trojans to access the resources on the network and the Internet. Therefore, the applications mis-

[1]A well-known Microsoft Visual C++ utility, called `dumpbin`, dumps the relevant information which is present inside a given PE file [15]. Another freely available utility, called `pedump`, also does the required task [16].

[2]The details of the datasets and their categorization are available in chapter 5.

Table 2. Mean values of the extracted features. The bold values in every row highlight interesting outliers.

Dataset	**VX Heavens**									**Malfease**
Name of Feature	**Benign**	**Backdoor + Sniffer**	**Constructor + Virtool**	**DoS + Nuker**	**Flooder**	**Exploit + Hacktool**	**Worm**	**Trojan**	**Virus**	-
`WSOCK32.DLL`	0.037	**0.503**	0.038	**0.188**	**0.353**	**0.261**	**0.562**	**0.242**	0.053	0.065
`WININET.DLL`	0.073	**0.132**	0.009	0.013	0.04	**0.141**	0.004	**0.103**	0.019	0.086
Number of Symbols	**430.2**	2.0x10^6	14.7	59.4	25.8	3.5x10^6	38.8	4.1x10^6	1.0x10^6	2.7x10^7
Major Linker Version	**4.7**	14.4	11.2	14.1	12.1	12.3	18.7	12.2	19.3	6.5
Size of Initialized Data (x10^5)	**4.4**	1.1	0.5	0.4	0.8	0.7	0.4	0.4	0.1	0.6
Major Image Version	**163.1**	1.6	6.3	0.4	0.6	11.2	0.3	6.0	53.6	0.2
Checksum (x10^5)	**8.8**	0.6	1.2	0.7	1.2	4.6	0.2	0.4	**83.2**	1.0
DLL Characteristics	**5.8x10^3**	0.0	0.0	0.0	0.0	24.9	0.0	3.1	230.8	18.7
Size Export Table (x10^2)	**13.7**	2.4	1.7	**14.1**	5.0	0.3	1.2	2.1	0.9	0.05
Size Import Table (x10^2)	5.8	**19.2**	6.1	7.9	**20.8**	7.1	**23.4**	**10.3**	6.2	4.7
Size Resource Table (x10^4)	**32.6**	5.5	1.5	1.4	6.2	1.0	2.6	2.2	0.5	5.9
Size Exception Table	12.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**3.5**
.data Size of Raw Data (x10^3)	**25.2**	8.4	5.6	6.3	6.0	7.9	6.1	5.5	6.7	22.1
Number of Cursors	**14.5**	6.4	6.7	7.4	6.1	5.9	5.8	6.0	3.0	6.8
Number of Bitmaps	**12.6**	1.2	0.0	1.0	0.6	0.7	1.2	1.4	2.4	0.5
Number of Icons	**17.6**	2.5	1.9	2.7	2.0	2.1	1.8	1.9	4.5	2.2
Number of Dialogs	**10.9**	3.2	1.5	3.2	1.5	2.0	1.9	1.7	2.2	2.3
Number of Group Cursors	**11.6**	6.0	6.6	7.2	5.8	5.8	5.4	5.7	2.7	6.7
Number of Group Icons	**4.1**	1.0	0.7	1.0	0.8	0.7	0.5	0.7	1.5	0.6

using these DLLs might provide a strong indication of a possible covert network activity.

COFF file header. The COFF file header contains important information such as the type of the machine for which the file is intended, the nature of the file (DLL, EXE, or OBJ etc.), the number of sections and the number of symbols. It is interesting to note in Table 2 that a reasonable number of symbols are present in benign executables. The malicious executables, however, either contain too many or too few symbols.

Optional header: standard fields. The interesting information in the standard fields of the optional header include the linker version used to create an executable, the size of the code, the size of the initialized data, the size of the uninitialized data and the address of the entry point. Table 2 shows that the values of major linker version and the size of the initialized data have a significant difference in the benign and malicious executables. The size of the initialized data in benign executables is usually significantly higher compared to those of the malicious executables.

Optional header: Windows specific fields. The Windows specific fields of the optional header include information about the operating system version, the image version, the checksum, the size of the stack and the heap. It can be seen in Table 2 that the values of fields such as the major image version, the checksum and the DLL characteristics are usually set to zero in the malicious executables. In comparison, their values are significantly higher in the benign executables.

Optional header: data directories. The data directories of the optional header provide pointers to the actual data present in the sections following it. It includes the information about export, import, resource, exception, debug, certificate and base relocation tables. Therefore, it effectively provides a summary of the contents of an executable. Table 2 highlights that the size of the export table is higher for the benign executables and nukers as compared to those of other malicious executables. Another interesting observation in Table 2 is that the backdoors, flooders, worms and trojans mostly have a bigger import table size. It can be intuitively argued that they usually import network functionalities which increases the size of their import table. The size of the resource table, on the other hand, is higher for the benign executables as compared to those of the malicious executables. The exception table is mostly absent in the malicious executables.

Section headers. The section headers provide important characteristics of a section such as its address, size, number of relocations and line numbers. In this study, we have only considered text, data and resource sections because they are commonly present in the executables. Note that the size of the .data section (if present) is relatively higher for the benign executables.

Resource directory table & resources. The resource directory table provides an overview of the resources that are present in the resource section of an executable file. We consider the actual count of various types of resources that are present in the resource section of an executable file. The typical examples of resources include cursors, bitmaps, icons, menus, dialogs, fonts, group cursors and user defined resources. Intuitively and as shown in Table 2, the number of these resources is relatively higher for the benign executables.

2.2 Feature Selection/Preprocessing

We have now identified our features' set that consists of a number of statically computable features (189 to be precise) based on the structural information of the PE files. It is possible that some of the features might not convey useful information in a particular scenario, therefore, it makes sense to remove or combine them with other similar features to reduce the dimensionality of our input feature space.

Moreover, this preprocessing on the raw extracted features' set also reduces the processing overheads in training and testing of classifiers, and can possibly also improve the detection accuracy of classifiers. In this study, we have used three well-known features' selection/preprocessing filters. We provide their short descriptions in the following text. More details can be found in [17].

2.2.1 Redundant Feature Removal (RFR)

We apply this filter to remove those features that do not vary at all or show significantly large variation i.e. they have approximately uniform-random behavior. Consequently, this filter removes all features that have either constant values or show a variance above a threshold or both. In our framework, we set this threshold parameter at 0.99.

2.2.2 Principal Component Analysis (PCA)

We apply PCA filter to remove/combine correlated features for dimensionality reduction. A relevant parameter present in most implementations of PCA is the *variance covered*. This parameter determines the minimum number of principal components required to cover the given variance. In this study, we set this parameter to be 0.95. We have set a conservative value for this parameter to avoid loss of any important information that may later become useful in classification.

2.2.3 Haar Wavelet Transform (HWT)

The wavelet transform has also been used for dimensionality reduction. The wavelet transform technique has been extensively used in the image compression but is never evaluated in the malware detection domain. The Haar wavelet is one of the simplest wavelets and is known to provide reasonable accuracy. The application of Haar wavelet transform requires the data to be normalized. Therefore, we have passed the data through *normalize* filter before applying HWT.

2.3 Classification

Once the dimensionality of the input features' set is reduced by applying one of the above-mentioned preprocessing filters, it is given as an input to the well-known data mining algorithms for classification. Most of these algorithms require a training and testing phase. In this study we have used five classifiers: (1) instance based learner (IBk) [18], (2) decision tree (J48) [19], (3) Naïve Bayes [20], (4) inductive rule learner (RIPPER) [21], and (5) support vector machines using sequential minimal optimization (SMO) [22]. We use the implementations of following classification algorithms which are available in Wakaito Environment for Knowledge Acquisition (WEKA) [17]. We provide their brief overview as follows:

2.3.1 Instance Based Learner (IBk)

The instance based classifier (IBk) is the simplest of all algorithms used in our comparative study. The classification is done on the basis of a majority vote of k neighboring instances [18]. In our study, we use default parameters for instance based classifier (IBk) implemented in WEKA. We use the value of k as 5 and the window size is set to be 0 allowing maximum number of instances in the training pool with no replacements. We have not used distance weighting method. No internal cross validation is used in the algorithm to determine the value of k.

2.3.2 Decision Tree (J48)

Decision trees are usually used to map observations about an item to conclusions about the item's target value using some predictive models [19]. They are very easy to understand and are efficient in terms of time especially on large datasets. They can be applied on both numerical and categorical data, and statistical validation of the results is also possible.

We use C4.5 decision tree (J48) that is implemented in WEKA. We use the default parameters for J48. We do not utilize binary splits on nominal attributes for building trees because all of our selected features are numeric except the class labels. The confidence factor for pruning is set to 0.25, where lower values lead to more pruning. The minimum number of instance per leaf equals 2. The number of folds of training data is set to 3, where one fold is used for pruning and the rest are used for growing the tree.

2.3.3 Naïve Bayes (NB)

Naïve Bayes is a simple probabilistic classifier assuming naïve independence among the features i.e. the presence or absence of a feature does not affect any other feature [20]. The algorithm works effectively and efficiently when trained in a supervised learning environment. Due to its inherent simple structure it often gives very good performance in complex real world scenarios. The maximum likelihood technique is used for parameter estimation of Naïve Bayes models.

We use the default parameters for Naïve Bayes in WEKA. We neither use kernel estimator functions nor numeric attributes for supervised discretization that converts numeric attributes to nominal ones.

2.3.4 Inductive Rule Learner (RIPPER)

We also use a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER), proposed by William W. Cohen as an optimization of IREP [21]. We chose rule based learners due to their inherent simplicity that results in a better understanding of their learner model. RIPPER, performs quite efficiently on large noisy datasets with hundreds of thousands of examples. It caters for missing attributes, numerical variables and multiple classes. The algorithm works by initially making a detection model composed of rules which are improved iteratively using different heuristic techniques. The constructed rule set is used to classify the test cases.

We use default parameters for RIPPER in WEKA. The training set is divided into 3 types, one is used for pruning and the rest are used for growing the rules. The minimum total weight of the instances in the rule is 2. The optimization of the rule set is done twice.

2.3.5 Support Vector Machines using Sequential Minimal Optimization (SMO)

The concept of Support Vector Machine (SVM), invented by Vladimir Vapnik, in its simplest form aims to develop a hyperplane that separates a set of positive samples from a set of negative samples with a maximum margin [22]. For linear separability of the problem space, SVMs use a kernel function for mapping training data to a higher-dimensional space. It is a quadratic programming (QP) problem, which can be very time consuming on larges dataset like ours. However, we are using SMO (Sequential Minimal Optimization) which is a fast and efficient SVM training algorithm implemented in WEKA [22]. It breaks the QP problem into smaller subsets which are later solved analytically reducing the processing overheads.

We use the default parameters for SMO in WEKA with a linear kernel. The complexity parameter is set at 1. The threshold on round-off error is set to 10^{-12}. We are using training data to generate a logistic model. The tolerance parameter for the experiments is set at 0.0010.

CHAPTER 3
BACKGROUND STUDIES

Malware detection has been one of the most important research area in information security recently. A lot of work has been done in order to cater for this problem [[2], [3], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47]]. The research done in this field can be broadly divided in 2 sub categories based on the technique and methodology used to detect malware i.e, anomaly based detection and signature based detection. Anomaly based detection is done on the basis of deviation from the learned normal or benign model. Another kind of anomaly based detection is specification based detection in which there is some rule set or specification which is considered as valid behavior. The programs which does not follow the rule set or specification are deemed as malicious and vice versa. The other subcategory of malware detection is signature based detection, which uses signature or exact snippets of bytes for the detection of malicious files. This characterization is the key to signature based detection and the main limitation to it as well.

The Figure 3 shows the hierarchy of the malware detection techniques. These techniques are further divided into three sub-categories on the basis of their approaches, i.e. static, dynamic and hybrid. Static approach analyzes the structural features and syntax of the program for example exact byte values in the file. Dynamic approach analyzes the run time information of the program for example the API calls it is requesting. Hybrid approaches combine both the static and dynamic analysis for malware detection.

We shall briefly explain the sub-categories for host based malware detection and some of the literature work as their examples. Quite a lot of work has been

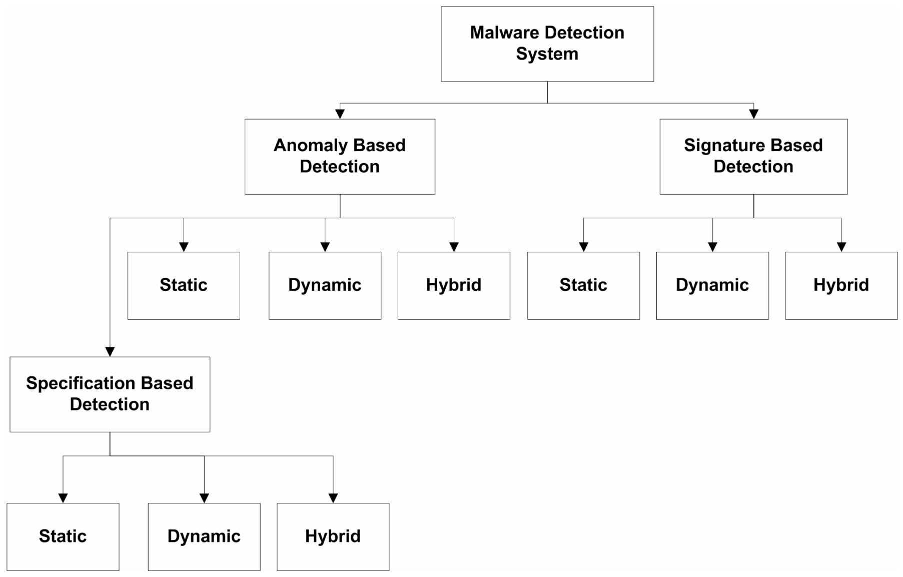

Figure 3. Malware Detection System

done in specification based detection, so although it is a part of anomaly based detection we are explaining it separately.

