A Framework for Malware Detection with Static Features using Machine Learning Algorithms

A THESIS

Submitted by

Ajit Kumar

in partial fulfillment of the requirements for the award of the degree

of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF ENGINEERING AND TECHNOLOGY
PONDICHERRY UNIVERSITY
PUDUCHERRY - 605014

AUGUST 2017

Contents

Certificate

This is to certify that this thesis titled ***"A Framework for Malware Detection with Static Features using Machine Learning Algorithms"*** *submitted by* ***Mr. Ajit kumar****, to the Department of Computer Science, School of Engineering and Technology, Pondicherry University, Puducherry, India for the award of the degree of* ***Doctor of Philosophy*** *in* ***Computer Science and Engineering*** *is a record of bonafide research work carried out by him under our guidance and supervision.*

This work is original and has not been submitted, in part or full to this or any other University/Institution for the award of any other degree.

Dr. K.S. Kuppusamy.,Ph.D
(Co-Supervisor)

Prof. G. Aghila.,B.E.(Hons.),M.E.,Ph.D
(Supervisor)

Department of Computer Science,
School of Engineering and Technology,
Pondicherry University,
Puducherry-605014, India.

Place: Puducherry
Date :

Declaration

I hereby declare that this thesis titled ***"A Framework for Malware Detection with Static Features using Machine Learning Algorithms"*** *submitted to the Department of Computer Science, School of Engineering and Technology, Pondicherry University, Puducherry, India for the award of the degree of* ***Doctor of Philosophy*** *in* ***Computer Science and Engineering*** *is a record of bonafide research work carried out by me under the guidance and supervision of* ***Prof. G. Aghila*** *and* ***Dr. K.S Kuppusamy****.*

This work is original and has not been submitted, in part or full to this or any other University/Institution for the award of any other degree.

Place: Puducherry

Date : **Ajit Kumar**

Acknowledgement

I thank my parents without whom this work would not have been possible. I also would like to thank my wonderful sisters who had been very cooperative and gave me unsolicited affection and emotional support during the course of my education and the Research Work. I would like to acknowledge my thanks to all my family members who have supported me and their encouragement.

It is a genuine pleasure to express my deep sense of thanks and gratitude to my Ph.D. Supervisor, **Dr. G. Aghila**, Professor, Department of Computer Science and Engineering, National Institute of Technology Puducherry, Karaikal and Co-supervisor, **Dr. K.S. Kuppusamy**, Assistant Professor, Department of Computer Science, School of Engineering and Technology, Pondicherry University, Puducherry, for their support and encouragement throughout the course of this study. Their dedication, invaluable suggestions and keen interest above all their overwhelming attitude to help had been solely and mainly responsible for completing my work. Timely advice, meticulous scrutiny and scholarly advice which I recieved from them during the study and the preparation of this thesis empowered me to present in this form.

Besides my supervisor and co-supervisor, I would like to express my thanks to my doctoral committee members: **Dr. H.P. Patil** and **Dr. S.Kanmani** for their invaluable suggestions, encouragement and for the thoughtful questions which incented me to improve my research from various perspectives.

I sincerely thank **Prof. P.Dhanavanthan**, Dean of School of Engineering and Technology and **Dr. T. Chithralekha** Head i/c of the Department of Computer Science, Pondicherry University for providing institutional support to carry out the research work.

I also want to thank my former teachers **Dr. S. Sivasathya**, **Dr. R. Sunitha**, **Dr. V. Uma**, **Dr. P. Shanthi Bala**, **T. Sivakumar** and all of my JNV Shekhpura teachers. Besides my former teachers, I would like to thanks **Dr. P. Thiyagarajan**

and **Mr. Suprabhat Kumar** who taught me life lessons and guided through all ups and downs of my life.

I am also thankful to **Mr. K. Suresh**, senior technical assistant, who maintained the machine in my lab so efficiently that I never had to worry about not having my own computer.

I also would like to express my deepest gratitude to the Honorable Vice-Chancellor (Officiating) of Pondicherry University, **Prof.) Anisa Basheer Khan**, the Registrar and the Assistant Registrar of all the departments, and Director (Research) at Pondicherry University, Puducherry for their official support to make this thesis in time.

I express my deepest gratitude to office staff of my department for their support. I extend my thanks towards the staffs of Ananda Rangapillai Library of Pondicherry University for their service and efforts to conduct many workshop and seminar related to research, which helped to furnish the research in the form of thesis.

I also would like to thank all my colleagues and friends including but not limited to, Mr. Agam Kumar, Mr. Rajesh Kumar, Mr. Uday kumar, Dr. Smarak Samarjeet, Mr. Naveen Kumar, Dr. Richa Mishra, Mr. Rajnish Mishra, Ms. Leena Mary Francis, Mr. Vikash Kumar, Mr. Abhishek Kumar, Mr. Rajesh Kumar, Ms. Jonti Deuri, Dr. Bithin Thakur, Dr. Manu C. Sakaria, Ms. Lois Jose, Mr.Jayapradapan, Dr. Ajay Harit, Mr. Shailesh Khapre, Dr. D. Chandramohan and my lab mates Mr. Balaji V., Mr. Gunikhan Sonowal, Mr. Abid Ismali, Mr. AB Shqoor Menengroo, Mr. Pathula Mulridhar and Mr. Pawan Kumar Ojha for their fruitful support and encouragement during the course of this work.

I dedicate this thesis to my parents, sisters and friends.

Abstract

Malicious software (Malware) are programs written intentionally to harm computing environments. With advancements in computing, the modern malware has also evolved in its evasion and attack techniques. Most of the current malware detection techniques are based on the signature-based method and so suffers from inherent limitations such as (1) inability to detect *unknown* and *zero-day* malware; (2) frequent signature updates and larger signature database; (3) complex and time intensive malware analysis.

Generally, the signature-based methods use a *sequence of bytes* as the signature to detect malware while non-signature-based methods work on profile matching which is also known as Anomaly detection. Among different techniques, machine learning based malware detection has major success and it is able to overcome the limitations of signature-based detection.

Lately, research and development for malware detection techniques are moving towards non-signature based method. The *feature engineering* is the most important step of machine learning based malware detection. The features from malicious and benign programs can be extracted either by static or dynamic methods. The existing research has shown that the features extracted through static method are able to achieve good accuracy with low false positives.

In this research work, a framework for detection of the malicious portable executable and Android apps is developed. The framework is based on static features and used machine learning algorithms to build malware detectors. The resulted malware detectors are capable of grouping *unknown* and *zero-day* sample into its respective class i.e. malware and benign. Under the proposed framework four feature sets are compiled using various sets of features (raw features, derived features, section names, weighted permission, and features from the Image representation of apps) which are created using feature engineering methods. All the four proposed feature sets, namely *integrated feature set* (include raw and derived features), *section-name based feature set*, *weighted*

permission based feature set and *image-based feature set* have been tested and compared with existing feature set.

With the proposed *integrated feature set* built from the portable executable header fields, Random forest achieved the accuracy of 98.4% with 10-fold cross-validation and 89.23% accuracy with test dataset evaluation. The *section-name based feature set* built with only section name of the portable executable file achieved the accuracy of 93% with features having *non-zero information gain score* and 92% accuracy on *top 20 features* with Random forest classifier.

The proposed *weighted permission based feature set* performed better than it's counterpart boolean feature set and was able to achieve 94.84% accuracy with Random forest classifier. In the *image-based feature set* created using GIST features extracted from the grayscale representation of Android apps, the Random forest classifier yields the best result having 91% accuracy and only 8% error rate.

From the experimental result, it is observed that the performance of the *integrated feature set* are better than commonly used raw feature set. For the feature set generated by using *section name,* as the boolean feature, an accuracy of 93% is achieved which clearly indicates the potential of *section name*. Besides the moderate accuracy of *section name* based feature set, it can be used with other features to achieve a better performance.

The weighting of permission improves the performance of the classifiers which can be observed by the experimental results i.e. 94.84% accuracy. The performance of most of the classifiers are better with weighted permission based feature set. The time taken to calculate *permission frequency* for permission scoring is only for the proposed feature set but not for its counterparts feature set. The *permission scoring* is a one-time process so the time consumption can be traded with the improved accuracy. By converting *apk* to *image* it became possible to extract various image-based features which can be used for Android malware detection. With the GIST features an accuracy of 91% is achieved which opens a new dimension for exploring features for Android malware detection. The performance of image-based feature would be improved further by the use of other image features i.e. image descriptor and different image representation.

This thesis explores the potential of four proposed static feature sets with machine learning algorithms for malware detection.

List of Tables

List of Figures

List of Abbreviations

ADB	Android Debug Bridge.
API	Application Programming Interface.
APK	Android Package Kit.
ASCII	American Standard Code for Information Interchange.
AUC	Area Under Curve.
AV	Anti-Virus.
COFF	Common Object File Format.
COM	Component Object Model.
CSV	Comma Separated Value.
CYMK	Cyan, Magenta, Yellow, Key (blacK).
DDoS	Distributed Denial of Service.
DLL	Dynamic Link Library.
DOS	Disk Operating System.
DT	Decision Tree.
ELF	Executable and Linkable Format.
EXE	Executable.
FN	False Negative.
FP	False Positive.
FPR	False Positive Rate.
GUI	Graphical User Interface.

HoG	Histogram of Oriented Gradients.
HPC	Hardware Performance Counter.
HSL	Hue, Saturation, Lightness.
HSV	Hue, Saturation, Value.
IAT	Import Address Table.
IoT	Internet of Things.
IR	Information Retrieval.
kNN	k-Nearest Neighbors.
LDA	Linear Discriminant Analysis.
LR	Logistic Regression.
MASM	Microsoft Macro Assembler.
MD	Message Digest.
ML	Machine Learning.
MSDN	Microsoft Developer Network.
NB	Gaussian Naive Bayes.
NN	Neural Networks.
OOA	Objective Oriented Association.
OS	Operating System.
PE	Portable Executable.
RF	Random Forest.
RGB	Red, Green, Blue.
ROC	Receiver Operating Characteristic.
RVA	Relative Virtual Address.

SD	Standard Deviation.
SHA	Secure Hashing Algorithm.
SIFT	Scale Invariant Feature Transform.
SURF	Speeded Up Robust Features.
SVM	Support Vector Machine.
TN	True Negative.
TP	True Positive.
TPR	True Positive Rate.
URL	Uniform Resource Locator.
USB	Universal Serial Bus.
UTF	Unicode Transformation Format.
VM	Virtual Machine.
XML	Extended Markup Language.

CHAPTER 1

Introduction

This thesis entitled "A Framework for Malware Detection with Static Features using Machine Learning Algorithms" aims to improve malware detection and overcome the limitations of traditional signature-based detection methods. The omnipresence of malware is an established truth and is a critical threat for all kinds of computing environments. This research work focuses on Portable Executable (PE) and Android Package Kit (APK), the file formats for Windows and Android Operating System (OS) respectively. The selection of these two file formats is based on *the highest number of users of respective OS*. Windows and Android have highest number of users with respect to their competitive OSs for Desktop and Smartphone device respectively. This work statically analyzes and uses the structure information of PE file and uses permissions extracted from *manifest* file of app to build feature sets to train and test various machine learning algorithms for malware detection.