3.1 Anomaly based detection

Anomaly based detection system is a system for detecting anomalies by analyzing system activity and classifying it as either normal or anomalous. The classification is based on heuristics or rules, rather than patterns or signatures, and will detect any type of misuse that falls out of normal system operation. There are two phases in an anomaly based approach, training phase and testing phase. In training phase, a normal or a benign model is constructed using the normal samples, while in testing phase, the suspicious samples are classified as malicious if they deviate from the normal model obtained from the training and vice versa. The main advantage of the anomaly based detection is their ability to detect zero-day malware, as they learn the behavior rather than the exact byte values of malware. However, the high false alarm rate associated with these systems can not be ignored, as well as the learning of what specific features to use in the model is an important consideration. The selection of features/behaviors is the basis of the numerous approaches that have been proposed in the literature.

3.1.1 Static anomaly based detection

Static anomaly based detection system leverage structural characteristics and properties of the program under observation to detect malware. The main advantage of static analysis is that the malware does not get the chance to execute.

Christopher et al. [35]

Current system call pattern based intrusion detection systems have become highly sophisticated. This includes checking the value of the PC. This makes the job of the attacker harder. However, mimicry attacks which are exploits carefully crafted with valid sequence of system calls, can deceive these systems. One way of countering this is to validate return addresses on the call stack. This only allows the attacker's code to execute one system call before it loses control. For the attacker's code to regain control after a system call the attacker can change the function pointer to a library function or influence the application to alter the return address of a function. This however requires a complex static analysis of the application binary. This is made easy by 'Symbolic Execution'. The execution state and path constraints are used with the goal of finding memory instructions that can be used to return control to the attacker's code. Three real life applications apache2, ftpd, and imapd were analyzed with a success rate of about 85%.

Lakhotia et al. [38]

The approach to verifying virus and worm binaries uses a combination of techniques from the reverse engineering and model checking domains. A malicious program behavior is characterized using predicates. A predicate is a Boolean outcome of abstract action present in a worm or virus program. An action is a sequence of one or more function/system calls, in a program, connected through a flow relationship. The authors have introduced the term organ for the functional elements of a malicious code, which according to them are; survey, concealment, propagation, injection, self identification.

The authors use Model Checking technique to match a sample of malicious code with a supplied behavior. The prime advantage of using an established technique is efficiency and ease of use. Another advantage this approach has is that it takes a malicious code and a behavioral pattern to be matched. This allows the technique to be implemented as both signature matching and non signature matching scheme. If the behavioral pattern is generated using a virus signature the technique becomes a signature based scheme. Whereas if the patterns are generic and represent the general behavior of malicious code, then they this technique can be used as a non-signature based scheme.

3.1.2 Dynamic anomaly based detection

The dynamic anomaly based detection learns the normal run time behavior of programs in the training phase. The behavior of any program which deviates from this normal model is deemed as anomalous or malicious. The literature work done in dynamic anomaly based malware detection is briefly explained as follows:

Sekar et al. [37]

This paper aims to develop an anomaly based intrusion detection system. This system works by learning the sequence of system calls called by the programs under scrutiny, a compact finite state automaton (FSA) is developed for the sequence of the system calls for capturing normal behaviors. After that each program is dynamically analyzed on the basis of the FSA already developed and according to that analysis it is termed as malicious or normal. The novel approach adopted by the authors is the method to develop a compact FSA which can not only take less space but also capture much more information about program structures such as branches, loops etc. This method also takes less time and computation costs to develop an FSA and also to perform matching operations. This is achieved by not only capturing the system calls called by the programs but also the locations from where they are called.

The idea presented in this paper is simple and can be implemented with less computation, time, and space costs as compared to other similar methods e.g. other FSA based and N-gram Based ID methods. It also gives a high convergence and low false positive rates as compared to the above mentioned approaches.

As this method uses the value of program counter it may not work well with dynamically linked code. This convergence of this method depends on repeat- ing program behavior. This technique may not work pretty well when working on http servers because of varying behavior. This would result in more space requirements.

Yoshinori et al. [46]

This paper presents an idea to develop an anomaly intrusion detection technique based on process profiling. This technique is different from misuse detection techniques which record the malicious behavior and then classify analyzed software on the basis of comparisons done with the recorded malicious behavior. This process works by recording and analyzing the system calls made by setuid and daemon programs. Three types of profiles are developed which state the names of system calls with their ranks decided according to their frequencies. The profiles are then developed for every incoming software and are matched with the already developed profiles. The difference in distances of these profiles rate the nature of these threats. There are situations when the incoming patterns differ with the existing patterns only due to the time lagging problem i.e. they may perform the same functionality but with some time lag. The approach discussed above would generate a high difference rate between these profiles which actually is not true. This problem is solved by applying DP (Dynamic Processing) techniques which would contract or expand time to make the similar patterns match and solve this problem.

3.1.3 Hybrid anomaly based detection

The hybrid anomaly based detector uses the combination of static and dynamic approaches. Few interesting papers using this approach are briefly explained as follows:

Paul et al. [42]

The paper presents 'PolyUnpack', which uses a behavior-based approach that uses a combination of static and dynamic analysis to automate the process of extracting the hidden-code of unpack executing malware. First a static code view is generated for a piece of code by static analysis. Next, in the dynamic analysis the code is executed in a sterile and isolated environment. The execution is paused after each instruction and the context of the instruction is examined to find out whether it is the static code view or not. If it is not found in the static code view, representations of that unknown instruction sequence are written out and the malware's execution is halted. A formal definition of an unpack-executing is given, and an instruction execution bound n is introduced. Based on these two, an algorithm, to decide whether a given piece of code is unpack-executing or not, is provided. To avoid false positives due to the execution of code called from DLLs by programs, the pausing and comparing after each execution is disabled as long as the code from a dll is executing. There are a variety of challenges posed due to the complex nature of the 80x86 ISA and assembly. To overcome these and the overhead of determining whether an instruction being executed is a subsequence of an instruction in the static code view; the problem of detecting unpacked code via instruction subsequence is mapped into a series of statically assigned and dynamically created bounds checks that test whether the current value of PC points to a location statically or dynamically identified as code. The results of experiments although show that PolyUnpack is about 3 times faster on 40% of the sample data and about 12 times faster on the rest of the data as compared to manual unpacking.

Engin et al. [36]

This paper presents an idea to overcome the weaknesses of traditional signature based anti spyware software. Their idea is based on intrusion detection system approach which already has the knowledge of how non-self would behave. The technique uses both static and dynamic analysis of the specific objects i.e. Microsoft browser helper objects and toolbars. These objects are commonly used by spywares to launch most of their attacks usually comprising of capturing and leaking sensitive information. In the dynamic analysis step the interaction of under analysis software's component with the browser is monitored and all the browser's COM functions which are involved in the response to the events are recorded. Secondly the code regions which are responsible for handling events are determined. In the static analysis step these code regions are explored and analyzed. Their behavior is represented by control flow graphs. API functions called by these regions are recorded. In the end some limitations of this approach are discussed by presenting some ideas which would help the spyware writers to evade detection techniques. Some statistics of tests performed are shown which clearly show the static and dynamic analysis techniques both used together give much better results in not only discovering spywares but false positives as well.

3.2 Specification based detection

Specification based detection is the subset of anomaly based detection so it also has the training and testing phases. In the training phase of these systems usually some rule set or heuristics are developed which act as the exact specifications of the behavior a normal program should exhibit. However, to come up with the exact and accurate specifications and rule set is a daunting task, which in turn increases the complexity of these systems.

3.2.1 Static specification based detection

The static specification based detectors, analyze the structural properties and features of the program to make the training model, which is then used for classification in testing phase.

Bergeron et al. [33]

This paper addresses the problem of static slicing on binary executables for the purpose of the malicious code detection in COTS components. Static slicing is useful to extract security critical code fragments. As a first step of the developed methodology, the binary is disassembled to obtain an assembly code. This code is then transformed into its high level imperative representation, without losing its semantics. For instance the elimination of the stack, recovering of subroutine parameters, recovering of return results, etc are examples of code transformation. This substantially improves the analyzability of code. This also decreases the complexity of the detection problem, by only considering a potentially malicious code instead of the whole file. As a first step of transformation, idioms can be identified. These are set of assembly commands which have a logical meaning that cannot be fully understood from only one of them. By the application of data flow analysis we can improve the analyzability of the code. This can be done by the elimination of the stack, by treating the stack as a set of temporary variables. APIs and library subroutine prototypes can be used to compute the actual parameters and return values. Intra-procedural program slicing is applied to simplify register jumps instructions. The standard backward slicing algorithm is applied for program slicing to retain only those parts of code that are relevant to the suspicious parts of the program. The slicing algorithm uses system, procedure, data, and control dependence graphs (SDG, PDG, DDG, and CDG). Alias analysis allows the DDG to link the correct set of instructions to each other that further allows the complexity of detection to be reduced.

In their extended work [34], they presented a tool that detects malicious behavior of a program by analyzing it statically. It first generates an intermediate representation of the binary, analyzes the control and data flows, and then does the static verification using security automaton.

3.2.2 Dynamic specification based detection

In dynamic specification based system, the run time behavior of the programs is used for malware classification.

Linn et al. [39]

Executables can be obfuscated to complicate the process of reverse engineering to protect them from piracy and theft. The technique presented here in however, is focused on making executables resistant to disassembly in the first place. The difference in the start address of instructions obtained after static disassembly and that obtained at runtime, is a direct measure of the error in the disassembly. The goal therefore is to maximize this difference. One aspect that must be taken into account is the self repairing behavior (re-synchronization) of these differences because of the variable length nature of instructions (especially in the most widely used x86 ISA). Junk Insertion is done to 'confuse' the disassembly process as much as possible. Next, it is determined how many bytes of the said instruction should be inserted to confuse the disassembly process maximum. The above technique confuses Linear Sweep (a commonly used disassembly algorithm) by a confusion factor of 26%-30% (i.e. 26%-30% instructions are interpreted incorrectly) that can increase up to confusion factor of about 70% for Linear Sweep. To confuse Recursive Traversal (another commonly used disassembly algorithm), the basic assumption, and weakness of the algorithm is exploited, i.e. the control flow of the program can correctly be determined. The values of opaque expressions with artificial jump table constructs to mislead a disassembler. This technique is called 'jump table spoofing'.

Kirda et al. [41]

This paper has presented an idea of a personal web based firewall, to subvert the possibilities of exploiting vulnerabilities in web applications. As web applications are spreading at a fast rate, vulnerabilities are being discovered in them at a proportional rate as well. Injection of malicious Java Script code in the web applications is one of the vulnerabilities, which could be subverted by reducing the access to resources these applications can access. But now this approach has failed because even in this case malicious Java Script can be unknowingly downloaded (injected by an attacker in it) from a trusted site which can be executed at client side causing the leakage of private information to several unwanted locations. This problem can be subverted by overcoming it on the server side i.e. having strict security policies but then all the users are left on the mercy of the people on the server side that will they implement them or not. Authors in this paper have introduced a client side solution which analyzes the code being executed on the client side and tries to find any suspicious code or activities being performed. It notifies the user about an extra ordinary event to check if user wants to proceed with it or not. After the response from the user it acts accordingly and also records this rule to not to disturb the user about the same activity again for some time. This reduces the burden on the user as well and makes its working more efficient. To make this method more efficient other portions of the code e.g. the static links which are usually considered safe are analyzed as well because some attackers can use them as well. A rule is defined by the user according to them to decide when to notify the user about suspicious activity going on. Some tests were performed in this case which clearly showed the effectiveness of this tool. Some previous work in this field was analyzed as well which is mostly based on predefined policies, which has proven to be inefficient.

3.2.3 Hybrid specification based detection

These systems use both the static features and run time behavior of the programs to classify them as benign or malicious. Some of the papers are discussed below.

Rabek et al. [43]

This paper presents DOME, a host-based technique for detecting several general classes of malicious code in software executables. The key idea of DOME is to preprocess software executables to identify the locations of Win32 API calls in the software, and then to verify that every Win32 API call observed at runtime is made from a location identified during preprocessing. The Win32 API calls made by injected code and those made by obfuscated or dynamically generated code will not be detected during preprocessing, because these will not be present (for dynamically generated or injected code) and because no attempt is made to deobfuscate intentionally obfuscated code. DOME detects malicious code before it has a chance to interact with the OS or any of its resources. It also pinpoints the parts of MC where the calls were made from. The proof-of-concept study shows that DOME shows 100% detection for relevant MC classes. The performance overhead is however, 5% per Win32 API call.

3.3 Signature based detection

Signature based detection systems work by extracting signatures from the malicious files. These signature may be byte values, sequence of API calls or a combination of both. The signatures can be extracted manually as well as automatically. These signatures are then searched in programs to classify them as malicious or benign. As these approaches are dependent on the signature databases, the main drawback of signature based detection system is their failure to detect zero-day malware, i.e. they fail to detect the malware whose signature is missing in the signature database/repository.

3.3.1 Static signature based detection

The static signature based detection is done using the signature obtained from sequence of byte values or instructions that are malicious. The only advantage of this detection system is that the malicious program is classified before execution, only if its signature is present in the knowledge base. Normally manual analysis is required to generate the static signatures which increases the vulnerability period of a computer system.

Sandeep et al. [44]

The paper presents a generic virus scanner in C++. The scanner is completely generic and is independent of the platform it runs on and the platform of the files it scans. The virus scanner can be sure of the integrity of the byte sequences it reads if the files cannot execute on the host platform. Software monitors that use profiles to detect viruses have a great advantage as they can detect both known and unknown viruses. However, they require the normal usage profile to be fairly different from the malicious code activities, which may not always be the case. Detection by emulation is done by emulating program behavior for a specific input to decide whether it is malicious or not. This process is highly dependent on input and thus is less precise. An executable can be classified by checking if it confirms to a defined policy. This policy defines what a valid behavior of a program is. This however, requires the source of the program. Storing a checksum with a program and checking it each time the program runs is an excellent way of insuring that the program has not been infected, however, this method is ineffective when the program has already been infected. Checksums can be applied to each separate module of an executable to ensure integrity at runtime. This however, requires hardware support and trusted read and checksum operations. Time stamping can also be used as an alternate for checksuming, but it is vulnerable to the system clock being reset and other means of changing file modification times. Detection by signature matching is the most commonly used and cost effective method there is. Though it is vulnerable to obfuscation of malicious code and signature extraction takes significant effort, which may fail to keep up with the growing rate of virus proliferation. This paper presents a generic implementation of a virus scanner. It first reads from a virus signature database and arranges them in a sparse tree. The virus signature uses wild cards such as ? and * to compensate for code obfuscation.

Sung et al. [45]

This paper has taken into consideration the growing threat of malicious software and the destruction they can cause. Authors have discussed the types of different type of malicious code and have classified them on the basis of their properties and evolution. Authors explain in this paper the inefficiency of the current detection techniques whether static or dynamic (sand box techniques). They have explained that how these detection techniques fail due to obfuscation techniques used by malicious code writers and time constraints. In this paper the authors have introduced a tool known as SAVE (static analyser for vicious executables) to analyse portable executables (PE) and detect the malicious ones by analysing the calling sequence of API calls. This is a signature based tool i.e. a database of known patterns of sequence calls of viruses is build and then these patterns are matched with the incoming patterns of the exe files being analysed. This tool works on the binary code and explores the structure of portable executables to extract the sequence of API calls. It works quite well as compared to the common anti-viruses against polymorphic viruses as the patterns of API calls do not change entirely. The matching takes place by implying some matching formula like Euclidean and others to test if the two patterns match to the extent that they can be considered similar in functionality. Then that pattern is submitted to the database for future use. Many experiments were performed using obfuscated codes which show a much better performance by SAVE as compared to commercial anti virus tools which almost had 0

3.3.2 Dynamic signature based detection

Dynamic signature based techniques classify the program malicious on the basis of information which is extracted at run time only. These techniques keep check of any malicious behavior at run time.

Yang et al. [48]

In this paper, the authors apply several machine learning (ML) algorithms to automatically generate signatures for polymorphic worms. The authors use Naive Bayes, Support Vector Machine (SVMs), Decision Tree, and Rule Learner for worm fingerprinting. Their system uses a predetermined heuristic to identify suspicious hosts and mark all traffic from these hosts as suspicious, leaving other flows as unsuspicious. The suspicious and unsuspicious flows are then fed as training data to the different ML algorithms to generate accurate worm signatures. Experimental results show that an existing decision-tree learning algorithm and a rule learner both run faster than Polygraph (the best known automatic fingerprint generator to date) and produce fewer errors.

3.3.3 Hybrid signature based detection

These systems use run time analysis as well as sequences of code and byte values malware classification.

Lo et al. [24]

The malicious code filter (MCF) presents a technique named program slicing to 'analyze' malicious code. The authors present a technique in which through static analysis the code of the program is 'sliced' into different functional parts. These slices are mainly based on a certain subset of system calls. The paper extensively discusses the different kinds of malicious code, and the 'tell-tail signs' they would show. There is an in depth discussion of what type of tell tail signs a type of malicious code would show. The tell-tail sign concept is used to generate a data flow information graph. This graph is used for further analysis. However, this data flow information can be mislead severely if the code under analysis is not 'well behaved' i.e. there is some pointer or value over flow in the code. This type of over ow tends to mislead the data flow information, and thus the MCF.

The paper makes the assumption that the programs are 'well behaved' and there is no pointer over flow. The paper then discusses the tell tail signs showed by certain classes of malicious code, these include, 1) Trojan Login, 2) Multistage Launcher, 3) Development System Attack.Some UNIX vulnerabilities are also discussed; 1) Finger Daemon (fingerd), 2) Mail Notifier (comsat).

The detection/analysis mechanism based on the tell-tail signs concept is mechanized as follows: First the program under inspection is represented as control flow graph. Although some restrictions have been imposed on this graph to make it independent of the particular code used to undertake a task, however, this control flow graph does to some extent depend on the particular instructions/calls used and the variables used. As a second step a global flow analysis is done to see if one procedure alters the data in another procedure and the effects of the same. This allows for the development of on overall model of what a piece of code does. A malicious code writer may try to hide the malicious nature of the code by composing the code of seemingly benign procedures, which when interact turn into a malicious code. Next the code is sliced for different types of activities; such as; file access, time dependent computations, and race conditions. A similar technique is presented in another paper, S̈tatic Analysis of Executables to Detect Malicious Patterns". Many improvements have been introduced which render the system more reliable and real testing results are also given.

Mori et al. [40]

Authors developed a tool for analysis and detection of viruses and internet worms using code simulation, static code analysis, and OS execution emulation. It inspects the code and identifies commonly found mal behaviors like mass mailing, self duplication, and registry over write. Policies are used to define mal behaviors at API library function call level in a state transition like language. Policies include mass email, registry modification, file infection, file modification, process scan, self modification, anti-debugger/emulation, out of bounds execution, external execution, illegal address call, self duplication, and network connection policies. It targets win32 binary programs in Intel IA32 architecture.

The good thing in this approach is that it gives 95% detection on the sample set of 600 virus/worm using IAC, OBE, self duplication, and file modification policies. Over 80% detection using only file modification, self duplication, and external execution. Successful detection of unknown viruses. However, the tool is run in a virtual environment with CPU simulation, OS execution simulation etc.

CHAPTER 4

RELATED WORK

A significant amount of research has been conducted to analyze the characteristics and effects of malware. However, to maintain focus in this thesis, we include techniques proposed by Perdisci et al [1], Schultz et al [2] and Kolter et al [3] for detecting `Win32` malicious PE files. These techniques use *static analysis* and utilize n-grams and data mining algorithms to detect malicious PE files.

4.1 Perdisci et al [1] — McBoost

Recently, Perdisci et al have used a set of structural heuristics for detection of *packed executables* [49]. The authors argue to use their technique as a preprocessor before signature-based detectors. In this way, the executables need to be unpacked by a universal unpacker before doing the scanning. They use 9 heuristic features with different pattern recognition techniques for this purpose. The features include the number of standard sections, the number of non-standard sections, the number of executable sections, the number of readable/writable/executable sections, the number of entries in the import address table, the entropy of PE file header, the entropy of the code sections, the entropy of the data sections and the entropy of an entire PE file. A Multi-Layer Perceptron (MLP) classifier is used for eventual classification of the PE files. However, the scope of their work is limited to the detection of packed executables only.

In an extension to their previous work, Perdisci et al proposed *McBoost*, a statistical malware collection tool [1]. McBoost consists of three modules, **A**, **B** and **C**. The module **A** differentiates between packed and non-packed executables. It is further composed of three sub-modules: A1 is a heuristic-based classifier; A2 and A3 are n-gram based classifiers similar to those proposed by Kolter et al [3] (explained later in the chapter). A2 operates only on the code section of a PE file and A3 operates on an entire PE file. All executables classified as packed by the module **A** are sent to the module **B**. It contains a custom implementation of an unpacker for hidden code extraction. The implementation of module **B** uses QEMU emulator [50] similar to Renovo [51]. The module **C** performs eventual classification of malware. It also consists of two sub-modules: (1) C1 is an n-gram based classifier which operates on the code section of PE files classified as non-packed by the module **A**, and (2) C2 is also an n-gram based classifier which operates on the hidden code (extracted by the module **B**) of PE files classified packed by the module **A**.

The authors performed experiments on a dataset consisting of $5,586$ malicious PE files from Malfease dataset [10] and $2,258$ benign PE files. 37% malicius PE files are packed (detected using PEiD [52] and F-Prot antivirus [53]). Only 3% malicious PE files are non-packed and the rest of the files cannot be classified as packed or non-packed. The authors used Polyunpack [54] and custom developed unpacker for unpacking. On the other hand, only 2% benign executables are packed and rest 98% are non-packed in the original dataset. The skewness in the distribution of packed/non-packed malware and benign files is obvious.

McBoost is primarily a malware collection tool and its utility as an online realtime malware detection tool is limited due to high processing overheads and relatively lower detection rates. In [1], the authors report that McBoost requires approximately 1.06 seconds for detection of non-packed executables, 4.7 and 5.6 minutes for detection of packed PE files and benign PE files respectively. Further, McBoost is unable to analyze $1,030$ (approximately 13%) "heavily packed" executables which effectively require manual analysis for eventual classification.

4.2 Schultz et al [2] — Strings

In [2], Schultz et al use several data mining techniques to distinguish between the benign and malicious executables in Windows or MS-DOS format. They have done experiments on a dataset that consists of $1,001$ benign and $3,265$ malicious executables. These executables have 206 benign and 38 malicious samples in the PE file format. They collected most of the benign executables from Windows 98 systems. They use three different approaches to statically extract features from executables.

The first feature extracts DLL information inside PE executables. Further, the DLL information is extracted using three types of feature vectors: (1) the list of DLLs (30 boolean values), (2) the list of DLL function calls ($2,229$ boolean values), and (3) the number of different function calls within each DLL (30 integer values). RIPPER (an inductive rule-learning algorithm) is used on top of every feature vector for classification. These schemes based on DLL information provides an overall detection accuracy of 83.62%, 88.36% and 89.07% respectively.

The second feature extraction approach extracts the strings from the executables using GNU *strings* program. Naïve Bayes classifier is used on top of extracted strings for detection. This scheme provides an overall detection accuracy of 97.11%.

The third feature extraction approach uses byte sequences (n-grams) using hexdump. The authors do not explicitly specify the value of n used in their study. However, from an example provided in the paper, we deduce it to be 2 (bi-grams). The Multi-Naïve Bayes algorithm is used for classification. This algorithm uses voting by a collection of individual Naïve Bayes instances. This scheme provides an overall detection accuracy of 96.88%.

The authors also compare their proposed schemes with a custom developed signature-based detector that uses traditional byte sequence based signatures. Such schemes are meant for low false positive rates. The signature-based detector provides an overall accuracy of 49.28% only.

The results of their experiments reveal that Naïve Bayes algorithm with strings is the most effective approach for detecting the unseen malicious executables with reasonable processing overheads. The authors acknowledge the fact that the string features are not robust enough and can be easily defeated. Multi-Naïve Bayes with byte sequences also provides a very high detection accuracy, however, it has large processing and memory requirements. These overheads make the approach infeasible for realtime deployment.

4.3 Kolter et al [3] — KM

In [3], Kolter et al use n-gram and data mining approaches to detect malicious executables in the wild. They use n-gram analysis to extract features from $1,971$ benign and $1,651$ malicious PE files. The PE files have been collected from machines running Windows 2000 and XP operating systems. The malicious PE files are taken from an older version of the VX Heavens Virus Collection [9].

The authors evaluate their approach for two classification problems: (1) classification between the benign and malicious executables, and (2) categorization of executables as a function of their payload. The authors have categorized three types of malware, mailer, backdoor and virus due to the limited number of samples.

Top n-grams with the highest information gain as binary features (`T` if present and `F` if absent) in every classification problem. The authors have done pilot studies to determine the size of n-grams, the size of words and the number of top n-grams to be selected as features. A smaller dataset consisting of 561 benign and 476 malicious executables is considered in this study. They used 4-grams, one byte word and top 500 n-grams are selected as features.

A number of inductive learning methods, namely instance-based learner, Naïve

Bayes, support vector machines, decision trees and boosted versions of instance-based learner, Naïve Bayes, support vector machines and decision trees are used for classification. The same features are provided as an input to all classifiers. They report their results as the area under an ROC curve (AUC) which is a more complete measure compared with the detection accuracy [55], [11]. AUCs show that the boosted decision trees outperform the rest of the classifiers for both classification problems.

CHAPTER 5
DATASETS

In this chapter, we present an overview of the datasets used in our study. We have collected $1,447$ benign PE files from the local network of our lab. The collection contains executables such as Packet CAPture (PCAP) file parsers compiled by MS Visual Studio 6.0, compressed installation executables and MS Windows XP/Vista applications' executables. The diversity of the benign files is also evident from their sizes, which range from a minimum of 4 KB to a maximum of $104,588$ KB (see Table 3).

Moreover, we have used two malware collections in our study. First is the VX Heavens Virus Collection, which is *labeled* and is publicly available for free download [9]. We only consider PE executables to maintain focus. Our filtered dataset contains $10,339$ malicious PE files. The second dataset is the Malfease malware dataset [10], which consists of $5,586$ *unlabeled* malicious PE files.

In order to conduct a comprehensive study, we further categorize the malicious PE files as a function of their payload[1]. The malicious executables are subdivided into eight major categories such as *virus*, *trojan*, *worm*, etc. Moreover, we have combined some categories that have similar functionality. For example, we have combined *constructor* and *virtool* to create a single *constructor + virtool* category. This unification increases the number of malware samples per category. It will be helpful later when we categorize the malicious executables as a function of their payload. We now provide a brief introduction of each malware category, used in our study, to make it self contained [56].

[1]Since the Malfease malware collection is unlabeled, therefore, it is not possible to divide it into different malware categories.