1.1 Malware: A Cyber Threat

Today, the World is moving towards digital era where the use of Cyber technologies have become an integral part of daily life. The use of Computer and Internet is not only limited to personnel computation and information access,but its role is also extended to everything by the use of technologies such as Internet of Things (IoT), crypto-currency etc. Currently, the World is talking about *digital economy* to *cyber colonies*, such deep involvement of Computer and various other technologies brings new challenges to the digital World. Everyday, people and firms are witnessing different types of cyber attack

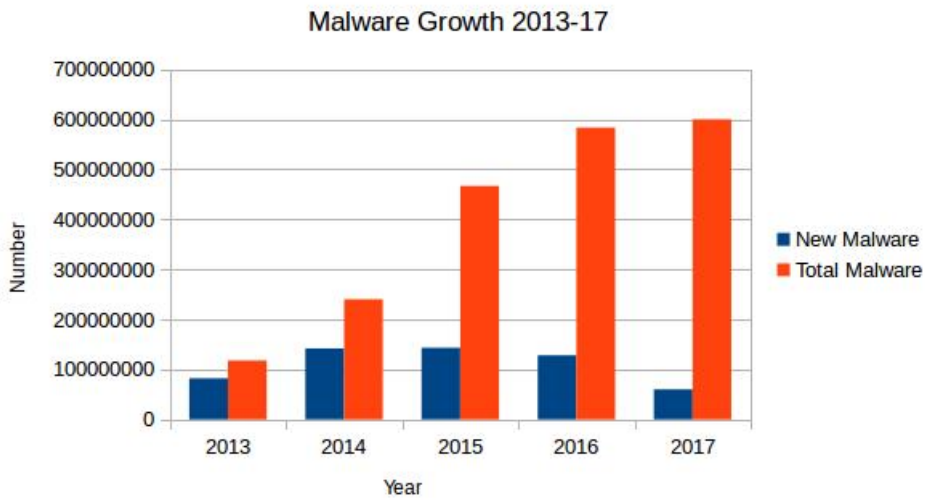


Figure 1.1: Total and new malware during the period 2013-17

on various cyber infrastructure from e-banking to e-commerce [1] (Anderson et al., 2013). Phishing, Distributed Denial of Service (DDoS), identity theft, Ransomware etc. are some of the common attack methods. Many of the modern cyber attacks are carried out by the direct or indirect use of the Malware.

Malicious program, commonly known as Malware is a computer program is intentionally developed for malicious activities, such as attacking computer networks, system hijacks, file deletion, information stealing, spamming and malware downloads (Szor, 2005). The list of malicious activities is very extensive and is growing with new entries with a fast and regular rhythm. For example, *Stuxnet* added Critical-Infrastructure as target to the malicious list (Langner, 2011). There are two classes of malware: the newly produced malware and variants of malware (Yu et al., 2011). Figure 1.1 shows the voluminous growth rate for *total malware* and *new malware* from 2013 to June, 2017 [2].

Malware sophistication in terms of, *self-mutation* (Bruschi et al., 2006), *crypto-virology* (Young and Yung, 1996; Shivale, 2011; Leder et al., 2009), *multi-propagation*, *multi-payload* [3] (example: Nimda worm), *multi-platform* [4] execution and various *obfus-*

[1] `http://www.livemint.com/Industry/MBqlWLIFkpR4W34sdA6TqN/50-cyber-attack-incidents-reported-in-financial-sector-govt.html`

[2] `https://www.av-test.org/en/statistics/malware/`(Accessed:June 2017).

[3] `http://www.sans.org/reading-room/whitepapers/malicious/worm-propagation-countermeasures-1410`,(Accessed: 2017-04-30)

[4] `http://searchsecurity.techtarget.com/magazineContent/Malware-trends-The-`

cation (You and Yim, 2010), and *anti-analysis* techniques (Moser et al., 2007; Bethencourt et al., 2008) are improving on a daily basis. These sophisticated malware and its exponential growth brings major challenge to the available anti-malware solutions. The motives of malware writers [5] are also changing; initially malware were written for the sake of fun, thrill and fame but presently the focus has shifted towards making profit (Caballero et al., 2011), supporting terrorism (Langner, 2011) (example: Stuxnet) and cyber espionage (Zhioua, 2013) (example: flame, duqu and dino).

1.2 Malware Detection

The exponential growth and sophistication of malware poses a critical challenge to the digital world. To control and minimize the loss caused by malware, many security solutions i.e. Anti-Virus (AV) techniques [6] have been developed and new approaches have been researched. These AV techniques are broadly classified as Signature-based and Non-Signature-based techniques. Signature-based AV software uses scanning technique. It scans the suspicious files for signature (specific sequence of bytes). This approach is very fast and gives nearly 100% accuracy for known malware but totally fails to detect the "zero-day[7]" (Bilge and Dumitras, 2012) and "unknown[8]" malwares (Hodgson, 2005; Murugan and Kuppusamy, 2011; Kumar and Pant, 2009). Signature-based techniques are limited by its signature databases and moreover signature creation itself is a time taking and complex process which could provide a larger *attack time window*[9] for the attacker.

Using *known* tools and techniques will dramatically decrease the operation's likelihood of success for the attacker (Potter and Day, 2009). So attacker must create a new tool for committing a successful cyber-crime and hence different anti-malware evading

rise-of-cross-platform-malware,(Accessed: 2017-04-30)

[5]The person who writes malware. The malware writer and the attacker can be the same or the different individual.

[6]Anti-virus techniques and malware detection techniques represent the same. Anti-Virus is a security product hence Anti-virus techniques are mostly used in business environment while malware detection techniques used in research.

[7]Malware which is recently released by attacker in wild and have not detected by any AV.

[8]Those malware for which signature is not available. Example: targeted malware

[9]Time between release of a malware in wild and updating anti-malware solution (collect sample, create and update signature, push update to AV and end-users).

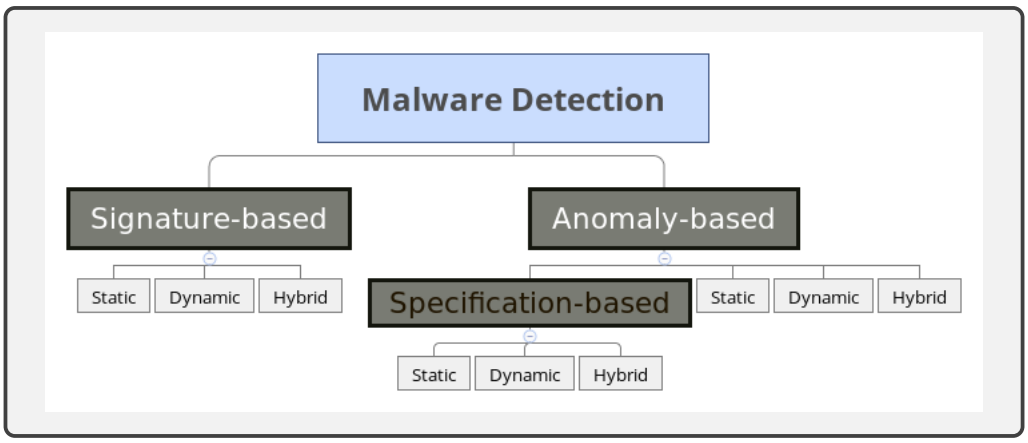


Figure 1.2: Types of Malware Detection Techniques

techniques (Christodorescu and Jha, 2004) are being developed. Anti-malware process involves many tasks amongst which detection is the most important and complex one. Many malware detection techniques have been discussed in literature and many of them are being used by the anti-malware industry.

In past, researchers have grouped malware detection techniques in different ways, *Signature-based* and *Anomaly-based* are two main classes of malware detection techniques (Idika and Mathur, 2007). Anomaly-based detection is further branched into specification-based detection. On the basis of underlying analysis techniques, all the aforementioned detection techniques are grouped into *static*, *dynamic* and *hybrid* techniques. Figure 1.2 depicts the classification of Malware detection techniques (Idika and Mathur, 2007). Static detection techniques are based on static analysis which does not execute the sample. Dynamic detection techniques use dynamic analysis which executes the sample in a controlled environment to extract out different behaviours of sample. Hybrid detection techniques use both static and dynamic analysis techniques.

1.2.1 Signature-based Detection

Signature-based detection uses the signature to detect malicious programs. Signature is a sequence of bytes extracted from previously known malware. Static signature-based detection does not run the program whereas dynamic signature-based detection executes the program in a safe environment and checks for the signature.

Signature-based detection gives higher accuracy for known malware but totally fails

when encountered with the "zero-day" and "unknown" malwares. The Signature-based detection is also limited to the signature database which demands a regular update of newly created signatures and storing the signature database at end-host requires space proportional to the number of signatures.

1.2.2 Anomaly-based Detection

Anomaly based detection which is a type of Non-Signature-based detection, is capable of overcoming the limitations of signature-based detection. The Anomaly-based detection does not use malware specific signature. For anomaly detection a normal profile is developed and any diversion from normal profile is treated as malicious. This work focuses on static analysis based malware detection using machine learning algorithms which is a type of anomaly-based detection.

Basically, features are the core of the Machine Learning (ML)-based malware detector (Yan et al., 2013b). This work aims to devise and create new feature sets which can improve the malware detection performance. In this research work, four feature sets are created to detect malicious Portable Executable (PE) and Android applications using the machine learning algorithms. This work has used static analysis method which is simple and fast for feature extraction and also yields effective performance. The research problem statement and the related specific research questions which are answered by this thesis is presented in the following section 1.3.

1.3 The Problem Statement

Portable executable and *apk* are two major file format used by Windows and Android OS respectively, hence most of malware in wild are in these two file formats. Traditional signature-based malware detection techniques are unable to detect "zero-day" and "unknown" malware, which necessitate to explore alternative non-signature-based detection techniques. Machine Learning based methods serve an alternative solution which can overcome the limitations of traditional system with the use of static or dynamic features. The exponential growth in number of malware and its increasing sophistication makes malware detection time consuming. In many cases malware escape the detection which is the major concern for Cyber space and the users. The research

problem focused by this thesis can be stated as follows:

> *To build a framework for malware detection by exploring the potential of static features with feature engineering techniques to enrich the discriminative capability of features to achieve better malware classification results with machine learning algorithms.*

This thesis explores the above stated research problem with the help of following research questions:

Question 1: How the performance of PE headers-based features can be improved?

As stated earlier, the PE is main file format used by Windows OS. Many static features can be extracted from PE to build ML based malware detector. Most of the existing works have used fields value as feature value without any processing. This thesis has proposed an integrated feature set having combination of *derived* and *raw* features based on various fields of PE headers. All these features are extracted with static method but values for derived features are obtained after pre-processing the field's value. Various experiments shows that it improve the performance of ML based malware detector in comparison to existing feature set.

Question 2: What is the potential of *section name* as features to build ML based malware detector?

Each PE file is organized in various sections and each section has a *name* with the respective header. In traditional signature-based system few *section names* are used as the part of signature and many ML based detector uses just *total section* as feature. This research has proposed a feature set based on just *section name* and explores the potential of it with various ML algorithms.

Question 3: How weighting the permission of Android applications will affect the detection performance of ML algorithms?

As stated in the problem statement above Android applications are packaged only as *apk* format. Each *apk* has many information which are used as static features to train ML algorithms. Permission present in the *manifest* file inside the apk is one among them. Earlier works have considered permission as boolean features while this thesis assign a weight to each permission to create *numerical* feature set using

permissions. With various experiments conducted, the performance improvement over its boolean counterpart has been studied.

Question 4: Do the image representation of Android application can improve the malicious android applications detection?

Using *apk* structure to extract features yields only values supported by the format. Treating *apk* as binary file and converting it into image can make it possible to use features available for image classification also. This research uses *apk* to image conversion and explores the effect of image features for malicious Android application detection.

Question 5: How the various feature selection and machine learning algorithms impact the performance of the features?

Various features have unequal discriminative capability, some are very significant in deciding the class label while others are least. This research explores the feature selection methods to choose best performing features among various features extracted from PE headers and Android applications. The working principle of machine learning algorithm varies one to another. So, it is possible that an algorithm can perform very well with a feature set but their performance may get degraded with other feature sets. This research experiments with various ML algorithms on the proposed feature sets which help to decide and choose the best ML algorithm for malware detection.