Table 3. Statistics of the data used in this study.

Dataset	**VX Heavens**									**Malfease**
Title of Statistic	**Benign**	**Backdoor + Sniffer**	**Constructor + Virtool**	**DoS + Nuker**	**Flooder**	**Exploit + Hacktool**	**Worm**	**Trojan**	**Virus**	-
Quantity	1,447	3,455	367	267	358	243	1,483	3,114	1,052	5,586
Average Size (KB)	1,263	270	234	176	298	156	72	136	50	285
Minimum Size (KB)	4	1	4	3	6	4	2	1	2	1
Maximum Size (KB)	104,588	9,277	5,832	1,301	14,692	1,924	2,733	4,014	1,332	5,746
Packed using UPX	17	786	79	15	32	43	353	622	48	470
Packed using ASPack	2	432	21	16	25	15	66	371	10	187
Packed using Other	372	325	47	31	58	38	471	170	71	1,909
Compiled using Borland C/C++	15	56	8	15	10	6	13	63	18	11
Compiled using Borland Delphi	13	589	13	65	64	8	76	379	71	342
Compiled using Visual Basic	4	719	106	39	126	38	210	674	119	809
Compiled using Visual C++	526	333	19	51	29	59	89	619	96	351
Compiled using Visual C#	56	0	0	0	1	0	5	1	6	1
Compiled using Other	9	49	9	2	3	2	4	15	7	5
Total Non-packed (%)	43.1	50.5	42.2	64.4	65.1	46.5	26.8	56.2	30.1	27.2
Total Packed (%)	27.0	44.7	40.1	23.2	32.1	39.5	60.0	37.4	12.3	46.6
Not Found (%)	29.9	4.8	17.7	12.4	2.8	14.0	13.2	6.4	57.6	26.2

Backdoor + Sniffer.

A backdoor is a program which allows bypassing of standard authentication methods of an operating system. As a result, remote access to computer systems is possible without explicit consent of the users. Information logging and sniffing activities are possible using the gained remote access.

Constructor + Virtool.

This category of malware mostly includes toolkits for automatically creating new malware by varying a given set of input parameters. Virtool and constructor categories are combined because of their similar functionality.

DoS + Nuker.

Both DoS and nuker based malware allow an attacker to launch malicious activities at a victim's computer system that can possibly result in a denial of service attack. These activities can result in slow down, restart, crash or shutdown of a computer system.

Email- + IM- + SMS Flooder.

The malware in this category initiate unwanted information floods such as email, instant messaging and SMS floods.

Exploit + Hacktool.

The malware in this category exploit vulnerabilities in a system's implementation which most commonly results in buffer overflows.

Email- + IM- + IRC- + Net Worm.

The malware in this category spreads through instant messaging networks, IRC networks and port scanning.

Trojan.

A trojan is a broad term that refers to stand alone programs which appear to perform a legitimate function but covertly do possibly harmful activities such as providing remote access, data destruction and corruption.

Virus.

A virus is a program that can replicate itself and attach itself with other benign programs. It is probably the most well-known type of malware and has different types.

Table 3 provides the detailed statistics of the malware used in our study. A careful reader can rightly conclude that the average size of the malicious executables is smaller than that of the benign executables. Further, some executables used in our study are encrypted and/or compressed (packed). The detailed statistics about packing are also tabulated in Table 3. We use PEiD [52] and Protection ID for detecting packed executables [57][2].

Our analysis shows that VX Heavens Virus collection contains 40.1% packed and 47.2% non-packed PE files. However, approximately 12.7% malicious PE files cannot be classified as either packed or non-packed by PEiD and Protection ID. The Malfease collection contains 46.6% packed and 27.2% non-packed malicious PE files. Similarly, 26.2% malicious PE files cannot be classified as packed or non-packed. We can, therefore, say that packed/non-packed malware distribution in the VX Heavens virus collection is relatively more balanced than the Malfease dataset. In our collection of benign files, 43.1% are packed and 27.0% are non-packed PE files respectively. Similarly, 29.9% benign files are not detected by PEiD and Protection ID. An interesting observation is that the benign PE files are mostly packed using nonstandard and custom developed packers. We speculate

[2]We acknowledge the fact that PEiD and Protection ID are signature based packer detectors and can have false negatives.

that a significant portion of the packed executables are not classified as packed because the signatures of their respective packers are not present in the database of PEiD or Protection ID. Note that we do not manually unpack any PE file prior to the processing of our PE-Miner.

CHAPTER 6
EXPERIMENTAL RESULTS AND EVALUATION

Recall that we set two objectives for doing a comparative evaluation of different techniques: (1) the primary objective is to distinguish between benign and malicious PE files, and (2) the secondary objective is to categorize the malicious executables as a function of their payload. We have compared our PE-Miner framework with recently proposed promising techniques by Perdisci et al [1], Schultz et al [2] and Kolter et al [3]. We briefly summarize their working principles in the following paragraphs again.

In [1], the authors proposed McBoost that uses two classifiers, C1 and C2, for classification of non-packed and packed PE files respectively. A custom developed unpacker is used to unpack the packed PE files and these unpacked executables are given as an input to the C2 classifier. Unfortunately, we could not obtain its source code or binary due to licensing related problems. Further its implementation is not within the scope of our current work. Consequently, we only evaluate the C1 module of McBoost which works only for non-packed PE files. We, therefore, acknowledge that our McBoost results should be considered preliminary. However, they do provide useful insight into the detection behavior of McBoost.

In [2], Schultz et al have proposed three independent techniques for detecting malicious PE files. The first technqiue, uses the information about DLLs, function calls and their invocation counts. However, the authors did not provide enough information about the used DLLs and function names; therefore, it is not possible for us to implement it. But we have implemented the second approach (titled *strings*) which uses strings as binary features i.e. present or absent. The third technique uses two byte words as binary features. This technique is later improved in a sem-

Table 4. AUCs for detecting the malicious executables. The bold entries in each column represent the best results.

Dataset	VX Heavens									Malfease
Malware	Backdoor + Sniffer	Constructor + Virtool	DoS + Nuker	Flooder	Exploit + Hacktool	Worm	Trojan	Virus	Average	-
PE-Miner — RFR										
IBK	0.992	0.996	0.995	0.994	0.998	0.979	0.984	0.994	**0.992**	0.986
J48	0.993	**0.998**	0.987	0.993	0.999	0.979	**0.992**	0.993	**0.992**	0.979
NB	0.971	0.978	0.966	0.973	0.987	0.972	0.974	0.986	0.976	0.976
RIPPER	**0.996**	0.996	0.977	0.981	0.999	**0.988**	0.988	0.996	0.990	0.985
SMO	0.991	0.990	0.991	0.993	0.997	0.975	0.978	0.992	0.988	0.963
PE-Miner — PCA										
IBK	0.989	0.996	0.994	0.995	0.998	0.976	0.984	0.993	0.991	0.984
J48	0.980	0.966	0.929	0.960	0.987	0.936	0.951	0.985	0.962	0.945
NB	0.961	0.990	0.993	0.996	0.996	0.964	0.956	0.990	0.981	0.898
RIPPER	0.982	0.978	**0.996**	0.974	0.977	0.949	0.968	0.987	0.976	0.952
SMO	0.990	0.992	0.989	0.995	0.995	0.958	0.965	0.992	0.985	0.954
PE-Miner — HWT										
IBK	0.991	0.996	**0.996**	**0.998**	**1.000**	0.978	0.985	0.995	**0.992**	0.986
J48	0.995	0.997	0.993	0.988	0.997	0.978	0.991	0.999	**0.992**	0.977
NB	0.989	0.982	0.983	0.987	0.990	0.978	0.972	0.990	0.984	0.960
RIPPER	0.994	0.997	0.982	0.990	0.997	0.983	0.990	**1.000**	**0.992**	**0.987**
SMO	0.990	0.995	0.991	0.996	**1.000**	0.972	0.973	0.994	0.989	0.964
McBoost — C1 only										
IBK	0.941	0.935	0.875	0.960	0.832	0.938	0.930	0.914	0.916	0.949
J48	0.866	0.895	0.809	0.893	0.731	0.906	0.902	0.882	0.860	0.860
NB	0.831	0.924	0.723	0.889	0.795	0.873	0.886	0.844	0.846	0.817
RIPPER	0.833	0.888	0.744	0.918	0.660	0.866	0.838	0.844	0.824	0.860
SMO	0.802	0.887	0.759	0.910	0.678	0.854	0.805	0.827	0.815	0.835
Strings										
IBK	0.949	0.860	0.902	0.980	0.925	0.928	0.863	0.952	0.920	0.944
J48	0.913	0.834	0.862	0.695	0.871	0.908	0.836	0.938	0.857	0.929
NB	0.920	0.830	0.882	0.726	0.886	0.901	0.828	0.905	0.860	0.930
RIPPER	0.843	0.797	0.714	0.578	0.712	0.892	0.743	0.929	0.776	0.927
SMO	0.855	0.817	0.705	0.775	0.583	0.871	0.756	0.883	0.781	0.933
KM										
IBK	0.984	0.934	0.983	0.971	0.983	0.987	0.979	0.986	0.976	0.980
J48	0.953	0.940	0.916	0.907	0.916	0.957	0.951	0.953	0.937	0.952
NB	0.943	0.959	0.961	0.952	0.961	0.968	0.954	0.954	0.957	0.961
RIPPER	0.951	0.944	0.924	0.921	0.924	0.964	0.948	0.948	0.941	0.971
SMO	0.949	0.946	0.952	0.927	0.952	0.961	0.940	0.938	0.946	0.960

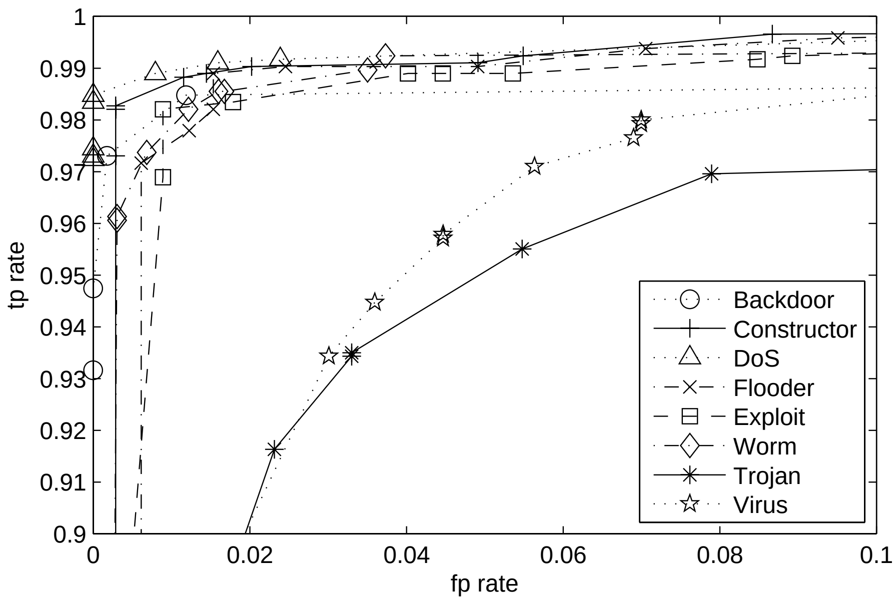

Figure 4. The magnified ROC plots for detecting the malicious executables using PE-Miner utilizing J48 preprocessed with RFR filter. The results are shown for VX Heavens dataset.

inal work by Kolter et al [3] which uses 4-grams as binary features. Therefore, we include the technique of Kolter et al (titled *KM*) in our comparative evaluation.

We have used the standard 10 fold cross-validation process in our experiments: the dataset is randomly divided into 10 smaller subsets, where 9 subsets are used for training and 1 subset is used for testing. The process is repeated 10 times for every combination. This methodology helps in evaluating the robustness of a given approach to detect malicious PE files that contain malware without any a priori information. The ROC curves are generated by varying the threshold on output class probability [55], [11]. The AUC is used as a yardstick to determine the detection accuracy of each approach. We have done the experiments on an Intel Pentium Core 2 Duo 2.19 GHz processor with 2 GB RAM. The Microsoft Windows XP SP2 is installed on this machine. We now separately report the detection accuracies of different approaches for our primary and secondary objectives.

6.1 Primary Objective: Malicious PE File Detection

In our first experimental study, we attempt to distinguish between benign and malicious PE files. To get better insights, we have done independent experiments with benign and each of the eight types of the malicious executables. The five data mining algorithms, namely IBk, J48, NB, RIPPER and SMO, are deployed on top of each approach (namely PE-Miner with RFR, PE-Miner with PCA, PE-Miner with HWT, McBoost (C1 only) by Perdisci et al [1], strings approach by Schultz et al [2] and KM by Kolter et al [3]). This results in a total of 270 experimental runs each with 10-fold cross validation. We tabulate our results for this study in Table 4 and now answer different questions that we raised in Chapter 2 in a chronological fashion.

6.1.1 Which features' set is the best?

Table 4 tabulates the AUCs for PE-Miner using three different preprocessing filters (RFR, PCA and HWT), McBoost, strings and KM [3]. A macro level scan through the table clearly shows the supremacy of PE-Miner based approaches with AUCs more than 0.99 for most of the malware types and even approaching 1.00 for some malware types. For PE-Miner, RFR and HWT preprocessing lead to the best average results with more than 0.99 AUC.

The strings approach gives the worst detection accuracy. The KM approach is better than the strings approach but inferior to our PE-Miner. This is expected because the string features are not stable as compiling a given piece of code by using different compilers leads to different sets of strings. Our analysis shows that KM approach is more resilient to variation in the string sets because it uses a combination of string and non-string features. The results obtained for KM approach are also consistent with the results reported in [3]. Its average AUC is about 0.95. The C1 module of McBoost also provides relatively inferior detection

Table 5. The processing overheads (in milliseconds/file) of different feature selection, extraction and preprocessing schemes.