In order to find answers for all the above listed research questions, this study has conceptualized a layered research framework, under which various experiments are carried out and the steps of machine learning such as pre-processing, feature extraction, feature selection and training & testing are done. The framework and its layers are explained in the following section 1.4.

1.4 The Proposed Research Framework

To explore the problem domain and seek answers for the proposed research questions, a six layered research framework has been conceptualized, which helped in carrying the research work systematically. Figure 1.3 shows the all six layers of the proposed

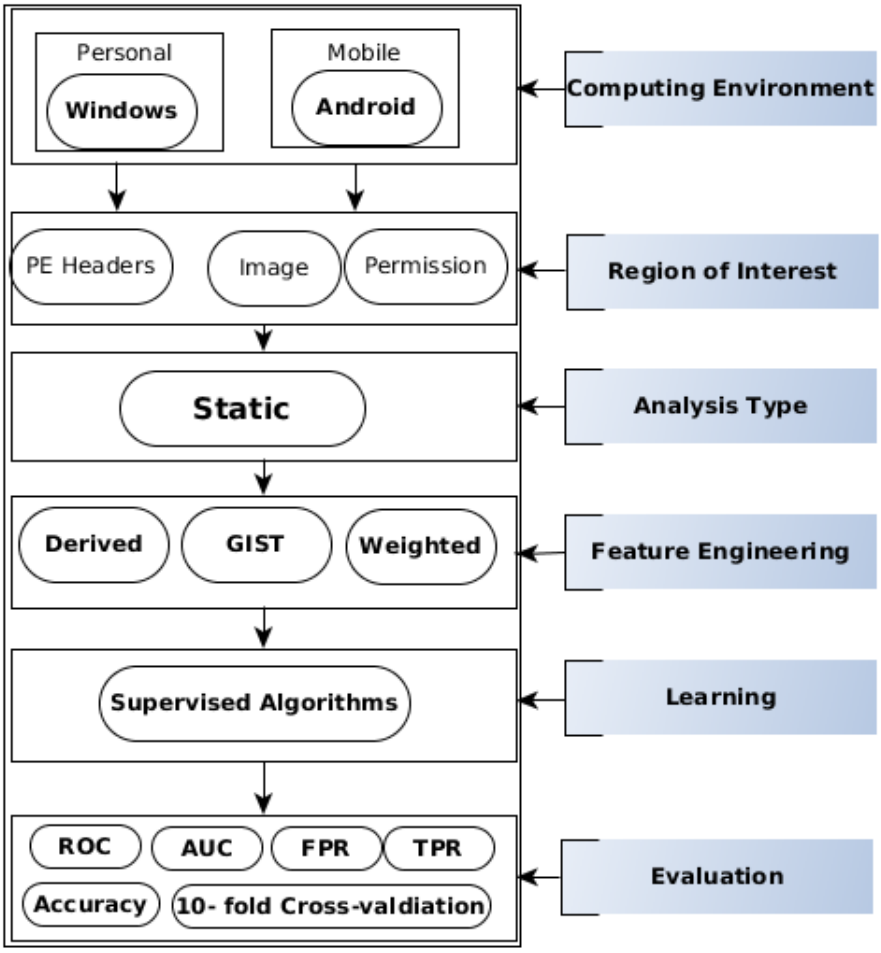


Figure 1.3: The Research framework

research framework and in subsequent sections a brief description of each layer with it components is presented.

1.4.1 Research Objectives

The overall objective of this research work is to enrich the discriminative capacity of existing static features and devise new features for malware detection using machine learning algorithms. Below the specific objectives are listed.

1. To build an integrated feature set from various fields of portable executable headers and compare it with the raw feature set for malware classification.

2. To construct a boolean feature set with section names of the portable executable file and evaluate its performance for malware classification.

3. To extract and build a feature set from the image representation of apps for Android malware classification.

4. To construct a permission based feature set using the weighted score and comparing it with permission based boolean feature set for Android malware classification.

1.4.2 The Components of Research Framework

As stated earlier, to carry out the research work, a six layered research framework has been conceptualized. This section presents a brief summary of each layer and its components.

1.4.2.1 Computing Environments

Computing environment is the term used to describe a complete computing platform comprising of hardware, OS, and other software. In recent time, the exponential growth in Cyber attacks in general and malware attacks particularly forced each computing environment to have Anti-virus softwares. This research work aims to improve the detection of such AV softwares. This work has used PE files which is the file format used by Windows OS known for personal computing and used *apk* which is the application packaging system for Android OS known for mobile computing. The selection of these two were made on the basis of number of users. These two have highest users in respective OS types [10]. In Figure 1.3 the top layer is the computing environment which is the first thing to decide to move further.

1.4.2.2 Region of Interest (Source of Features)

The PE and *apk* have many areas (header, meta-data and code) which can be used to extract various features to build malware classifiers. This study has focused on PE headers for *PE files* and *image format* & *manifest* files for Android applications. This section gives a brief introduction about these feature sources. The detailed description of each source and feature set is presented in forthcoming Chapters 3 to 6. In Figure 1.3,

[10] `https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0`

the second layer is the *Region of Interest* which lists the sources used in this work to extract features to build the feature set.

1.4.2.2.1 PE Headers: Every PE file has two main headers namely *DOS header* and *NT header* having various fields to hold values used for loading and executing the PE files. PE files are organized in section and each section has a header which has fields holding information about the section. In existing works, these field values are used as feature without any pre-processing (Schultz et al., 2001b; Awan and Saqib, 2016). This work has proposed an integrated feature set based on these fields and rules applicable on these fields. This work also considers PE sections and used *section name* as features to build a boolean feature for malware detection.

1.4.2.2.2 Image: With advancements in domain of Digital image processing, many image features have been identified to use with machine learning algorithms. To harness the benefits of this mature field, this work has converted *apk* into *image* and used *GIST* feature to train and test various ML algorithms for building malware detector based on image features.

1.4.2.2.3 Manifest: Each Android application has a *manifest* file which along with other metadata and have a list of permissions required by the application to perform it functionalities on user device. The work has used this manifest file and extracted permissions to build a weighted permission based feature set.

1.4.2.3 Analysis Types

Feature extraction is dependent on analysis type and so features differs on the basis of analysis type. Sample analysis and feature extraction can be done in two ways, static and dynamic. In Figure 1.3, the third layer is the *analysis* layer which mentions the selected static analysis method.

1.4.2.3.1 Dynamic analysis: In dynamic analysis, the sample is executed in a controlled environment and the behaviours are recorded to create features. It is complex, time consuming and requires intensive processing. It is suitable for sample which are packed and for which static analysis is not possible. Use of the *virtual machine* makes

it possible to run different OS and take *snapshot* [11] of its state which is very helpful for execution of malware during dynamic analysis.

1.4.2.3.2 Static analysis: In static analysis, the sample is not executed and only the structure is analyzed by going through the hexadecimal or assembly representation of the sample. Static analysis is done using reverse engineering of the sample, various disassembler and hex editors which converts the sample into assembly equivalent or shows in hexadecimal equivalent of its binary values. This thesis work has chosen the static analysis for all the feature extraction tasks, which is simple, fast and require less processing than dynamic analysis.

1.4.2.4 Feature Engineering

It is the process of using domain knowledge of the data to create features that make machine learning algorithms work. It is very fundamental to the application of machine learning and is complex and expensive[12]. This thesis work is focused on the domain of malware detection. Complex task of feature engineering is very challenging with malware samples, since it may infect the experimental computing environment and can misguide the analysis and extraction process. This work has created and experimented with various features and feature set in order to improve the malware detection using machine learning algorithms. In Figure 1.3, the fourth layer is the *feature engineering* layer which is core of this research work.

1.4.2.4.1 Derived Feature: As stated earlier, earlier works have used field's value as feature without any processing and so those are considered and termed as *raw features* in this work. Instead of using *raw values* from PE headers, in this work *raw values* are compared with the *field rule* and the resulting output are considered as feature value.

1.4.2.4.2 GIST: There are various image descriptors which are being used in digital image processing for various classification and detection tasks. In this work, *GIST* descriptors of each image representation of malware and benign apk are extracted and used to train machine learning algorithms.

[11] It records the state of operating system such as installed softwares, files status etc.

[12] `https://en.wikipedia.org/wiki/Feature_engineering`

1.4.2.4.3 Weighted Permission: Many of existing works have used permission as boolean feature to use with machine learning algorithms (Sanz et al., 2012; Idrees et al., 2017). In this work, each permission has been assigned a weight using their frequency in malware and benign sample. The resulting weight were considered as feature value instead of traditional boolean value to build feature set.

1.4.2.5 Learning Types

Learning is the process of passing feature set to machine learning algorithm where the relation between features are learnt and the learned relation is use to classify new sample. Learning is mainly of two types: supervised and unsupervised. In supervised learning each row of the feature set has a respective class label whereas in unsupervised learning feature sets are not labeled. Supervised learning mostly used for classification task which matches the malware detection problems.

1.4.2.5.1 Machine Learning Algorithms Machine learning algorithms are mainly of two types: 1) Supervised 2) Unsupervised. Supervised algorithms require *labeled feature set* while unsupervised algorithms work *without labeled feature set*. Supervised algorithms are used for classification tasks whereas unsupervised algorithms are used for clustering task. In this thesis work eight supervised machine learning algorithms (Logistic Regression (LR), Decision Tree (DT), k-Nearest Neighbors (kNN), Linear Discriminant Analysis (LDA), Gaussian Naive Bayes (NB), Support Vector Machine (SVM),Random Forest (RF), and AdaBoost) are selected which are used in various experiments. LR is a type of regression that is used to estimate the probability that the dependent variable will have on a given value, instead of estimating the value of the variable. The DT is a tree based supervised learning technique which recursively partitions the features space to model the relationship between features and the target categorical variable. The KNN is distance based algorithm and the working principle of *nearest neighbor* method is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. LDA having a linear decision surface. It has closed-form solutions that can be easily computed and are inherently multi-class. NB method works on applying Bayes' theorem which assume independence between every pair of features. SVM works by classifying data through finding the line which

separates data into classes. It tries to maximize the distance between the various classes and referred as *margin maximization*. The RF uses ensemble methods on a number of decision tree classifiers on various subsamples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. AdaBoost (Freund and Schapire, 1995) is an ensemble learner. It works by trying to fit a sequence of *weak learners* [13] on repeatedly modified versions of the data.

This thesis work has used *Scikit-learn* implementation of all the selected algorithms. The *Scikit-learn* is a Python framework which has the implementation of many popular machine learning algorithms. In this work, most of the algorithms are used with default parameters and it has been indicated whenever any changes are made.

1.4.2.6 Evaluation

Evaluation is carried to make conclusions about the performance of feature set and the respective machine learning algorithms. It helps in deciding the features or feature set that performs better or how they are performing with different machine learning algorithms. This thesis work has used standard evaluation methods to evaluate the proposed feature set with various machine learning algorithms. This is the sixth step of the proposed research framework and is depicted as last layer in Figure 1.3.

1.4.2.6.1 Cross-Validation: It is a technique to evaluate trained machine learning algorithm by partitioning the original sample into a training and testing set. It works with folds. For example in a k-fold validation, the sample is randomly partitioned into k equal size sub-sample and $k-1$ sets are used for training and the one left out set is used to evaluate. The final result is average of k repeat, it helps in reducing the biasness and over-fitting or under fitting. In this thesis, 10 fold cross-validation testing method is used along with other traditional evaluation methods such as *train-test split* and *with news test dataset*.