	PE-Miner			McBoost	Strings	KM
	(RFR)	(PCA)	(HWT)			
Selec.	-	-	-	2839	5289	31499
Ext.	228	228	228	198		220
Preproc.	7	9	12	-	-	-
Total	**235**	237	240	3037	5419	31719

accuracies which are as low as 0.66 for exploit+hacktool category. It is important to note that the C1 module of McBoost is functionally similar to the techniques proposed by Schultz et al and Kolter et al. The only significant difference is that C1 operates only on the code sections of the non-packed PE files whereas the other techniques operate on complete files.

It is important to emphasize that both strings and KM approaches incur large overheads in the feature selection process (see Table 5[1]). Kolter et al have confirmed that their implementation of information gain calculation for feature selection took almost a day for every single run. To make our implementation of n-grams more efficient, we use `hash_map` STL containers in the Visual C++ [58]. Our experiments show that the feature selection process in KM still takes more than 31000 milliseconds per file with our optimized implementation. The optimized strings approach takes, on the average, more than 5000 milliseconds per file for feature selection. The optimized McBoost (C1 only) approach takes an average of more than 2000 milliseconds per file for feature selection[2]. Therefore, these approaches do not satisfy our definition of *realtime deployable*. This is because the processing overheads in calculation of information gain increase exponentially with the number of unique n-grams (or strings). On the other hand, PE-Miner does not suffer from such serious bottlenecks. The application of RFR, PCA or HWT

[1]The results in Table 5 are averaged over 100 runs.

[2]Note that the complete McBoost system also uses universal unpacker for extraction of hidden code. This process is very time consuming. Therefore, the processing overheads for McBoost represent the best-case scenario (all executables are non-packed)

Table 6. The processing overheads (in milliseconds/file) of different features and classification algorithms.

	IBK	J48	NB	RIPPER	SMO
Training					
PE-Miner (RFR)	-	8	1	269	199
PE-Miner (PCA)	-	7	1	264	179
PE-Miner (HWT)	-	7	1	252	147
McBoost	-	21	4	1305	1122
Strings	-	9	2	799	838
KM	-	24	4	1510	1018
Testing					
PE-Miner (RFR)	32	1	2	2	2
PE-Miner (PCA)	35	1	1	1	2
PE-Miner (HWT)	32	1	2	1	2
McBoost	218	10	7	5	22
Strings	163	3	3	2	3
KM	254	18	7	5	20

filters takes only about a few milliseconds.

6.1.2 Which classification algorithm is the best?

We can conclude from Table 4 that the J48 outperforms the rest of the data mining classifiers in terms of the detection accuracy in most of the cases. Moreover, Table 6 shows that J48 has one of the smallest processing overheads both in training and testing. RIPPER and IBk closely follow the detection accuracy of J48. However, they are infeasible for realtime deployment because of the high processing overheads in the training and the testing phases respectively. The processing overheads of training RIPPER are the highest among all classifiers. In comparison, IBk does not require a training phase but its processing overheads in the testing phase are the highest. Further, Naïve Bayes gives the worst detection accuracy because it assumes independence among input features. Intuitively speaking, this assumption does not hold for all features' sets used in our study. Note that Naïve Bayes has very small learning and testing overheads (see Table 6[3]).

[3]The results in Table 6 are averaged over 100 runs.

6.1.3 Which malware category is the most challenging to detect?

An overview of Table 4 suggests that the most challenging malware categories are worms and trojans. The average AUC values of the compared techniques for worms and trojans are approximately 0.95. The poor detection accuracy is attributed to the fact that the trojans are inherently designed to appear similar to the benign executables. Therefore, it is a difficult challenge to distinguish between trojans and benign PE files. Our PE-Miner still achieves on the average 0.98 AUC for worms and trojans which is quite reasonable. Figure 4 shows that for other malware categories, PE-Miner (with RFR preprocessor) has AUCs more than 0.99.

6.2 Secondary Objective: Malicious Executable Detection as a Function of Payload

In our second comparative study, we attempt to categorize the malicious executables as a function of their payload. This means that we want to know the category of a malware in a given malicious PE file. Recall from Chapter 5 that we have eight different categories of malware. This is in fact a multi-class classification problem and is therefore significantly more challenging than the primary objective of just categorizing the benign and malicious PE files. The secondary objective is not a necessity from the point-of-view of use in a COTS AV product. However, it is definitely a *nice-to-have* feature for system administrators and malware forensic experts. We follow the same 'one-vs-all' classification approach as is used by the authors in [3]. We have done experiments with the VX Heavens dataset in which each malware category is labeled as a separate class. Table 7 tabulates the results from our second study. We again answer the same questions raised in Chapter 2 in a chronological fashion for our second study.

Table 7. AUCs for detecting the malicious executables as a function of their payload (only on VX Heavens dataset). The bold entries in each column represent the best results.

Classifier	**Backdoor + Sniffer**	**Constructor + Virtool**	**DoS + Nuker**	**Flooder**	**Exploit + Hacktool**	**Worm**	**Trojan**	**Virus**	**Average**
PE-Miner — RFR									
IBK	0.936	0.853	0.803	0.729	0.849	0.931	0.908	0.890	0.862
J48	**0.972**	**0.946**	0.920	0.840	0.897	**0.969**	**0.957**	0.936	**0.929**
NB	0.747	0.774	0.756	0.653	0.831	0.795	0.837	0.738	0.766
RIPPER	0.937	0.918	**0.926**	0.806	0.884	0.959	0.943	**0.938**	0.914
SMO	0.910	0.928	0.844	0.749	0.936	0.891	0.912	0.892	0.883
PE-Miner — PCA									
IBK	0.933	0.862	0.794	0.735	0.854	0.929	0.910	0.885	0.863
J48	0.862	0.752	0.707	0.665	0.723	0.861	0.833	0.805	0.776
NB	0.816	0.829	0.814	0.784	0.842	0.850	0.802	0.813	0.819
RIPPER	0.847	0.802	0.763	0.716	0.797	0.875	0.824	0.819	0.805
SMO	0.899	0.932	0.822	0.735	0.934	0.873	0.897	0.879	0.871
PE-Miner — HWT									
IBK	0.934	0.856	0.794	**0.886**	0.854	0.932	0.912	0.886	0.882
J48	0.866	0.763	0.703	0.812	0.767	0.865	0.826	0.812	0.802
NB	0.824	0.833	0.816	0.790	0.862	0.858	0.811	0.814	0.826
RIPPER	0.942	0.925	0.908	0.778	0.876	0.958	0.947	**0.938**	0.909
SMO	0.908	0.924	0.849	0.748	**0.949**	0.889	0.913	0.888	0.884
McBoost — C1 only									
IBK	0.795	0.749	0.747	0.650	0.636	0.777	0.696	0.771	0.728
J48	0.739	0.614	0.669	0.597	0.573	0.716	0.644	0.703	0.657
NB	0.714	0.660	0.673	0.609	0.577	0.656	0.622	0.693	0.651
RIPPER	0.644	0.581	0.622	0.579	0.537	0.625	0.600	0.654	0.605
SMO	0.735	0.709	0.726	0.672	0.617	0.734	0.647	0.730	0.696
Strings									
IBK	0.741	0.666	0.780	0.673	0.665	0.690	0.659	0.740	0.702
J48	0.730	0.670	0.718	0.600	0.657	0.674	0.656	0.739	0.681
NB	0.654	0.659	0.770	0.633	0.657	0.592	0.572	0.722	0.657
RIPPER	0.584	0.542	0.621	0.568	0.579	0.594	0.563	0.574	0.578
SMO	0.717	0.654	0.773	0.677	0.692	0.656	0.636	0.721	0.691
KM									
IBK	0.853	0.769	0.800	0.741	0.686	0.851	0.793	0.907	0.800
J48	0.771	0.722	0.727	0.672	0.625	0.731	0.689	0.849	0.723
NB	0.718	0.660	0.644	0.684	0.699	0.715	0.627	0.814	0.695
RIPPER	0.715	0.700	0.705	0.663	0.670	0.678	0.704	0.812	0.706
SMO	-	-	-	-	-	-	-	-	-

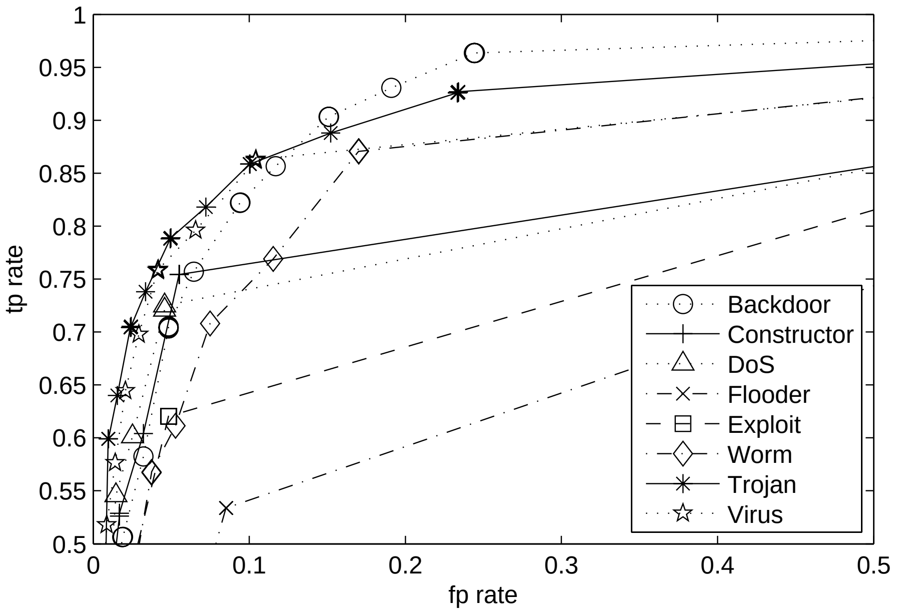

Figure 5. The magnified ROC plots for detecting the malicious executables as a function of their payload using PE-Miner utilizing J48 preprocessed with RFR filter. The results are shown only for VX Heavens dataset.

6.2.1 Which features' set is the best?

The results of the payload based malicious executable classification follow similar patterns as are observed in the first experimental study. The best detection accuracy is again achieved with the PE format specific structural feature extraction algorithm. PE-Miner with RFR and HWT preprocessing on the average has 0.93 AUCs. In comparison, the detection accuracies of McBoost, strings and KM approaches are significantly deteriorated and the average AUCs are in the range of 0.60 – 0.70. This further strengthens our thesis that the format specific structural feature extracting scheme can achieve both primary and secondary objectives with the highest detection accuracies and the lowest processing overheads.

6.2.2 Which classification algorithm is the best?

The relative performance of the data mining classification algorithms is similar to the previous study. Here again, PE-Miner using J48 performs significantly better

Table 8. The processing overheads (in milliseconds/file) of different features and classification algorithms with standard deviations.

	IBK	J48	NB	RIPPER	SMO
Training					
PE-Miner (RFR)	-	5 (10)	4 (8)	14 (8)	137 (36)
PE-Miner (PCA)	-	3 (6)	2 (5)	8 (8)	140 (43)
PE-Miner (HWT)	-	11 (8)	11 (9)	37 (17)	203 (69)
McBoost	-	67 (18)	16 (10)	155 (33)	243 (47)
Strings	-	17 (10)	3 (6)	32 (14)	144 (4)
KM	-	42 (15)	15 (4)	110 (26)	208 (33)
Testing					
PE-Miner (RFR)	3 (6)	1 (3)	3 (6)	0 (0)	0 (2)
PE-Miner (PCA)	3 (7)	0 (0)	1 (3)	0 (2)	0 (0)
PE-Miner (HWT)	7 (8)	0 (2)	6 (11)	0 (0)	0 (2)
McBoost	15 (6)	0 (0)	9 (8)	0 (0)	0 (2)
Strings	7 (8)	0 (0)	1 (4)	0 (2)	0 (2)
KM	11 (7)	0 (0)	9 (8)	0 (2)	0 (2)

than the other classifiers. Another important observation in these experiments is that the processing overheads of all approaches have significantly increased for the payload based malicious executable classification. In fact, the KM approach with SMO classifier does not finish despite running for several days and hence no values are reported for it in Table 7. Our observation is that the J48 gave the best detection accuracy with relatively smaller processing overheads in the training and testing phases.

6.2.3 Which malware category is the most challenging to categorize?

It is interesting to note in Table 7 that the worm and trojan categories have the best AUCs. This trend is opposite to the pattern observed in the previous study where these categories of malware were the most difficult to detect. Here the flooder and nuker categories have the worst AUCs. Our analysis reveals that several flooder samples are wrongly classified as nuker and vice-versa leading to worst accuracies. This is because both of them have similar functional behavior (refer to Chapter 5). This trend can also be observed in Figure 5.

Table 8 shows the processing overheads of the the used algorithms with standard deviations. This timing analysis has been done on a sample dataset of 100 files with 10-fold cross validation.

6.3 Miscellaneous Discussions

The training models of the J48 and RIPPER also provide interesting insights into the characteristics of different categories of malware. The partial subtrees of J48 for categorizing benign and malicious PE files are shown in Table 9. The message tables mostly do not exist in the backdoor+sniffer and constructor+virtool categories. The TimeDateStamp is usually obfuscated in the malicious executables. The number of resources are generally smaller in malicious PE files, whereas the benign files tend to have larger number of resources such as menus, icons and user defined resources. Moreover the reference of `WSOCK32.DLL` increases the probability that a given PE file might be doing a trojan-like activity. We note that the sizes of the certificate table, resource section (if exists), uninitialized data and code section are smaller for viruses compared with those of benign executables. Similar insights are also provided by the rules developed in the training phase of the RIPPER.