1.4.2.6.2 Performance Metrics The *performance metrics* are the systematic measures to evaluate the effectiveness of a system. In machine learning domain various performance metrics are used to evaluate the effectiveness of the feature set, feature selection algorithms, and machine learning algorithms. In this thesis work various

[13] Those models that are only slightly better than random guessings, such as small decision trees.

performance metrics are used to evaluate the discriminative capacity of the proposed feature sets in comparison to its counterpart other feature sets. A set of performance metrics is used to evaluate classification performance of the feature set along with various machine learning algorithms. Some of these metrics are very primitive such as True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). These primitive metrics are used to calculate various advanced metrics such as, *Precision*, *Recall*, *Accuracy*, *F1-score* and *AUC*.

The four advanced metrics accuracy, precision, recall, and F1-score have listed in Eq. 1.1 to Eq. 1.4 which are used in this thesis work to evaluate the performance of all the four proposed feature sets (respective section of all four feature sets are coded as I to IV for avoiding repeated section heading.). Along with these metrics, there is two more measure to evaluate the performance Receiver Operating Characteristic (ROC) and Area Under Curve (AUC). The ROC curve is created by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. The AUC is total area under the plotted ROC curve. The result from AUC can be interpreted as "greater the area better the performance".

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{1.1}$$

$$Precision = TP/TP+FP \tag{1.2}$$

$$Recall = TP/TP+FN \tag{1.3}$$

$$F1-score = 2 \times \frac{precision+recall}{precision \times recall} \tag{1.4}$$

1.5 Organization of the Thesis

- Chapter II presents a survey of Malware detection techniques including earlier works which have used Machine Learning algorithms and with static features.
- Chapter III introduces derived features which are the outcome of field's values of PE headers and associated rules for the fields. Combining derived features with

raw features as integrated feature set improved the classification performance (accuracy, TFP etc.) of malware classifiers.

- Chapter IV explains and presents the potential of *section name* as binary feature to classify sample into malware and benign class. It requires very low processing for extraction and feature set preparation and provides adequate classification performance.

- Chapter V presents the weighted permission as the feature and compares performance with traditional binary permission feature. The permission weighting was carried out by using a scoring engine which utilizes permissions frequency as metric to calculate various scores. The best performing classifier was used to build a tool for classifying installed apps of the live Android device.

- Chapter VI explore the potential of features extracted from image representation of Android apps. Each app was converted to image based on four different color channels (Grayscale, Red, Green, Blue (RGB), Cyan, Magenta, Yellow, Key (blacK) (CYMK),and Hue, Saturation, Value (HSV)) and GIST features were used to train and test Machine Learning algorithms.

- Chapter VII discusses the conclusions and point out the future scopes and further improvements.

CHAPTER 2

Related works

This chapter presents the existing works related to malware detection using Machine Learning algorithms with static features. Works related to detection of both malicious Portable Executable (PE) and Android applications are organized and presented in respective sections. This study helps to understand the limitations and scope of improvements in existing approaches and techniques. The entire study focuses on the existing works which have used static features with machine learning to detect malware. In the set of static features, works which have used Portable Executable (PE) header field as features as well as Android application's permissions as features or any other static features to classify Android and PE applications have been explored.

Section 2.1 lists and explains various static features which are used with machine learning algorithms. On the basis of analysis type, these static features are classified into *static*, *dynamic* and *hybrid*. Section 2.2 groups and explains features based on Portable Executable (PE) file whereas *apk-based* static features are explain in section 2.3. Section 2.4 critically analyze the existing related works and discuss them in details.

2.1 Features used with ML Algorithms for Malware Detection

Feature extraction is the process of extracting or calculating value from each sample of dataset. Feature extraction is domain dependent; for instance, Information Retrieval (IR) domain feature extraction is straight and simple whereas for recommendation system features set are structured and organized.

For malware research, dataset is normally the programs written in different forms

i.e. Dynamic Link Library (DLL), Executable (EXE), Component Object Model (COM) and code snippets. Features extracted from these binary program files either through static analysis or through dynamic analysis. Features extracted by static analysis techniques are called as static features and features extracted by dynamic analysis are termed as dynamic features. Figure 2.1 shows different types of **static**, **dynamic** and **integrated** feature set used for malware detection.

Feature sets are classified into three groups: *simple heuristic* feature (features extracted from PE header or strings extracted from executable body), *static* features (features extracted through static analysis) and *dynamic* features (features extracted at runtime)(Dai et al., 2009). One of the studies considered detection techniques as structural fingerprints which are statistical in nature and positioned it as 'fuzzier' between static and dynamic heuristics(Bilar, 2007). Komashinskiy et al. have presented classification of features in detail; they have divided whole features as external and internal. Later the internal features are grouped as static and dynamic, under these two groups generic feature is also listed such as string, n-gram etc.(Komashinskiy and Kotenko, 2012).

Yen et al. have presented the exploration of discriminatory features for ML-based malware classifier. They have experimented with various feature sets (byte-n-gram, opcode-n-gram, fields of PE header and dynamic trace) for malware families classification. They presented their result to show which algorithm works best with which feature set and how many features are optimal. With their study, they have suggested that Decision Tree (DT) performance is highest among all or equal to the performance of Support Vector Machine (SVM) for all features and provides highest accuracy with minimum features (Yan et al., 2013a).

In the following sections, static (section 2.1.1), dynamic (section 2.1.2) and hybrid (section 2.1.3) features are listed and explained with relevant existing works.

2.1.1 Static Features

As discussed earlier, on the basis of analysis type, features are mainly grouped into three classes i.e. static, dynamic and hybrid. In this section different class of static features are listed and explained with the reference of relevant works. Static features are extracted without running the executable file. Reverse engineering is a common approach to extract static information from a binary. Disassembly and hexadecimal

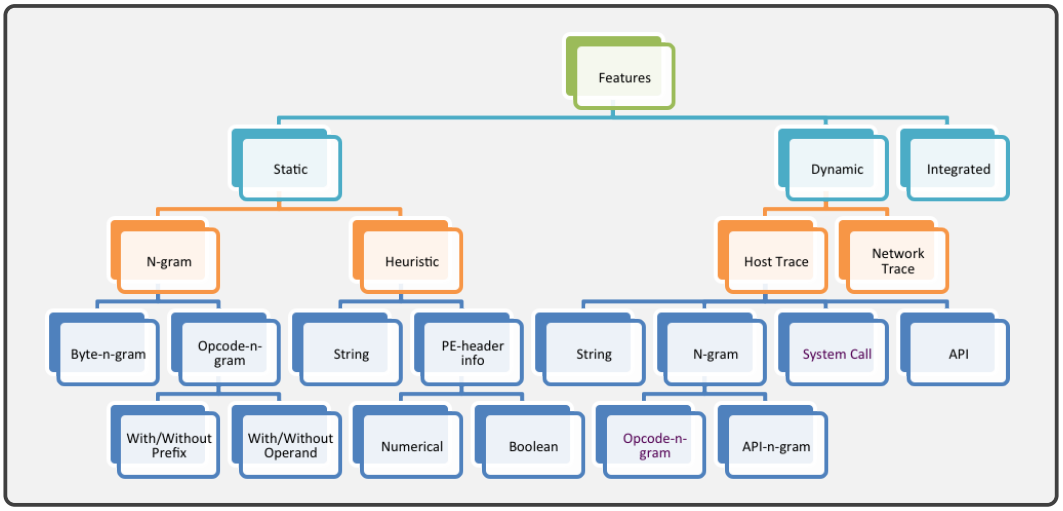


Figure 2.1: Types of features used for building machine learning based malware classifier

dumping of binary file are the two main techniques to pre-process and get static features from the sample.

The Static features can be of two types: *n-gram-based* and *heuristic-based*. N-gram is a concept taken from Information Retrieval (IR), which refers to the *N* consecutive character with a shift of size *n* from a text document. N-gram for malware can be *byte-n-gram* (n-consecutive byte sequences taken from a *hexdump*[1] output) or *opcode-based* (n opcode sequence taken together from a disassembled binary). *Ida-pro* and *Objdump* are two common tools used for the disassembly.

A taxonomy for classifying detection methods of malware by machine learning methods based on static features extracted from executable is presented by (Shabtai et al., 2009b). They have listed and explained five static features: Byte-n-grams, OpCode-n-grams, Function-based, PE feature and Strings. In the following sections each of these static features along with other static features are explained.

2.1.1.1 Byte-n-grams

Byte-n-grams are the byte sequence of length *n* that occurs in the target (Schultz et al., 2001a; Nissim et al., 2014; Shahzad et al., 2010; Shabtai et al., 2012; Moskovitch et al., 2008b; Stibor, 2010; Santos et al., 2011c; Abou-Assaleh et al., 2004a,b; Elovici et al.,

[1]Hexdump is a tool for converting binary file to its equivalent hexadecimal representation.

2007). Theoretically, it represents program's structural components, fragments of instructions and data (Dube et al., 2013).

Commonly, *Byte-n-grams* are used as boolean or numeric feature set to train and test with different machine learning algorithms. A generic malware detection classifier has fabricated by determining the probability of finding specific n-grams in malicious and non-malicious programs (Kephart and Arnold, 1994).

2.1.1.2 Opcode-n-gram

Opcode-n-gram is a sequence of *n* consecutive opcode, each sequence is created with a sliding window of *n* over the disassembled code of the sample. *Opcode-n-gram* with and without operand features has also been used as features for building malware classifier (Shahzad et al., 2011; Shahzad and Lavesson, 2012; O'Kane et al., 2013; Shabtai et al., 2012; Karim et al., 2005; Santos et al., 2011a,b, 2010, 2013; Lakhotia et al., 2013). A study carried out the statistical tests to verify the deviation of opcode frequency in malware and goodware (benign files)(Bilar, 2007). The study concluded that the malware opcode frequency distribution seems to deviate significantly from non-malicious software and rarer opcodes have more frequency variation. In the study it was also found that malware do have undocumented opcode such as *salc* and *icebp*.

Opcodes were used in a unique way by grouping those into three opcode features:1)branch opcodes, 2) unigrams and 3) bigrams (Raphel and Vinod, 2016). Opcode with and without first operand were used as the feature set to train and test the Decision Tree (DT) and Gaussian Naive Bayes (NB) classifiers (Wang et al., 2003). Authors concluded that the classifier perform well when each feature contains more information.

Opcode-n-gram can be extracted with both static and dynamic feature extraction methods. Most of works based on opcode-n-gram have considered only opcode sequence as features and shown a very good accuracy. But opcode-n-gram with operand can improve the accuracy further and reduces the False Positive Rate (FPR).

2.1.1.3 Strings

Every computer program either compiled or raw has many information in ASCII characters, for example, hard coded Uniform Resource Locator (URL), messages etc. Grouping these characters form strings and by limiting the length of strings, one can have

many meaningful strings.

These *strings* which are present in executable format are used as features in many existing works (Schultz et al., 2001a; Shabtai et al., 2009a; Rieck et al., 2008). *Strings* based feature set are normally of boolean type where presence and absence of the string is represented by '1' and '0' respectively. *Strings* from the sample can be extracted by static analysis (Schultz et al., 2001a; Shabtai et al., 2009a) or by the dynamic analysis (strings from runtime traces) (Rieck et al., 2008).

2.1.1.4 PE headers' fields

PE file format is structured through different headers and each header having different fields. Previously, these fields are used as feature and their fields' value considered as feature's value (Dube et al., 2012; Shafiq et al., 2009a; Sheen et al., 2013).

PE headers' fields of PE file are very good heuristic indicator of discriminative features for classifying malware and benign programs. PE header features are very important features to build automated malware classifier since extracting fields' value is computationally cheaper and these features possess high discriminative power (Yan et al., 2013a). This thesis work has proposed an *integrated feature set* based on the PE header values and has achieved an improved performance than existing works by introducing derived features which is the outcome of field's *value* and the *rule*.

2.1.1.5 Permissions and Intents

The *permissions system* is one of the security feature which ensure the authorized execution of Android applications. Most of the *Desktop OS* have *user-level* or *application-level* permissions. Android permissions are fine grained to action level i.e each separate action require user permission such as *Camera* permission is different than *Internet* permission for same application. The permissions and intents information are declared and stored in *manifest* file of each Android application. Many of earlier works have used permissions and Intents as feature to build malware classifiers or any other ML based classifiers (Di Cerbo et al., 2010; Geneiatakis et al., 2015; Sanz et al., 2012).

2.1.1.6 Image Feature

Image features are known as image descriptors which represents an image in an abstract form. Recently, the idea of visualizing the application and then using image feature to classify applications is getting attention from many researchers (Kancherla and Mukkamala, 2013). For example, *Binary files* from different malware families were converted into grey scale image and the *texture of the image* were used as feature to build malware families classifier (Nataraj et al., 2011).

2.1.1.7 Miscellaneous

Apart from the earlier listed static features, there are works which have used static analysis method to create features other than the commonly used features. In MIST, the program behaviors are represented using a three level structure (level 1, 2 and 3) and performed a similarity analysis of five malware classes based on level 1 and level 2 representation of the program (Trinius et al., 2009). Authors have also shown that MIST-based features representation are smaller in size than other format such as Extended Markup Language (XML).

This section grouped and explained the classes of static features used in the malware literature and in the following section the dynamic features are listed and explained.

2.1.2 Dynamic Features

Dynamic features are those features which can only be extracted through dynamic analysis. Dynamic features such as *function call*, *network use*, *system call* and *Hardware Performance Counter* (Wang et al., 2016) can be recorded after executing the sample in controlled environment. Dynamic approach overcomes some of the limitations of static analysis. With static analysis packed, polymorphic (Aljawarneh et al., 2016), metamorphic (Raphel and Vinod, 2016) and encrypted or obfuscated sample can escape the analysis or make it difficult while dynamic analysis handle this because it works by executing malware's payload and recording sample behaviors during run time.

Dynamic process also has limitations such as speed, native run, cleaning experimental system and system overhead. A survey work has presented dynamic malware analysis techniques and provided a detailed discussion on such analysis tools (Egele

et al., 2012). Dynamic features can be grouped into two major groups: *host trace* and *network track-based* features.

2.1.2.1 Host Trace

The activities of internal memory, files and file system, registry, hardware performance counters and status of running processes of host are considered as host trace and used as features. Strings (Yan et al., 2013a; Moskovitch et al., 2008a), n-grams (O'Kane et al., 2013; Ahmed et al., 2009; Salehi et al., 2012; Santos et al., 2013; Sundarkumar and Ravi, 2013; Islam et al., 2012), system calls (Yavvari et al., 2012; Gurrutxaga et al., 2008) and API calls (Ahmed et al., 2009; Salehi et al., 2012; Santos et al., 2013; Sundarkumar and Ravi, 2013; Islam et al., 2012) which have been extracted by dynamic analysis are host based-trace. For training and testing machine learning algorithms, these features are used in a similar way as the static features. Dynamic analysis method produce varying features and values which impact the performance of ML algorithms.

2.1.2.2 Network Trace

Network trace-based features are log of the network activities (Bayer et al., 2009; Zhang et al., 2016). It can be captured at host machine by network sniffing tools [2] or at network entry point by the use of proxy system. Various implementation of *Pcap* [3] is used for network packet capture and these network traces are mainly used for intrusion detection system but can be use for malware detection.

Among aforementioned dynamic features, Hardware Performance Counter (HPC)-based features are recent and seems to be promising for malware detection (Wang et al., 2016). API and opcode are very interesting features because these two can be extracted by both static and dynamic feature extraction.

2.1.3 Hybrid Features

Hybrid feature is a combination of both static and dynamic features. For creating hybrid feature set, features are extracted by using both static and dynamic analysis. An

[2]Tool which capture the network packet during transmission, such as wireshark etc.

[3]pcap(packet capture) consists of an (API) for capturing network traffic. Different OS have different implementation of *pcap*.

integrated features were created by mixing both static and dynamic features. Experimentally, it is found that integrated feature set yields higher accuracy than individual feature set (Santos et al., 2013; Islam et al., 2012). A recent study has proposed the integration of static and dynamic features for malware classifier (Awan and Saqib, 2016). In the study static features *PE header* information and *printable strings* were used along with dynamic features API call logs (Awan and Saqib, 2016).

Comparing different features is a complex task. Based on the requirement, the selection of features can be done on the basis of various metrics. Feature extraction time, accuracy and FPR are the crucial metrics to make a choice to select a feature set. PE header features extraction is fast and require very less space in comparison to byte-n-gram and opcode-n-gram.

This section has presented and explained about various features which are used with machine learning algorithms for building malware detector. This thesis work involves static analysis and use Portable Executable (PE) file and *apk* file for building static features & feature set for malware detection. So, in the following sections, features specific to Portable Executable (PE) file and Android's *apk* file are listed and discussed with references to related earlier works.

2.2 Static Features from Portable Executable Headers

PE is a file format used in Windows OS to construct, organized and execute different file types such as executable (.exe), (Dynamic Library (.dll) etc. Similar to *PE*, *Executable and Linkable Format (ELF)* and *Mach-o* are file format used in *Linux* and *Mac OS X* respectively. The PE format is implemented as a data structure to encapsulate the information necessary for the loader to manage and use various Windows files. Every file in PE format consists of two main headers (*DOS header* and *NT headers* explained further in Section 3.1.1 and Section 3.1.2) placed at beginning and further followed by some *section header* (one for each available section). The Microsoft Corporation's specification document provides the description of the structure of PE files (Visual and Unit, 1999). These fields of PE headers have been used in various ways to make it more discriminative for classification of malicious and benign samples. PE features are divided under six categories: data extracted from PE headers, Optional PE header in-

formation, Import section, Export Section, Resource directory and Version information (Shabtai et al., 2009b). This section presents a category-based discussion by grouping previous works on the basis of features types as *headers fields' value*, *Dynamic Link Library (DLL)*, *Application Programming Interface (API) calls*, *data directories*, *section headers* and *section name*.

2.2.1 Headers Fields' value

File header and *Optional header* are specific to PE format, so the fields values from these two headers should come under PE headers features. Previously, PE headers fields are used as features and value from these fields were directly used as feature's value. For example, one of the study has used PE Header Entries as feature set and achieved 96.9% True Positive Rate (TPR) and 0.0984 False Positive Rate (FPR) with Random forest classifier (Vinod et al., 2011). Similarly, an another study extracted a total of 38 features from file header and Optional header and used these features with other PE features (Shafiq et al., 2009b). In one of the work, PE header entries were extracted and the SVM classifier was trained using selected features. The trained SVM model detects viruses and worms with considerable accuracy but the detection accuracy were lower for Trojans and backdoors (Wang et al., 2009). A hand crafted rule based algorithm was implemented for classification using five selected PE header fields as features (Liao, 2012). One study considered all the fields of various headers as integer features and a feature set was created having a total of 68 features (Baldangombo et al., 2013). An another work used all fields of PE headers except characteristic and image resource *NameID* fields as the numerical feature while each bit of characteristic was used as boolean features (Yan et al., 2013b). A recent study has used 5 fields of *file header* and 16 fields of *optional header* and only the e_lfanew field of *DOS header* as integer features (Bai et al., 2014). In one of the study, only fields from the PE-optional header are used as features (Belaoued and Mazouzi, 2015). In an earlier work, a feature set with 1867 features only has used 15 PE headers fields as numeric features (Walenstein et al., 2010). A recent work created a feature set having 44 features which were extracted from different PE headers and having 2 features based on *section entropy* (Markel and Bilzor, 2014; Markel, 2015). A recent study created a feature set by integrating static and dynamic features so the *PE header* information were used as static features and only

30 most relevant PE header features were selected by *information gain* method (Awan and Saqib, 2016).

From the aforementioned discussion, it can be easily concluded that PE headers fields are very effective in malicious program detection and have been used previously either as a complete feature set or mixed with other features. Most of the works have used raw value of fields for features but this thesis work has pre-processed these value with the rule associated with each field and used them by calling derived features. These derived features are then mixed with selected raw features and used as proposed Integrated feature set.

2.2.2 Dynamic Link Library (DLL)

As per the discussion, PE headers are implemented as data structures which have information in linked format. The last field of *Optional header* is *DATA_DIRECTORY* which is an array of data structure having provision for 16 data directory definition. Each data directory is specific to a different task. Information about every DLL which would be used by PE file and need to import during linking and execution is stored in *DIRECTORY_ENTRY_IMPORT*. By reading and parsing a PE file, one can easily list out all required DLL. DLL precisely map the functionality of any file at an abstract level. Using every DLL as a feature would result in more accurate malware classifier. DLL information can be utilized in the various ways to create a feature set.

In a previous study to carried out different experiments three feature set were created using DLL information (Schultz et al., 2001b). Those feature sets are, *DLL name* as boolen features, *DLL & function calls (API)* as boolean features and *number of different function calls* within each DLL as integer features (Schultz et al., 2001b). The performance of only DLL based feature sets are moderate for example *detection rate* for *RIPPER* classifiers were 57.89%, 71.05% and 52.63% respectively. In another work 73 DLL names were used as boolean features with other PE features (Shafiq et al., 2009b). A similar study has used 792 DLLs name as boolean features along with other PE features (Baldangombo et al., 2013). In a different work 30 selected DLLs names were used as boolean feature along with the *number of DLLs referred* by each PE file as an integer feature (Bai et al., 2014). In one of the study, GNU Binutils *objdump* program was used to extract 1852 imported DLLs name and used as boolean features (Walenstein

et al., 2010). A work on malicious PE file detection has used a total of 4167 boolean features among other features DLLs name based boolean features were also used (Yan et al., 2013b).

As one can observe that features based on DLL information have used extensively in the literature and have limited scope of improvement. Hence, the proposed framework has not included DLL features in the Integrated feature set which helps to observe behaviors of other features more clearly. Similar to DLL earlier works have also used API information as features which is explained in the following section.

2.2.3 Application Programming Interface (API) calls

API is a set of methods and procedures which allow accessing the features of data of an OS, application or another service. In an operating system context, API calls which are related to a specific task are grouped together and called as DLL or library. After accessing the DLL information from import data directory, the necessary API calls specific to the PE file can also be extracted. API is one level closer to OS system call than DLL, hence the list of all API calls by a PE file can be very discriminative to identify the real intentions of a PE file. API calls information can also be used to create a feature set and so many of previous works have used this information as features.

An earlier study has used API information and created two feature set for the experiment (Schultz et al., 2001b). One was the boolean feature set having DLL & function calls as the feature and other was integer feature set having the DLLs with counted function calls as the feature. As mentioned earlier, only 71.05% and 52.63% detection rate with RIPPER classifier was achieved in the study. In another work, API call sequence of PE file (which were extracted using Import Address Table (IAT) field) were used as features and with Objective Oriented Association (OOA) mining method, an accuracy of 93.7% was achieved (Ye et al., 2008). A study has used API calls of DLL as features and by using Information Gain reduced the total number of features to 11 in the final feature set (Altaher et al., 2012). A similar work has used $24,662$ *API function call name* as boolean features along with other boolean DLL and PE features (Baldangombo et al., 2013). A recent work has used 30 selected APIs name as the boolean feature and also used the number of APIs referred by each PE file as an integer feature (Bai et al., 2014).