PE-Miner framework exploits our analysis of learning models of classifiers that helped in improving its accuracy in realtime. We tabulate the AUC and the scan time of the best techniques in Table 10. Moreover, we also show the scan time of two well-known COTS AV products for doing the *realtime deployable* analysis of different non-signature based techniques. It is clear that PE-Miner (RFR) with J48 classifier is the only non-signature based technique that satisfies the criteria of being *realtime deployable* introduced in Chapter 2. One might argue that PE-Miner framework provides only 0.014 AUC improvement over the KM approach. But then KM has the worst scan time of 31.97 seconds per file (see Table 10). It is very important to interpret the results in Table 10 from a security expert's perspective. For example, if a malware detector scans ten thousand files with an AUC of 0.97, it will not detect approximately 300 malicious files. In comparison, a detector with an AUC of 0.99 will miss only 100 files, which is a 66.6% improvement in the number of missed files [59]. The number of undetected files in the wild represent a serious threat because "the longer a threat remains undiscovered in the wild, the more opportunity it has to compromise computers before measures can be taken to protect against it. Furthermore, its ability to steal information increases the longer it remains undetected on a compromised computer" [6]. We therefore argue that from a security perspective, even a small improvement in the detection accuracy is significant in the limiting case when it approaches to 1.00.

We conclude this chapter with another important discussion that the authors do in [1] about the ability of different schemes to detect packed/non-packed malware. They show that the detection accuracy of KM approach degrades on a "difficult dataset" consisting of packed benign and non-packed malicious PE files. Our experimental study on VX Heavens and Malfease dataset shows that the results of KM approach remain consistent if *enough* training samples are used. The results of our PE-Miner framework also show consistent detection accuracies on the "difficult dataset" derived from both malware collections. This shows its robustness to the limited number of training samples and the packing problem.

6.4 Cross Dataset Analysis

In this section, we would like to test our approach rigorously using cross dataset training-testing. Earlier we have been using 10-fold cross validation within a dataset. The 10-fold cross validation basically means that we are dividing the dataset in 10 subsets, then training the classifier using 9 subsets and testing on the 10th subset. This process is repeated ten times so that every subset is tested. Although cross validation provides a very authentic method of testing whole of the dataset, it gives the inherent bias of learning the pattern/flow of the dataset to the classifier. It would be interesting to see how accurate our scheme turns out to

Table 9. Portions of developed decision trees

```
NumMessageTable <= 0
|    SizeLoadConfigTable <= 0
|    |    TimeDateStamp <= 1000000000
|    |     |      NumCursor <= 1
|    |     |       |    NumAccelerators <= 0
|    |     |       |     |      NumBitmap <= 0: malicious
|    |     |       |     |      NumBitmap > 0: benign
|    |     |       |    NumAccelerators > 0:malicious
|    |     |      NumCursor > 1:malicious
```

(a) between benign and backdoor+sniffer

```
NumMessageTable <= 0
|    NumBitmap <= 0
|    |    NumAccelerators <= 1: malicious
|    |    NumAccelerators > 1
|    |     |     NumUserDefined <= 0: malicious
|    |     |     NumUserDefined > 0: benign
|    NumBitmap > 0
|    |    NumUserDefined <= 0: malicious
|    |    NumUserDefined > 0: benign
NumMessageTable > 0: benign
```

(b) between benign and constructor+virtool

```
NumDialog > 8
|    NumIcon <= 2
|    |    NumMenu <= 1: malicious
|    |    NumMenu > 1: benign
|    NumIcon > 2
|    |    WSOCK32.DLL <= 0: benign
|    |    |    NumIcon <= 8: benign
|    |    |    NumIcon > 8: malicious
```

(c) between benign and trojan

```
SizeCertificateTable <= 2500
|    .rsrcSizeofRawData <= 17000
|    |   SizeofUnintializedData <= 4096: malicious
|    |   SizeofUnintializedData > 4096
|    |   |      SizeofCode <= 30208: malicious
|    |   |      SizeofCode > 30208: benign
|    .rsrcSizeofRawData > 17000
SizeCertificateTable > 2500: benign
```

(d) between benign and virus

Table 10. Realtime deployable analysis of the best techniques

Technique	Classifier	AUC	Scan Time (sec/file)	Is Realtime Deployable?
PE-Miner (RFR)	J48	0.991	0.244	**Yes**
McBoost	IBk	0.926	3.255	No
Strings	IBk	0.927	5.582	No
KM	IBk	0.977	31.973	No
AVG Free 8.0 [60]	-	-	0.159	-
Panda 7.01 [61]	-	-	0.131	-

be when it is trained with one dataset and tested with the others.

We are using three datasets for the cross dataset analysis. Two of them, Malfease and VX Heavens are detailed in Chapter 5. The testing of these datasets is done using another dataset which is obtained from `OffensiveComputing.net` [62], we will refer this dataset as Óffensive' from now onwards. This dataset contains about half a million malwares. For the sake of initial evalution, we are using a subset of the whole dataset and have selected approximately thirty thousand malware samples randomly.

The Offensive dataset is the most difficult and challenging in its nature as it is the biggest malware repository which is open for information security research. Malfease dataset comes second in terms of difficulty level with respect to malware analysis, where as VX Heavens is considered to be the least difficult dataset as compared to others. It would be interesting to see how the difficulty level of these datasets affect our cross dataset evaluation. Intuitively, we expect the performance of our scheme to degrade to a small extent when we train our classifiers with the less difficult dataset and test with difficult ones and vice versa.

We have categorized our cross dataset analysis in 3 cases:

- Case 1: Train with VX Heavens
- Case 2: Train with Malfease
- Case 3: Train with Offensive

Table 11. Cross Dataset Analysis with PE-Miner

	Train: VX Heavens		Train: Malfease		Train: Offensive	
	Test		Test		Test	
Classifier	Malfease	Offensive	VX Heavens	Offensive	VX Heavens	Malfease
PE-Miner						
IBK	0.949	0.866	0.904	0.894	**0.998**	**0.999**
J48	**0.983**	0.937	0.945	**0.896**	0.98	0.982
NB	0.957	0.84	0.951	0.649	0.921	0.902
RIPPER	0.777	0.848	**0.963**	0.886	0.95	0.95
SMO	0.936	**0.972**	0.915	0.87	0.928	0.721
PE-Miner (RFR)						
IBK	0.949	0.884	0.904	0.894	**0.998**	**0.999**
J48	**0.983**	**0.937**	0.945	0.878	0.98	0.982
NB	0.957	0.854	0.951	0.864	0.921	0.902
RIPPER	0.975	0.874	**0.964**	**0.929**	0.955	0.956
SMO	0.936	0.893	0.915	0.869	0.723	0.721

According to the difficulty level of the datasets, we are expecting average performance with lowest AUCs in Case 1 and best performance with highest AUCs in Case 3.

We have used all of our selected classifiers to first train with one dataset and then test the other two and vice versa. In this way our classification would not be biased with respect to the dataset and the actual effect of the structural feature analysis would be observed. We have performed the cross dataset experiments with two approaches in mind. Firstly, we have used PE-Miner without any preprocessing filter and secondly, we have used PE-Miner with RFR, in order to check the effect of preprocessing filter while testing new datasets. It was not possible to apply preprocessing filters of PCA and HWT, as the features filtered from one dataset do not match with the features obtained from the other one, which results in dataset mismatch. The results of our experiments are shown in Table 11 and Figure 6, 7. We observe interesting patterns of varying AUC with respect to the datasets.

6.4.1 CASE 1 - Train with VX Heavens

In this case we train the classifiers with VX Heavens dataset and test Malfease and Offensive datasets with the obtained model.

We observe that when we test Malfease dataset we get the AUC upto 0.983 with J48 using PE-Miner and 0.972 with SMO using PE-Miner with RFR. Also, the performance of RIPPER with PE-Miner without preprocessing is significantly degraded having 0.777 AUC. This is due to the fact that without optimized feature set many malware programs are being misclassified as benign. It is interesting to note that with PE-Miner (RFR), RIPPER gives an AUC of 0.975. This shows the impact of feature preprocessing in our approach. The rule set of RIPPER with PE-Miner has 13 rules and that of PE-Miner (RFR) has 12 rules. There is no significant effect of RFR processing on AUCs obtained from J48, IBK, NB, and SMO.

The performance of the system further degrades when Offensive dataset is tested with the VX Heavens trained model as we expected. The Offensive testing being the most difficult one follows Malfease one with respect to preprocessing filter.

6.4.2 CASE 2 - Train with Malfease

In this case we train the classifiers with Malfease dataset and test with VX Heavens and Offensive. As per our expectation we see better AUCs with VX Heavens, being the least difficult dataset and degraded AUCs with Offensive.

RIPPER shows the best AUC when we train it with Malfease and test VX Heavens on it. Decision tree J48 with AUC 0.945 closely follow the AUC of RIPPER 0.963.

As in Case 1, only the performance of RIPPER has improved with the preprocessing filter, while the other classifiers are unaffected. The testing of Offensive

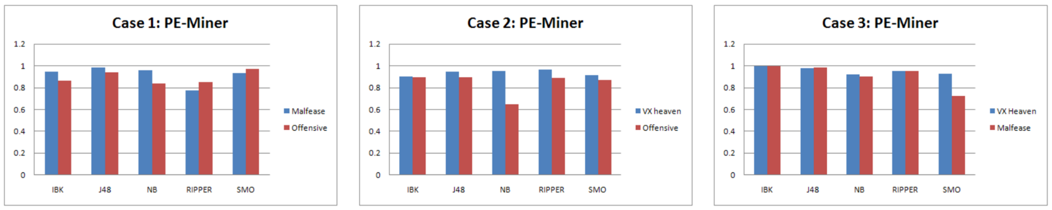

Figure 6. AUCs of PE-Miner

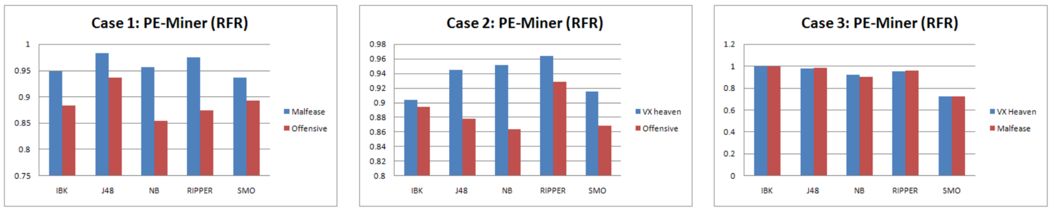

Figure 7. AUCs of PE-Miner (RFR)

dataset also shows that with optimized feature set, RIPPER out performs all the other classifiers, with quite a margin.

6.4.3 CASE 3 - Train with Offensive

In this case we train the classifiers with randomly selected thirty thousand malware samples of Offensive dataset and test VX Heavens and Malfease with this model. This case shows best accuracies as compared to previous two cases, as we expected. The instance based classifier IBk outperforms rest of the classifiers with AUC 0.99 on testing VX Heavens and Malfease.

The AUC show that feature preprocessing does not plays a significant rule when the classifiers have a very extensive train set. However, we observe that the performance of SMO suffers quite a lot after the preprocessing filter. We hope to have even better results when we train the classifiers with complete Offensive dataset.

6.4.4 Discussion

As discussed earlier Offensive and Malfease datasets are supposed to be difficult and more challenging datasets than VX heavens Our packed/non-packed analysis of the datasets shows that VX Heavens virus collection is relatively more balanced than the Malfease dataset. As we had already discussed in Chapter 5, the VX Heavens Virus collection contains 40.1% packed and 47.2% non-packed PE files, where as approximately 12.7% malicious PE files cannot be classified as either packed or non-packed by PEiD and Protection ID. On the other hand, the Malfease collection contains 46.6% packed and 27.2% non-packed malicious PE files. Similarly, 26.2% malicious PE files cannot be classified as packed or non-packed. We have not done the packed/non-packed analysis of Offensive Dataset in this study but we assume it to be evenly distributed like VX Heavens.

Based on the assumed difficulty level of the datasets, with Malfease being more challenging than VX Heavens, we expected to see deteriorating performance of PE-Miner in Case 1 and like wise in Case 2. We observe best performance in Case 3 with AUCs reaching 0.999. Interestingly, when we test Offensive with VX Heavens training model, the detection accuracy is better than that of Malfease training model. This can mean two things, one the malware samples of VX Heavens are similar with Offensive, and second the packed/non-packed samples in Malfease are not evenly distributed and thus average performance is seen.

It is worth noting that preprocessing filter does not contribute much towards the accuracies but with certain exceptions like RIPPER in Case 1 and SMO in Case 3 we get good AUCs upto 0.975 and 0.929 as compared to 0.777 and 0.866 respectively .

6.5 Preprocessing Algorithm Evaluation

We performed another set of experiments to gain insights into the effect of varying parameters of the pre-processing algorithms on the AUC. For this purpose we analyzed the algorithms to select the parameters that would be varied. They are explained as follows:

6.5.1 Redundant Feature Removal (RFR)

The only parameter in RFR is **maximumVariancePercentageAllowed**, which set the threshold for the highest variance allowed before a nominal attribute will be deleted. Specifically, if (number_of_distinct_values / total_number_of_values * 100) is greater than this value then the attribute will be removed. We varied this parameter from 99.0 to 1 and the number of removed features stayed the same, i.e only 6 features removed. This essentially means that the AUCs remained the same.