2.2.4 Data Directories

Optional header has an array of *structure field* holding information about 16 *DATA DIRECTORY*. Although, currently only 15 data directories definition are in use. Each of these data directories gives the address and size of a table or string that OS uses. Some of the important directory table names are, *export*, *import*, *resource*, *exception*, *certificate*, *debug*, *security*, etc. These directories are having lots of information about the PE file and so fields from the various directory can be used as features. In an earlier work, 30 features from data directory were used as features in conjunction with other features (Shafiq et al., 2009b). A recent work has used 16 integer features extracted from data directories of PE files as features (Baldangombo et al., 2013). In a similar work, 32 features were created from *data directory* and used as integer features (Bai et al., 2014).

The works related to data directories information as features are limited, although these data directories have lots of information which can be a good source for features. One of the reasons of such limited work can be the complex structure of directories which makes extraction process difficult.

2.2.5 Section Headers

Portable Executable (PE) file are organized and structure in various section on the basis of nature of information, for example *.CODE* section contains the program text of PE file. These sections based organization is optional and so it is not mandatory that each PE file should have a particular section or all the sections. Each available section of the PE file has a respective section header. The number of sections present in the PE file is given in *NumberOfSections* field in the *File header*. There are 10 fields in every section which holds information such as name, size, Characteristics, etc. of a particular section. The extracted value from all fields of available sections and other information about each section can be used as a feature.

An earlier study used 27 features from the header fields of three section (*.text*,*.data* and *.rsrc*) in conjunction with other features (Shafiq et al., 2009b). A recent work used 11 features from the header fields of each of the five sections (*.text*, *.data*,*.rsrc*, *.rdata* and *.reloc*) as integer features (Bai et al., 2014). From the aforementioned literature, it

is clearly observed that all fields of every sections are seldom used, but section related information is used for features.

2.2.6 Section Name

Sections name has classified as *suspicious* and *non-suspicious* on the basis of standard and non-standard name respectively (Perdisci et al., 2008). The total number of *suspicious* sections name was used as a feature to classify packed and non-packed PE files. A comparative analysis of various fields of malicious and benign files and analysis about *NumberOfSections* in malware and benign shows that nearly 50% of malware sample have less than or equal to 3 sections whereas only 25% of benign files fall in this range (Yonts, 2012). After structural analysis of PE headers for the optimization of malware detection techniques, it was observed and concluded that the presence of sections without names or name has other than alphabetic character is a feature of malicious files (David et al., 2016).

Motivated by the use of *section name* as features and the established fact of its discriminative capabilities mentioned in earlier works, this thesis has proposed and explored further the potential of *section name* as boolean feature with machine learning algorithms for building malware detector. Further, in Chapter 4, methods and other details about*section name* as boolean feature is explained.

Table 2.1 summarize and present a comparison of earlier works which have used PE features. From the table, it can be observed that continuous works have carried out for PE malware detection using machine learning. PE headers features are either used independently or with other features. Use of strings present in PE files is seldom. Combining various PE features as single feature set is also rare.

Table 2.1: Comparison of PE features used in earlier works

Previous Works	Type of Features								Performance	
	DLL	API	Sections	Headers	Strings	DD	Mixed	Others	Accuracy	FPR
(Schultz et al., 2001b)	✓	✓	✗	✓	✓	✗	✗	✗	97.11	3.80
(Ye et al., 2008)	✗	✗	✗	✗	✗	✓	✗	✗	93.00	NA
(Shafiq et al., 2009b)	✓	✗	✓	✓	✗	✓	✗	✗	99.00	0.50
(Wang et al., 2009)	✗	✗	✓	✓	✗	✓	✗	✗	98.86	NA
(Walenstein et al., 2010)	✓	✗	✗	✓	✗	✗	✗	✗	98.90	0.014
(Vinod et al., 2011)	✗	✗	✓	✓	✗	✗	✗	✗	96.80	0.13
(Altaher et al., 2012)	✗	✗	✗	✗	✗	✓	✗	✗	99.00	1.00
(Liao, 2012)	✗	✗	✗	✓	✗	✗	✗	✗	99.00	0.20
(Yan et al., 2013b)	✓	✓	✓	✓	✗	✓	✗	✗	NA	NA
(Baldangombo et al., 2013)	✓	✓	✗	✓	✗	✓	✓	✗	99.60	NA
(Bai et al., 2014)	✓	✓	✓	✓	✗	✓	✗	✓	99.01	1.40
(Belaoued and Mazouzi, 2015)	✗	✗	✗	✓	✗	✗	✗	✗	97.25	7.14
(Awan and Saqib, 2016)	✗	✗	✗	✓	✓	✗	✓	✓	90	10
(Ahmadi et al., 2016)	✗	✗	✗	✓	✗	✓	✗	✗	99.77	NA

In this section, various static features based on PE file were discussed along with the related works and relevance of these features with the thesis. In section 2.3, list of static features which can be extracted through the *apk* file is discussed. The features which are relevant to the thesis are also pointed out and explained in respective sections.

2.3 Static Features from Android's APK

Android OS is a popular smartphone OS and has the largest number of users with respect to the total number of the smartphone users. The application (apps) for Android is packaged as *apk* format which is very similar to the *compression* because all the required resources by the application is zipped as an *apk* file. So every *apk* file has *compiled code*, *metadata*, *configuration file* and *other resources* such as images, font etc.. With such vast information, *apk* becomes the primary source of features for creating machine learning based malware detector for Android application.

In literature, many works have used different components of *apk* to find out various features and building feature set for training and testing machine learning algorithms. By monitoring various dynamic features like *system calls*, *network activity*, *event log*, *user activity* and static features *permissions*, *intents*, *resource*, and *apps meta-data*, numerous categories in detection techniques are presented and explained (Suarez-Tangil et al., 2014b). In this section, various features based on earlier works have been presented and discussed along with notable similar works. Special note has been included for the features which are similar to the features used in this thesis, detailed discussion of such similar features are presented further in Chapter 5 and Chapter 6.

2.3.1 Used Feature

Every *apk* file is shipped with a *manifest* file which has information about *permissions*, *activities*, *intents* and *used features* required by the application. *Used feature* indicates the behavior of application by listing the features required and will be use by the application. For, this reason the *used feature* would be good features for building malware detector. Earlier works, like *PUMA*, (Sanz et al., 2013a) and MAMA (Sanz et al., 2013b) considered *used feature* by an app as feature and tested its discriminative power individually and with combination of permissions as features.

2.3.2 Meta-data

Android applications are distributed by various app-stores, *Google Play* [4] is official store for Android applications but there are many third-party app-stores which also distribute Android applications to the end users. Every store maintains lots of information like *user's review*, *file size*, *application version* etc. about every application and its version. Many of this meta-data like *review* and *rating* about application directly comes from the end users, so this tends to be very rich in deciding the nature of the application.

In many of the earlier works related to Android applications classification, these meta-data have been source for features extraction. Along with permissions comparison, *AForensic* also used meta-data information of third-party applications for triaging any suspicious applications (Di Cerbo et al., 2010). Printable string and meta-data such as *rating* and *file size* are also used to build the feature set (Sanz et al., 2012). *Manifest XML* based attributes and application *meta information* as feature are used to cluster app into *tool* and *business* category (Samra et al., 2013).

2.3.3 Code sample

As stated earlier, the *apk* file also has *compiled code* (.dex) which can be reverse engineered to get equivalent *Java class* files. The code has all the business logic which decides the real functionalities of any app. The code can be manually checked which is a time consuming and error prone process and require large number of *human resources* for analyzing the application code. To overcome the limitations of manual analysis, many features out of *Java class (after .dex to .jar conversion)* files are built and used machine learning to create malware detector or other Android application classifiers.

Dendroid which uses text mining approach to analyze and classify *code structures* in android malware families (Suarez-Tangil et al., 2014a). It uses *code chunks* as feature which is equivalence to a method of the apps. API calls along with permission were used to classify *over-privileged* malicious app (Geneiatakis et al., 2015). Features from *Java class* files and other *XML* files were extracted to build classifier to label an *apk* into *tool* or *game* category (Shabtai et al., 2010).

[4] `https://play.google.com/store`,Accessed:15-03-2017

2.3.4 Permissions

As aforementioned, every Android application has a *manifest* file which has required *permissions* by the application along with other information. Permissions system in the Android eco-system serves the purpose of *user-level* security, where an application will only get installed and run if all the required and requested permissions is granted by the user. To perform any hardware or software access, application must has related permission, for example, to access device's camera, the application must have permission related to camera access. Such explicit permission system also provides an opportunity to understand the application behavior by observing the required and requested permissions of an application.

Most of the works have used permissions-based feature set for building various classification system including Android malware detection system. A recent work has presented the detail study to understand Android's permission based security issues and its countermeasures (Fang et al., 2014). *Derbin*, followed statistical approach and included *used permissions* as features along with other static and dynamic features (Arp et al., 2014). *AForensic*, a forensic tool which extract permissions of third-party apps installed on a device and compare the set of extracted permissions to the *permission-based profile* built using Apriori algorithm which help to triage any suspicious application (Di Cerbo et al., 2010). Permissions (extracted from the applications and from the app markets) along with other features are used to build a feature set to train various machine learning based classifiers (Sanz et al., 2012). To group a given application into 34 different categories a *2-layer Neural Networks (NN)* were trained with only the extracted permissions as features (Ghorbanzadeh et al., 2013). One study considered *over-privileged* app as malicious and proposed a method that detects such app by using permissions along with API as features (Geneiatakis et al., 2015). The *requested permissions* with other static features were used to cluster apps into *tool* and *business* category (Samra et al., 2013). One of the earlier works has used *manifest's permissions* as boolean feature along with other static features for malware detection (Yerima et al., 2013). In another study permissions with API calls and combination of these two were used as feature set to train different machine learning algorithms for malware dectection (Peiravian and Zhu, 2013). MAMA created three feature set by extracting values

from different elements of manifest file (Sanz et al., 2013b). Among these three feature set, *permissions and features combined*, *permissions* and *feature-only*, combined feature set performed best. *Apk Evaluator*, is a permission based classification system for Android application. In training phase, it uses static analysis techniques to build a permissions based signature database which is used to characterize profile for Android applications in the evaluation phase (Talha et al., 2015).

It can be observed from this section discussion that *permissions-based* feature and feature set has been used extensively for malware detection and for various other tasks. Such uses established the discriminative potential of Android's permission system. This thesis has also proposed a weighted permission based feature set which improves the performance of machine learning algorithms further for detecting Android malaware. The details about feature set building process along with other methods and algorithms is presented further in Chapter 5.

2.3.5 Image based features

Recently, the idea of visualizing the application to classify applications is getting attention from many researchers. In one of the study a similar approach was implemented where the *byte code* of application was converted to machine level code and then the *opcode* instruction sequence was converted into image matrices (Han et al., 2013). Few recent works have also followed this basic approach i.e. converting application code to *images* and *entropy graphs* to detect and classify malware (Kancherla and Mukkamala, 2013; Han et al., 2015).

Another study has converted the *behaviour* of an application to color maps for malware detection (Shaid and Maarof, 2014). The study used a dynamic approach i.e. run the application in a Virtual Machine (VM) and generated color maps from the behaviour of application (Shaid and Maarof, 2014).

Converting Android applications to the image format makes it possible to use the tools and techniques from the field of image processing which is matured over time. This thesis has also used image features by converting benign and malicious Android sample into four different image format on the basis of color channels. Detail about the methods and algorithms are elaborated further in Chapter 6.

2.3.6 System call

The features presented earlier are static features which are extracted without executing the *apk* sample wheras the *system call* is a dynamic feature and can be extracted only after executing the *apk* sample. System calls are the OS functions which invokes by the user-application during the execution thus monitoring the system call reveals the true intention of running application. Creating feature set using system call would be very discriminative because of it represent the lower level of the application.

Most of the works have used *system calls* in various ways to draft features and create feature set to train machine learning algorithms. An android malware system was built on structural information of app which use embedded call graphs and classify the application based on the sequence of system calls (Gascon et al., 2013). A recent work has also defined a similar approach based on system calls for malware detection (Ham and Lee, 2014). An earlier study extracted static features such as API calls and Linux system commands to build a boolean feature set (Yerima et al., 2013). Permissions were used as feature along with API calls to build malware classifiers (Peiravian and Zhu, 2013).

This thesis is focused on static features so the *system calls* based feature set are not considered in current work. Table 2.2 summarizes the static features used in the literature for classification of Android applications including Android malware detection. The best accuracy of each work is mentioned in Table 2.2. Some of the works have used static features along with machine learning algorithms but either results are given on different metrics or not provided, such results are represented as NA (Not Available) in the table. Various observations have made during the literature review of the works related to *PE* and *apk* file based static features. Few critical observations out of these observations motivate to consider and carry out the works of this thesis. These observations and motivations are presented and discussed in the following section 2.4.

Table 2.2: Comparison of Android features used in earlier works

Previous Works	Type of Features								Performance	
	Used feature	Metadata	Code	Permissions	Image-based Feature	System call	Mixed	Others	Accuracy	FPR
(Di Cerbo et al., 2010)	✗	✓	✗	✓	✗	✗	✓	✗	NA	NA
(Shabtai et al., 2010)	✓	✗	✓	✓	✗	✗	✓	✗	92.20	0.190
(Sanz et al., 2012)	✗	✓	✗	✗	✗	✗	✓	✓	93.00	NA
(Sanz et al., 2013a)	✓	✗	✗	✓	✗	✗	✓	✗	86.41	0.19
(Sanz et al., 2013b)	✓	✗	✗	✓	✗	✗	✓	✗	95.22	0.06
(Samra et al., 2013)	✓	✓	✗	✓	✗	✗	✓	✗	71.00	NA
(Ghorbanzadeh et al., 2013)	✗	✗	✗	✓	✗	✗	✗	✗	65.1	NA
(Yerima et al., 2013)	✗	✗	✗	✓	✗	✗	✓	✓	92.10	0.061
(Peiravian and Zhu, 2013)	✗	✗	✗	✓	✗	✓	✓	✗	96.88	NA
(Gascon et al., 2013)	✗	✗	✗	✗	✓	✗	✗	✗	89.00	1.00
(Ham and Lee, 2014)	✗	✗	✗	✗	✓	✗	✗	✗	NA	NA
(Arp et al., 2014)	✓	✗	✗	✓	✗	✗	✓	✓	95.90	NA
(Suarez-Tangil et al., 2014a)	✗	✗	✓	✗	✗	✗	✗	✗	NA	NA
(Talha et al., 2015)	✗	✗	✗	✓	✗	✗	✗	✗	NA	NA
(Geneiatakis et al., 2015)	✗	✗	✓	✓	✗	✓	✓	✗	NA	NA

2.4 Observations and Motivations

By making comprehensive analysis on various static features and feature set, many observations have been made which motivated to carry out this thesis work. This section explains those observations which are related to the proposed feature set and discuss how these motivate to carry out the proposed works.

From PE headers based feature set, it is observed that all of the existing works have taken fields' value as it is for the header based features. Although these fields have a set of rules which must be satisfied by any benign PE file. Use of raw value increases the algorithms training time by having a diverse set of values. This motivate to create derived features which is used in this thesis as part of Integrated feature set. Derived features are those features which result from the comparison of field's value with the rules associated with the field. Details of the proposed integrated feature set are presented further in Chapter 3.

From the literature, it can be observed that *section name* of PE file is used as feature but have very limited use. *Section name* either used with other features or condenses to one or two features. For example, count of *suspicious* and *non-suspicious* section name, total number of sections are summarized representation of sections name. On the basis of earlier use of *section name* as feature and few statistical study about *section name* motivated to consider the potential of *section name* as feature and explore it further. This thesis has considered *section name* as boolean feature and has proposed a feature set only based on section name. The details about feature extraction and feature set creation process are explained in Chapter 4.

Permissions are extracted from *manifest* file of Android's *apk* and have been used rigorously as features for building classification system. Most of the research works have used permissions as features for building malware detector by using machine learning algorithms. One of the major limitations observed from the survey of existing works in this domain is the boolean representation of permission i.e. permission are used as boolean features. After performing few statistical comparison, a sharp difference between permissions pattern of benign and malicious Android applications motivate to look for alternative representation for permission based features. This thesis has proposed the *FAMOUS* (**F**orensic **A**nalysis of **MO**bile devices **U**sing **S**coring

of application permissions) model which incorporates a score based representation for permissions. These weighted permission representation are used to build a feature set which forms the basis of the machine learning model building process. The detail about feature building, scoring and about *FAMOUS* model is presented further in Chapter 5.

It is obvious from the literature that there is an upward trend for using visualization techniques and image feature (by converting program binary to image) for program classification. Earlier works are limited with the use of single color channel for binary to image conversion. There is also limited use of the image descriptors as feature. These limitations are the motivation to consider the proposed work which is based on various color channels. This thesis has used four color channels and converted each Android sample into four different image formats. In this thesis, the *GIST* image descriptor is experimented with all the image formats which helps to understand the best performing image format. The detail about *apk* to *image* conversion, feature extraction are explained and presented as algorithm in further Chapter 6.

2.5 Summary

This chapter has presented a discussion about static features for malware detection using machine learning algorithms. It has also presented and discussed the list of static features which can be extracted from Portable Executable (PE) headers and various files zipped inside Android's *apk* file. This thesis has used *PE headers' values*, *section name*, *permissions* and image based features, so a detailed discussion has presented in respective sections. All the feature classes have explained and supplemented with the discussion about earlier works related to the feature class. This Chapter provides motivations for this thesis on the basis of observations made during the study of the literature. The details of each of the proposed feature set are explained and discussed in forthcoming Chapter 3 to Chapter 6.

CHAPTER 3

Malicious Portable Executable Detection Using Integrated Feature Set

This Chapter explains and discusses the methodology of the proposed *integrated feature set* which is the amalgamation of the proposed set of *derived features* and other *raw features*. The values for the derived features are computed by using header fields, binary file data and the rule associated with each field. The values for the raw features are the value directly extracted from headers' fields. The objective of the proposed *integrated feature set* is to improve the performance of machine learning algorithms than earlier used raw feature set for malware detection.

This Chapter provides a brief introduction of PE file format by explaining it various headers along with various key fields of each header. Further, the method of both feature set (Raw and Integrated) generation is explained with the help of proposed *feature set generation* algorithm.

This Chapter, explains the *Raw* and *Derived* features in separate sections. Each of the *derived features* which encompasses entropy, year, packer information, number of suspicious sections, number of non-suspicious sections and 5 other boolean features are explained in detail. Details about *Dataset*, *Experimental System* and *Results* of Machine Learning algorithms are presented further in section 3.5.

3.1 Portable Executable File Format

Portable Executable (PE) is a file format used in Windows OS to construct, organized and execute different file types such as executable (.exe), (Dynamic Library (.dll) etc. Similar to *PE*, *Executable and Linkable Format (ELF)* and *Mach-o* is file format used

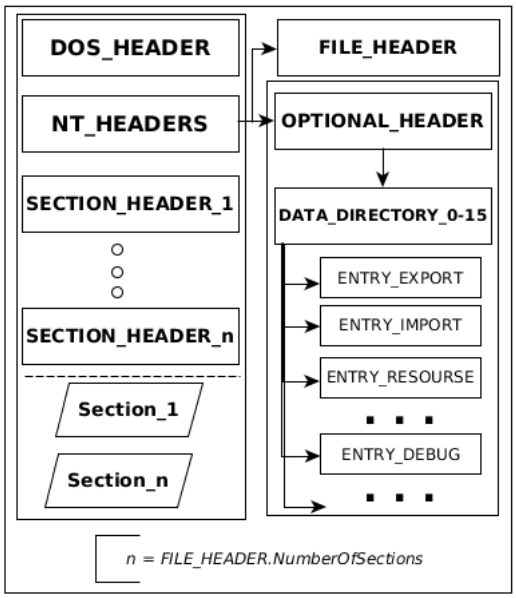


Figure 3.1: File format of portable executable

in *Linux* and *Mac OS X* respectively. File header part of the PE format is adopted from Unix Common Object File Format (COFF).

The PE format is implemented as a data structure to encapsulate the information necessary for the loader to manage and use various Windows files. Every file in PE format consists of two main headers (*DOS header* and *NT headers* explained further in Section 3.1.1 and Section 3.1.2) placed at beginning and further followed by some *section header* (one for each available section). Fig 3.1 shows the block diagram of PE file format where each header is represented as a block and blocks relation is shown through the one-sided arrow.

This section has given an overall view of PE format, further in following sections its various headers (DOS header and NT header) and sub-headers (File header and Optional header under NT header) are explained in details which will provide the necessary background to understand the features and feature set based on PE format. The important fields from every header are mentioned in discussion and explained under the respective section, further, every field is listed with a brief description in Appendix 2 to Appendix 4.

3.1.1 DOS Header

DOS header is for MS-DOS backward compatibility or graceful decline of new file types with a *stub* which inform that "This program cannot be run in Disk Operating System (DOS) mode". The header mainly has *relocation information* which helps executable to load multiple segments at arbitrary memory addresses. *DOS header* have a total of 19 fields out of which *e_res* and *e_res2* fields are reserved so it should have no value for any given sample. *e_csum* and *e_lfanew* which holds *checksum* and *next header address* respectively are very important fields for anomaly detection in any executable file. Appendix 2 lists out all the fields of *DOS header* and their short description [1]. *DOS header* is followed by *NT header* which has two sub-headers namely *File header* and *Optional header* which are explained and discussed in following sections.

3.1.2 NT Header

NT header which is subsequent to *DOS header* has three fields, a *signature* to recognize a PE file and two data structures, and these structures are PE specific headers (*File header* and *Optional header*). The value of Optional header's *magic* field is used to identify a 32-bit and 64-bit file type which is represented by *PE32* and *PE32+* respectively. Except *BaseOfData* in *PE32*, optional header has same fields for both *PE32* and *PE32+* formats.

File header contains info about the physical layout & properties of the file while *Optional Header* has logical layout information. Details about *File header* and *Optional Header* which sub-header under *NT header* is presented in following section 3.1.2.1 and section 3.1.2.2.

3.1.2.1 File Header

File header has abstract physical information about the PE file. It has seven fields, out of which the important fields are the *Machine*, which provides information about the target machine for which the executable is compiled for; the *NumberOfSections*, which tells about total available sections in file; the *TimeDateStamp*, which records time when the *File header* get generated or another way, it records the compiled time of the PE

[1] `http://www.pinvoke.net/default.aspx/Structures/IMAGE_DOS_HEADER.html`

file; and the *Characteristics* field which has a set of *bit flags*, each flag holding specific character of the file. All the seven fields of *File header* are listed in Appendix 3 with brief description. Inside *NT header* this *File header* is followed by the *Optional header* which is explained in following section.

3.1.2.2 Optional Header

Optional header is a sub-header within *NT header* and placed subsequent to *File header* inside PE file. It is a compulsory header for any PE file because it holds logical information about PE file which is used during linking and loading of the file. It has two sets of fields: standard fields and Windows specific fields. Eight standard fields are presented in *PE32* and nine are in *PE32+*, these fields are similar to COFF header of *Linux* system. Windows specific fields are total 21 in number and mainly have information that is required by the linker and loader in Windows OS. Appendix 4 lists out all fields of the *Optional header* which also includes *PE32* and *PE32+* fields.