6.5.2 Haar Wavelet Transform (HWT)

The Haar wavelet transform does not have any parameter's that we can tweak to improve accuracy.

6.5.3 Principal Component Analysis (PCA)

This filter performs a principal components analysis and transformation of the data. Dimensionality reduction is accomplished by choosing enough eigenvectors to account for some percentage of the variance in the original data – default 0.95 (95%). The important parameter related with the dimensionality reduction is **varianceCovered**, which is a measure to retain enough PC attributes to account for this proportion of variance.

We varied this parameter from 0.99 to 0.5 and the AUC's obtained are shown in Table 12.

Table 12. Varying the parameter 'variance' of PCA algorithm

Variance	0.99	0.95	0.9	0.85	0.8	0.75	0.7	0.65	0.6	0.55	0.5
Backdoor + Sniffer											
IBK	0.978	0.974	0.981	0.981	0.983	0.983	0.985	0.986	0.987	0.987	0.985
J48	0.986	0.98	0.98	0.978	0.98	0.977	0.978	0.978	0.98	0.982	0.984
NB	0.981	0.961	0.961	0.961	0.972	0.976	0.978	0.984	0.986	0.988	0.986
SMO	0.989	0.99	0.989	0.987	0.989	0.986	0.985	0.986	0.984	0.983	0.977
Ripper	0.982	0.983	0.982	0.979	0.981	0.98	0.984	0.986	0.986	0.985	0.987
Constructor & Virtool											
IBK	0.978	0.983	0.981	0.985	0.984	0.984	0.987	0.992	0.988	0.99	0.984
J48	0.978	0.966	0.969	0.968	0.971	0.974	0.978	0.981	0.982	0.982	0.984
NB	0.994	0.99	0.989	0.987	0.99	0.987	0.987	0.986	0.987	0.986	0.983
SMO	0.993	0.992	0.993	0.993	0.989	0.991	0.987	0.99	0.982	0.977	0.967
Ripper	0.98	0.978	0.973	0.979	0.985	0.985	0.97	0.973	0.983	0.98	0.983
Exploit + Hacktool											
IBK	0.987	0.981	0.984	0.983	0.983	0.98	0.979	0.985	0.991	0.968	0.956
J48	0.933	0.929	0.937	0.921	0.916	0.915	0.926	0.931	0.943	0.933	0.95
NB	0.995	0.994	0.99	0.984	0.985	0.969	0.961	0.966	0.955	0.951	0.946
SMO	0.995	0.989	0.987	0.985	0.986	0.984	0.977	0.973	0.965	0.891	0.849
Ripper	0.956	0.963	0.964	0.952	0.955	0.964	0.954	0.958	0.952	0.96	0.937
Email- + IM- + SMS Flooder											
IBK	0.986	0.985	0.985	0.981	0.984	0.987	0.985	0.986	0.985	0.979	0.973
J48	0.955	0.96	0.96	0.944	0.942	0.946	0.961	0.961	0.96	0.951	0.945
NB	0.996	0.995	0.995	0.994	0.994	0.994	0.993	0.994	0.995	0.988	0.985
SMO	0.994	0.991	0.991	0.992	0.984	0.982	0.983	0.979	0.98	0.942	0.928
Ripper	0.96	0.981	0.981	0.973	0.957	0.973	0.956	0.971	0.958	0.972	0.951
Email- + IM- + IRC- + Net Worm											
IBK	0.979	0.982	0.984	0.984	0.983	0.987	0.99	0.99	0.988	0.988	0.988
J48	0.987	0.985	0.984	0.987	0.985	0.981	0.981	0.98	0.981	0.982	0.983
NB	0.991	0.989	0.99	0.989	0.986	0.989	0.988	0.984	0.985	0.988	0.991
SMO	0.99	0.992	0.991	0.992	0.991	0.991	0.99	0.99	0.989	0.987	0.986
Ripper	0.989	0.987	0.991	0.99	0.988	0.991	0.991	0.99	0.992	0.991	0.985
DoS + Nuker											
IBK	0.986	0.989	0.989	0.984	0.986	0.988	0.99	0.986	0.992	0.971	0.968
J48	0.987	0.987	0.987	0.986	0.987	0.99	0.99	0.992	0.992	0.98	0.977
NB	0.996	0.996	0.995	0.993	0.992	0.984	0.981	0.979	0.988	0.975	0.975
SMO	0.996	0.995	0.996	0.998	0.997	0.998	0.996	0.998	0.997	0.924	0.911
Ripper	0.969	0.977	0.966	0.968	0.971	0.984	0.978	0.972	0.979	0.953	0.953
Trojan											
IBK	0.954	0.952	0.952	0.955	0.958	0.958	0.962	0.96	0.961	0.963	0.964
J48	0.933	0.936	0.939	0.94	0.943	0.939	0.941	0.946	0.951	0.948	0.953
NB	0.965	0.963	0.963	0.96	0.96	0.964	0.967	0.962	0.951	0.954	0.949
SMO	0.966	0.958	0.955	0.953	0.948	0.95	0.948	0.947	0.95	0.943	0.938
Ripper	0.948	0.949	0.953	0.958	0.953	0.952	0.955	0.949	0.957	0.95	0.952
Virus											
IBK	0.964	0.962	0.968	0.968	0.968	0.966	0.969	0.969	0.972	0.969	0.959
J48	0.952	0.951	0.956	0.956	0.959	0.953	0.963	0.966	0.965	0.956	0.965
NB	0.956	0.956	0.955	0.955	0.948	0.947	0.947	0.946	0.949	0.946	0.938
SMO	0.967	0.965	0.961	0.958	0.957	0.955	0.952	0.946	0.946	0.945	0.937
Ripper	0.969	0.968	0.963	0.968	0.967	0.972	0.972	0.966	0.97	0.971	0.97

6.6 Limitations and Potential Solutions

In this section, we present the limitations of our PE-Miner framework and their potential solutions.

- We acknowledge that the features proposed in our study are simple and if adversaries know our detection methodology then they can design strategies to evade detection by PE-Miner. For example, the number of resources (such as bitmaps, icons, etc.) is more in the benign executables as compared to those of the malicious executables. A crafty attacker may insert dummy resources in the malicious executables to increase the number of resources. But doing so will not only increase the size of the malware but also its processing overheads. An increase in the size of a malware significantly limits its capability to spread via replication without getting noticed. It also compromises its ability to fit into size constrained buffers in the buffer overflow attacks. Furthermore, PE-Miner uses a large number of features, covering structural information of all portions of a PE file, which makes it very difficult for an attacker to manipulate most of them at the same time. It is relevant to mention that simple structural features used by PE-Miner are not affected by code obfuscation and restructuring techniques. A majority of new malware that appears today consists of the variants of existing malware generated via such techniques . Therefore, PE-Miner can prove more effective in detecting zero-day malware.

- The strings technique proposed by Schultz et al and the KM technique proposed by Kolter et al are not dependent on the executable file format. Therefore, their techniques can possibly scale to malware targeted for UNIX and other non-Windows operating systems. However, PE-Miner is specific for the executables in the PE format which is used by Windows operating systems.

We envision that similar structural features can be extracted from other executables with a different file format and then PE-Miner can be ported to other operating systems as well.

- If a malware uses good packing techniques, then it can circumvent the DLL features of PE-Miner. However, PE-Miner uses a host of other structural features which are still able to provide important information for detecting malicious PE files. Our experiments on packed executables also confirm that PE-Miner is robust to different packing techniques.

CHAPTER 7
CONCLUSION & FUTURE WORK

In this thesis we present, PE-Miner, a technique for detection of malicious PE files. PE-Miner leverages the structural information of PE files and data mining algorithms to provide high detection accuracy with low processing overheads. Our implementation of PE-Miner completes a single-pass scan of all executables in the dataset (more than 17 thousand) in less than one hour. Therefore it meets all of our requirements mentioned in Chapter 2.

Our question oriented research methodology helped us in extensively searching almost all dimensions in the design space and the conclusion of the work is that three design options provide the best combination for detecting malicious PE files: (1) a rich PE format specific set of structural features which can be statically extracted from a PE file, (2) the preprocessing filters help in reducing the training and testing time of a classifier but they have minor role in improving the detection accuracy, (3) J48 classifier gives the best detection accuracy with low processing overheads. Therefore, our final PE-Miner framework has RFR preprocessing filter with J48 as the back-end classifier.

We believe that PE-Miner framework is ideally suited for detecting malicious PE files on resource constrained mobile phones running Windows CE. Our results are intriguing enough to port it to Windows CE mobile devices. Moreover, we also plan to evaluate PE-Miner on a much larger executable dataset (with size of 140 GB) that we have just obtained from `offensivecomputing.net`.

LIST OF REFERENCES

[1] W. L. R. Perdisci, A. Lanzi, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Annual Computer Security Applications Conference (ACSAC)*. IEEE Press, 2008, pp. 301–310.

[2] E. Z. S. S. M.G. Schultz, E. Eskin, "Data mining methods for detection of new malicious executables," in *IEEE Symposium on Security and Privacy (S&P)*, 2001, pp. 38–49.

[3] M. M. J.Z. Kolter, "Learning to detect malicious executables in the wild," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, USA, 2004, pp. 470–478.

[4] F. Veldman, "Heuristic anti-virus technology," in *International Virus Bulletin Conference*, USA, 1993, pp. 67–76.

[5] J. Munro, "Antivirus research and detection techniques," ExtremeTech, 2002. [Online]. Available: http://www.extremetech.com/article2/0,2845,367051,00.asp

[6] "Symantec internet security threat reports i-xi (jan 2002-jan 2008)."

[7] F.-S. Corporation, "F-secure reports amount of malware grew by 1002007," Press Release, 2007.

[8] H. Y. S. L. J. Cheng, S.H.Y. Wong, "Smartsiren: virus detection and alert for smartphones," in *International Conference on Mobile Systems, Applications and Services (MobiSys)*, USA, 2007, pp. 258–271.

[9] "Vx heavens virus collection." [Online]. Available: http://hvx.netlux.org

[10] "Project malfease." [Online]. Available: http://malfease.oarci.net/

[11] S. Walter, "The partial area under the summary roc curve," *Statistics in Medicine*, vol. 24(13), pp. 2025–2040, 2005.

[12] C. M. K. Kendall, "Practical malware analysis," in *Black Hat Conference*, USA, 2007.

[13] Windows Hardware Developer Central. "Microsoft portable executable and common object file format specification." March 2008. [Online]. Available: http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[14] "Pe file format," Webster Technical Documentation. [Online]. Available: http://webster.cs.ucr.edu/Page_TechDocs/pe.txt

[15] M. Help and Support, "Dumpbin utility, article id 177429, revision 4.0," 2005.

[16] M. Pietrek, "An in-depth look into the win32 portable executable file format, part 2," MSDN Magazine, March 2002.

[17] E. F. I.H. Witten, *Data mining: Practical machine learning tools and techniques*, 2nd ed. USA: Morgan Kaufmann, 2005.

[18] M. A. D.W. Aha, D. Kibler, "Instance-based learning algorithms," *Journal of Machine Learning*, vol. 6, pp. 37–66, 1991.

[19] J. Quinlan, "C4.5: Programs for machine learning," Morgan Kaufman, 1993.

[20] J. K. M.E. Maron, "On relevance, probabilistic indexing and information retrieval," *Journal of the Association of Computing Machinery*, vol. 7(3), pp. 216–123, 1960.

[21] W. Cohen, "Fast effective rule induction," in *International Conference on Machine Learning*, USA, 1995, pp. 115–123.

[22] J. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods–Support Vector Learning.* USA: MIT Press, 1998, pp. 185–208.

[23] W. A. D. C. G. T. S. W. J. Kephart, G. Sorkin, "Biologically inspired defenses against computer viruses," in *International Joint Conference on Artificial Intelligence (IJCAI)*, USA, 1995, pp. 985–996.

[24] R. O. R.W. Lo, K.N. Levitt, "Mcf: A malicious code filter," *Elsevier Computers & Security*, vol. 14(6), pp. 541–566, 1995.

[25] T. L. D. Y. Q. J. Y. Ye, D. Wang, "An intelligent pe-malware detection system based on association mining," *Journal in Computer Virology*, pp. 323–334, 2008.

[26] N. J. O. Henchiri, "A feature selection and evaluation scheme for computer virus detection," in *IEEE International Conference on Data Mining (ICDM)*, USA, 2006, pp. 891–895.

[27] P. G. P. Kierski, M. Okoniewski, "Automatic classification of executable code for computer virus detection," in *International Conference on Intelligent Information Systems (IIS).* Poland: Springer, 2003, pp. 277–284.

[28] V. K. R. S. T. A.-Assaleh, N. Cercone, "Detection of new malicious code using n-grams signatures," in *International Conference on Intelligent Information Systems (IIS).* Poland: Springer, 2003, pp. 193–196.

[29] P. D. J.H.Wang, "Virus detection using data mining techniques," in *IEEE International Carnahan Conference on Security Technology (ICCST)*, China, 2003, pp. 71–76.

[30] W. L. S.J. Stolfo, K. Wang, "Towards stealthy malware detection," *Advances in Information Security*, vol. 27, pp. 231–249, 2007.

[31] A. S. E. A. K. W.J. Li, S.J. Stolfo, "A study of malcode-bearing documents," in *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA).* Switzerland: Springer, 2007, pp. 231–250.

[32] M. F. M.Z. Shafiq, S.A. Khayam, "Embedded malware detection using markov n-grams," in *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA).* France: Springer, 2008, pp. 88–107.

[33] M. M. E. . B. K. J. Bergeron, M. Debbabi, "Static analysis of binary code to isolate malicious behaviors," in *Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, 1999.

[34] J. D. M. M. E. Y. L. J. Bergeron, M. Debbabi and N. Tawbi., "Static detection of malicious code in executable programs," in *Symposium on Requirements Engineering for Information Security*, 2001.

[35] C. Kruegel and E. Kirda, "Automating mimicry attacks using static binary analysis," in *USENIX Security*, 2005.

[36] G. B. G. V. Engin Kirda, Christopher Kruegel and R. A. Kemmerer, "Behavior-based spyware detection," Technical University Vienna and University of California, Santa Barbara, Tech. Rep., 2006.

[37] D. D. P. B. R. Sekar, M. Bendre, "A fast automaton-based method for detecting anomalous program behaviors," in *IEEE Symposium on Security and Privacy*, 2001.

[38] A. Singh, P.K. Lakhotia, "Static verification of worm and virus behavior in binary executables using model checking," in *IEEE Information Assurance Workshop*, 2003.

[39] S. D. Cullen Linn, "Obfuscation of executable code to improve resistance to static disassembly," in *Conference on Computer and Communications Security*, 2003.

[40] T. S. Akira Mori, Tomonori Izumida and T. Inoue, "A tool for analyzing and detecting malicious mobile code," in *International Conference on Software Engineering*, 2006.

[41] G. V. Engin Kirda, Christopher Kruegel and N. Jovanovic, “Noxes: A client-side solution for mitigating cross-site scripting attacks,” in *21st ACM Symposium on Applied Computing (SAC)*, 2006.

[42] D. D. R. E. W. L. Paul Royal, Mitch Halpin, “Automating the hidden-code extraction of unpack-executing malware,” in *Annual Computer Security Applications Conference*, 2004.

[43] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham, “Detection of injected, dynamically generated, and obfuscated malicious code,” in *The ACM Workshop on Rapid Malcode (WORM)*, 2003.

[44] E. H. S. S. Sandeep Kumar, “A generic virus scanner in c++,” in *The 8th Computer Security Applications Conference*, 1992.

[45] P. C. S. M. A. H. Sung, J. Xu, “Static analyzer of vicious executables (save),” in *The Annual Computer Security Applications Conference*, 2004.

[46] S. G. Yoshinori Okazaki, Izuru Sato, “A new intrusion detection method based on process profiling,” in *Symposium on Applications and the Internet*, 2002.

[47] M. F. S. Momina Tabish, M. Zubair Shafiq, “Malware detection using statistical analysis of byte-level file content,” in *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, France, 2009, pp. 23–31.

[48] R. H. T. C. Y. Z. M. R. Stewart Yang, Jianping Song, “Fast and effective worm fingerprinting via machine learning,” in *IEEE International Conference on Autonomic Computing*, 2006.

[49] W. L. R. Perdisci, A. Lanzi, “Classification of packed executables for accurate computer virus detection,” *Elsevier Pattern Recognition Letters*, vol. 29(14), pp. 1941–1946, 2008.

[50] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference, FREENIX Track,*, 2005, pp. 41–46.

[51] H. Y. M.G. Kang, P. Poosankam, “Renovo: A hidden code extractor for packed executables,” in *ACM Workshop on Recurring Malcode (WORM)*, USA, 2007, pp. 46–53.

[52] “Peid.” [Online]. Available: http://www.peid.info/

[53] “F-prot antivirus.” [Online]. Available: http://www.f-prot.com/

[54] D. D. R. E. W. L. P. Royal, M. Halpin, “Polyunpack: Automating the hidden-code extraction of unpack-executing malware,” in *Annual Computer Security Applications Conference (ACSAC)*, USA, 2006, pp. 289–300.

[55] T. Fawcett, “Roc graphs: Notes and practical considerations for researchers,” HP Labs, Tech. Rep. TR HPL-2003-4, 2004.

[56] “F-secure virus description database.” [Online]. Available: http://www.f-secure.com/v-descs/

[57] “Protection id - the ultimate protection scanner.” [Online]. Available: http://pid.gamecopyworld.com/

[58] Visual $C++$ Standard Library. “Hash map.” [Online]. Available: http://msdn.microsoft.com/en-us/library/6x7w9f6z.aspx

[59] S. Axelsson, “The base-rate fallacy and its implications for the difficulty of intrusion detection,” in *ACM Conference on Computer and Communications Security (CCS)*, Singapore, 1999, pp. 1–7.

[60] “Avg free antivirus.” [Online]. Available: http://www.avg.com/

[61] “Panda antivirus.” [Online]. Available: http://www.pandasecurity.com/

[62] “Offensive computing.” [Online]. Available: http://www.offensivecomputing.net/

BIBLIOGRAPHY

"Avg free antivirus." [Online]. Available: http://www.avg.com/

"F-prot antivirus." [Online]. Available: http://www.f-prot.com/

"F-secure virus description database." [Online]. Available: http://www.f-secure.com/v-descs/

"Offensive computing." [Online]. Available: http://www.offensivecomputing.net/

"Panda antivirus." [Online]. Available: http://www.pandasecurity.com/

"Pe file format," Webster Technical Documentation. [Online]. Available: http://webster.cs.ucr.edu/Page_TechDocs/pe.txt

"Peid." [Online]. Available: http://www.peid.info/

"Project malfease." [Online]. Available: http://malfease.oarci.net/

"Protection id - the ultimate protection scanner." [Online]. Available: http://pid.gamecopyworld.com/

"Symantec internet security threat reports i-xi (jan 2002-jan 2008)."

"Vx heavens virus collection." [Online]. Available: http://hvx.netlux.org

A. H. Sung, J. Xu, P. C. S. M., "Static analyzer of vicious executables (save)," in *The Annual Computer Security Applications Conference*, 2004.

Akira Mori, Tomonori Izumida, T. S. and Inoue, T., "A tool for analyzing and detecting malicious mobile code," in *International Conference on Software Engineering*, 2006.

Axelsson, S., "The base-rate fallacy and its implications for the difficulty of intrusion detection," in *ACM Conference on Computer and Communications Security (CCS)*, Singapore, 1999, pp. 1–7.

Bellard, F., "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track,*, 2005, pp. 41–46.

Cohen, W., "Fast effective rule induction," in *International Conference on Machine Learning*, USA, 1995, pp. 115–123.

Corporation, F.-S., "F-secure reports amount of malware grew by 1002007," Press Release, 2007.

Cullen Linn, S. D., "Obfuscation of executable code to improve resistance to static disassembly," in *Conference on Computer and Communications Security*, 2003.

D.W. Aha, D. Kibler, M. A., "Instance-based learning algorithms," *Journal of Machine Learning*, vol. 6, pp. 37–66, 1991.

Engin Kirda, Christopher Kruegel, G. V. and Jovanovic, N., "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *21st ACM Symposium on Applied Computing (SAC)*, 2006.

Engin Kirda, Christopher Kruegel, G. B. G. V. and Kemmerer, R. A., "Behavior-based spyware detection," Technical University Vienna and University of California, Santa Barbara, Tech. Rep., 2006.

Fawcett, T., "Roc graphs: Notes and practical considerations for researchers," HP Labs, Tech. Rep. TR HPL-2003-4, 2004.

Help, M. and Support, "Dumpbin utility, article id 177429, revision 4.0," 2005.

I.H. Witten, E. F., *Data mining: Practical machine learning tools and techniques*, 2nd ed. USA: Morgan Kaufmann, 2005.

J. Bergeron, M. Debbabi, J. D. M. M. E. Y. L. and Tawbi., N., "Static detection of malicious code in executable programs," in *Symposium on Requirements Engineering for Information Security*, 2001.

J. Bergeron, M. Debbabi, M. M. E. . B. K., "Static analysis of binary code to isolate malicious behaviors," in *Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, 1999.

J. Cheng, S.H.Y. Wong, H. Y. S. L., "Smartsiren: virus detection and alert for smartphones," in *International Conference on Mobile Systems, Applications and Services (MobiSys)*, USA, 2007, pp. 258–271.

J. Kephart, G. Sorkin, W. A. D. C. G. T. S. W., "Biologically inspired defenses against computer viruses," in *International Joint Conference on Artificial Intelligence (IJCAI)*, USA, 1995, pp. 985–996.

J.H.Wang, P. D., "Virus detection using data mining techniques," in *IEEE International Carnahan Conference on Security Technology (ICCST)*, China, 2003, pp. 71–76.

J.Z. Kolter, M. M., "Learning to detect malicious executables in the wild," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, USA, 2004, pp. 470–478.

K. Kendall, C. M., "Practical malware analysis," in *Black Hat Conference*, USA, 2007.

Kruegel, C. and Kirda, E., "Automating mimicry attacks using static binary analysis," in *USENIX Security*, 2005.

M.E. Maron, J. K., "On relevance, probabilistic indexing and information retrieval," *Journal of the Association of Computing Machinery*, vol. 7(3), pp. 216–123, 1960.

M.G. Kang, P. Poosankam, H. Y., "Renovo: A hidden code extractor for packed executables," in *ACM Workshop on Recurring Malcode (WORM)*, USA, 2007, pp. 46–53.

M.G. Schultz, E. Eskin, E. Z. S. S., "Data mining methods for detection of new malicious executables," in *IEEE Symposium on Security and Privacy (S&P)*, 2001, pp. 38–49.

Munro, J., "Antivirus research and detection techniques," ExtremeTech, 2002. [Online]. Available: http://www.extremetech.com/article2/0,2845,367051,00.asp

M.Z. Shafiq, S.A. Khayam, M. F., "Embedded malware detection using markov n-grams," in *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA)*. France: Springer, 2008, pp. 88–107.

O. Henchiri, N. J., "A feature selection and evaluation scheme for computer virus detection," in *IEEE International Conference on Data Mining (ICDM)*, USA, 2006, pp. 891–895.

P. Kierski, M. Okoniewski, P. G., "Automatic classification of executable code for computer virus detection," in *International Conference on Intelligent Information Systems (IIS)*. Poland: Springer, 2003, pp. 277–284.

P. Royal, M. Halpin, D. D. R. E. W. L., "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Annual Computer Security Applications Conference (ACSAC)*, USA, 2006, pp. 289–300.

Paul Royal, Mitch Halpin, D. D. R. E. W. L., "Automating the hidden-code extraction of unpack-executing malware," in *Annual Computer Security Applications Conference*, 2004.

Pietrek, M., "An in-depth look into the win32 portable executable file format, part 2," MSDN Magazine, March 2002.

Platt, J., "Fast training of support vector machines using sequential minimal optimization," in *Advances in Kernel Methods–Support Vector Learning*. USA: MIT Press, 1998, pp. 185–208.

Quinlan, J., “C4.5: Programs for machine learning,” Morgan Kaufman, 1993.

R. Perdisci, A. Lanzi, W. L., “Classification of packed executables for accurate computer virus detection,” *Elsevier Pattern Recognition Letters*, vol. 29(14), pp. 1941–1946, 2008.

R. Perdisci, A. Lanzi, W. L., “Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables,” in *Annual Computer Security Applications Conference (ACSAC).* IEEE Press, 2008, pp. 301–310.

R. Sekar, M. Bendre, D. D. P. B., “A fast automaton-based method for detecting anomalous program behaviors,” in *IEEE Symposium on Security and Privacy*, 2001.

Rabek, J. C., Khazan, R. I., Lewandowski, S. M., and Cunningham, R. K., “Detection of injected, dynamically generated, and obfuscated malicious code,” in *The ACM Workshop on Rapid Malcode (WORM)*, 2003.

R.W. Lo, K.N. Levitt, R. O., “Mcf: A malicious code filter,” *Elsevier Computers & Security*, vol. 14(6), pp. 541–566, 1995.

S. Momina Tabish, M. Zubair Shafiq, M. F., “Malware detection using statistical analysis of byte-level file content,” in *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, France, 2009, pp. 23–31.

Sandeep Kumar, E. H. S. S., “A generic virus scanner in c++,” in *The 8th Computer Security Applications Conference*, 1992.

Singh, P.K. Lakhotia, A., “Static verification of worm and virus behavior in binary executables using model checking,” in *IEEE Information Assurance Workshop*, 2003.

S.J. Stolfo, K. Wang, W. L., “Towards stealthy malware detection,” *Advances in Information Security*, vol. 27, pp. 231–249, 2007.

Stewart Yang, Jianping Song, R. H. T. C. Y. Z. M. R., “Fast and effective worm fingerprinting via machine learning,” in *IEEE International Conference on Autonomic Computing*, 2006.

T. A.-Assaleh, N. Cercone, V. K. R. S., “Detection of new malicious code using n-grams signatures,” in *International Conference on Intelligent Information Systems (IIS).* Poland: Springer, 2003, pp. 193–196.

Veldman, F., “Heuristic anti-virus technology,” in *International Virus Bulletin Conference*, USA, 1993, pp. 67–76.

Visual $C++$ Standard Library. “Hash map.” [Online]. Available: http://msdn.microsoft.com/en-us/library/6x7w9f6z.aspx

Walter, S., “The partial area under the summary roc curve,” *Statistics in Medicine*, vol. 24(13), pp. 2025–2040, 2005.

Windows Hardware Developer Central. “Microsoft portable executable and common object file format specification.” March 2008. [Online]. Available: http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

W.J. Li, S.J. Stolfo, A. S. E. A. K., “A study of malcode-bearing documents,” in *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA).* Switzerland: Springer, 2007, pp. 231–250.

Y. Ye, D. Wang, T. L. D. Y. Q. J., “An intelligent pe-malware detection system based on association mining,” *Journal in Computer Virology*, pp. 323–334, 2008.

Yoshinori Okazaki, Izuru Sato, S. G., “A new intrusion detection method based on process profiling,” in *Symposium on Applications and the Internet*, 2002.