This section explained about Portable Executable (PE) file format by discussing and explaining its various headers along with important fields of respective headers. After understanding about PE file, following section 3.2 explains about overall methodology adopted for the proposed *integrated feature set* (refer Figure 3.2) and method of feature set generation by discussing the proposed Algorithm 1. Later in section 3.3, *raw feature set* and section 3.4, *integrated feature set* are explained with its set of features.

3.2 Method for Feature set Generation

Feature set generation is the process of gathering values for each of the selected features from all the sample in the dataset. PE headers fields are used in different ways to create features and feature set. Values for each feature can be of different type i.e. Boolean, Integer, nominal etc. This work has used *Integer* and *Boolean/Binary* type of features. Fig. 3.2 depicts the overall process of creating *raw* and *integrated* feature set from raw *malware* and *benign* sample. These feature set are the input to the machine learning algorithms for comparing the performance of the two feature sets.

In Fig. 3.2, the process of building ML-based *malware detector* has divided into four main tasks: data collection, pre-processing, feature extraction and training & testing.

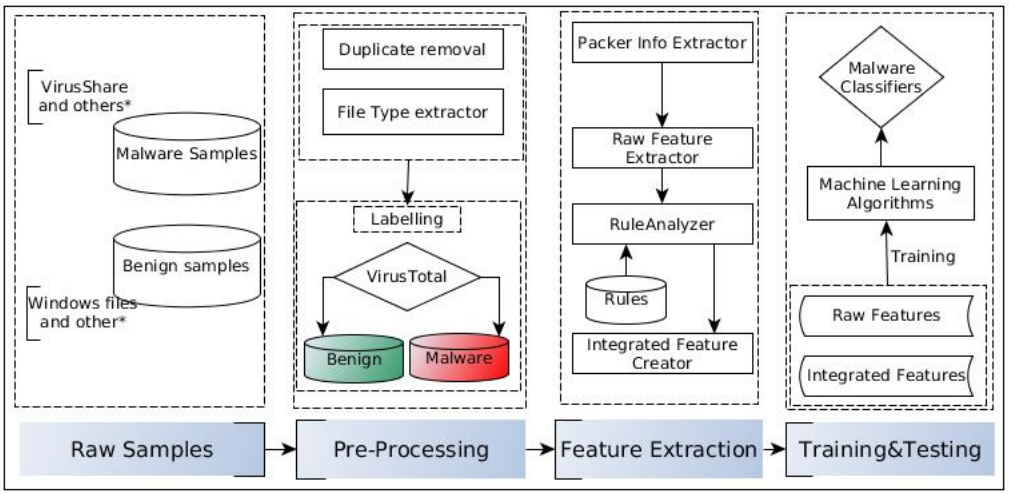


Figure 3.2: Block diagram of overall work flow

Data collection for malware domain is different from other areas of research because collecting and storing malware sample is risky and can harm the experimental system itself. New malware samples are coming daily with improved attacking methods and varying internal structure, in such case dataset for research must be updated with recent samples. In this work, old and recent malware samples were collected from public repositories whereas benign samples were collected from freshly installed Windows OS and other software repositories. Details about dataset used for the proposed *integrated feature set* is explained further in the section 3.5.1. *Pre-processing* is an important and immediate step after collecting malware and benign samples because these are computer programs so cannot be used directly with other feature extraction methods. During pre-processing stage tasks such as duplicate & unmatched (PE file type etc.) sample removal and representation conversion are carried out. The class labeling is a sub-process under pre-processing which assigns proper *class label* to every sample of the dataset for the supervised learning. After processing every sample process of feature extraction is carried out. During *feature extraction* every sample is passed through feature extractor module and value of each selected feature is extracted and stored with the respective class label. This work has extracted values for *raw features* and *derived features* which are explained in following section 3.3 and section 3.4 respectively. The output of feature extraction process is the feature set which has values for each feature from every sample of the dataset. During the training phase, the training part of the feature set is given input to machine learning algorithms and their performance is tested with the testing

part of the feature set. There are various ways to create training and testing feature set, the methods selected for this thesis work is explained and presented further in the section 3.5.3. The feature set generation is done in iteration by going through every sample of dataset and at every iteration it uses *feature extraction* method. Algorithm 1 summarizes the feature set generation process which is used for both feature (Raw and Integrated) set generation. It takes three input parameters i.e. $\Psi, \Omega, and \Phi$ which is *set of Malware sample*, *set of Benign sample*, and *the set of PE fields' rules* respectively. It return *Raw* and *Integrated* feature sets as output. The *set of PE fields' rules* is used for the *derived features* which is part of *integrated feature set*. Algorithm 1 is self explanatory, it loop through every sample of both malware and benign class and extracts *raw* and *derived* features for creating *raw* and *integrated* feature set respectively.

After generating the *Raw* and *Integrated* feature set through Algorithm 1, it pass as input to different machine learning algorithms. Using testing method feature set performance is tested on various metrics for each algorithms. The best performing features, feature set, and the algorithm can be recommended for building the real time malware classifier.

This section explained feature set generation process in general by not detailing about *raw* or *integrated* feature set. Further, a detail explanation is provided for both *raw* and *integrated* feature set in following sections. In section 3.3, details about *raw feature set* which is collection of only *raw features* is presented while section 3.4 discuss about *integrated feature set* which is an amalgamation of *raw* and *derived* features, so detail about these features are also presented in respective section.

3.3 Raw Feature Set

The *raw feature set* is based on the methods explained in the literature. It is created by using only *raw features* (i.e. fields value are not processed). In this thesis work, the *raw feature set* is created by extracting values from all fields of three main headers (*DOS header*, *File Header* and *Optional header, including standard and Windows-specific fields*) present in every PE file. Initially, *raw feature set* had 55 features in which 19 features were taken from *DOS header*, 7 features were taken from *File Header* and rest of 29 were taken from *Optional header*. During training *e_res* and *e_res2* fields

Algorithm 1 Feature set generation

procedure GENERATEFEATURESET(Ψ, Ω, Φ)
  $\alpha \leftarrow count(\Psi)$ ▷ Ψ : malware set
  $\beta \leftarrow count(\Omega)$ ▷ Ω : benign set
  $RawF[\alpha+\beta][] \leftarrow 0$
  $IntF[\alpha+\beta][] \leftarrow 0$
  for $\kappa \in \Psi$ **do** ▷ Extract features from malware
    $Class \leftarrow 0$
    $RawValue \leftarrow FetchFieldsValue(\kappa)$
    $FileValue \leftarrow ExtractFileFeatures(\kappa)$
    for $Value \in RawValue$ **do**
      $DerivedValue \leftarrow CheckRules(Value, \Phi)$ ▷ Φ : PE field rules
    $RawF \leftarrow UpdateRawF(RawValue \cup Class)$
    $IntF \leftarrow UpdateIntF(FileValue \cup DerivedValue \cup Class)$
  for $\kappa \in \Omega$ **do** ▷ Extract features from benign
    $Class \leftarrow 1$
    $RawValue \leftarrow FetchFieldsValue(\kappa)$
    $FileValue \leftarrow ExtractFileFeatures(\kappa)$
    for $Value \in RawValue$ **do**
      $DerivedValue \leftarrow CheckRules(Value, \Phi)$
    $RawF \leftarrow UpdateRawF(RawValue \cup Class)$
    $IntF \leftarrow UpdateIntF(FileValue \cup DerivedValue \cup Class)$
  return$(RawF, IntF)$

were removed from *raw feature set* because these two are reserved (according to the PE guideline document) and have no values for all samples in the dataset (including malware and benign sample). So, *raw feature set* finally has 53 features which were used for training and testing various machine learning algorithms. PE file processing and field's value extraction were performed on Linux machine with the help of *pefile* Python module.

Eq. 3.1 shows the features vector for raw feature set where *RawF* represents *raw feature set* and *DH*, *FH* and *OH* represents the header's fields of *DOS header*, *File header* and *Optional header* respectively.

$$RawF = DH \cup FH \cup OH \tag{3.1}$$

$$\begin{aligned} \text{where, } DH &= \{DH_1, \cdots DH_{19}\}, \\ FH &= \{FH_1, \cdots FH_7\}, \\ OH &= \{OH_1, \cdots OH_{29}\} \end{aligned}$$

This section explained about *raw feature set* and its various features which were selected from different headers. This feature set symbolized the method of feature extraction which were widely adopted in various earlier works, so in this thesis work it is created to compare the performance of the proposed *integrated feature set*. The training and testing of feature set with selected machine learning algorithms is presented and explained in section 3.5.3. In following section, the proposed *integrated feature set* along with selected *raw* and *derived* features are explained.

3.4 Integrated Feature Set

Integrated feature set is created by combining selected raw features (explained in section 3.4.1) and set of derived features (explained in section 3.4.2). The proposed method is designed to utilize the combinatorial benefits of both the raw and derived features as integrated features. Integrated feature set has total of 68 features in which 28 are same as in raw feature set, 26 boolean features are created by expanding individual flags of *Characteristics* and *DLLCharacteristics* as feature, these two are field of *File header*

and *Optional header Windows specific* respectively and 14 are derived features which are explained in further section 3.4.2. Benefits of using binary features for representing *DLLCharacteristic* and *Characteristics* is two folds. Firstly, using all bits of *DLLCharacteristic* and *Characteristics* as separate binary features will provide more information about PE file than a numeric value and secondly, the processing of boolean feature will be easier. Eq. 3.2, represents the proposed *integrated feature set* which is combination of three set of features (raw, expanded and derived). In Eq. 3.2, DH is *DoS header*, FH is *file header*, OH *Optional header*, *C* represents *Characteristics*, DC is used for *DLLCharacteristics* and subscript with these symbol represents the features count.

$$IntF = Raw \cup ExpandedRaw \cup Derived \tag{3.2}$$

$$\begin{aligned} \text{where, } Raw &= \{\{DH_1 \cdots DH_6\} \cup \{FH_1\} \cup \{OH_1 \cdots OH_{21}\}\}, \\ ExpandedRaw &= \{\{C_1 \cdots C_{11}\} \cup \{DC_1 \cdots DC_{15}\}\}, \\ Derived &= \{D_1 \cdots D_{14}\} \end{aligned}$$

Table 3.1 summarizes the count for raw features, expanded raw features and derived features which combined together *create integrated feature set*.

Table 3.1: Integrated feature set

Raw	Expanded	Derived	Total
28	26	14	68

This section provided an overview of the proposed *integrated feature set*, in the following section the *raw features* along with the fields which are used after expanded as binary feature and all the *derived features* are explained in details.

3.4.1 Raw Features

Raw features are those features for which values are directly extracted from PE header field for use. In this thesis work few of header's field value is used directly and selection of these header's fields are made on the basis of comparing their statistical properties (mean and standard deviation) between malware and benign samples. Windows OS

after *Windows3.1* version and Windows NT do not use *dos header* of PE file. So most of the fields of *dos header* are not useful as feature. Only the e_lfanew field of *dos header* is important which has offset value of the first byte of new PE header. This work has used a total of 6 fields out of total 19 fields from DOS header. Name of all the six fields are listed in the Appendix 1.

File header is sub-header of *NT header* which is subsequent to *DOS header* and it has abstract information about the whole file. *NumberOfSections* and *Characteristics* are more important than other fields. The linker uses these two fields' values to know about *total sections* and *type* of the file. In this thesis work, only *NumberOfSections* is used as raw features while *Characteristics* field's value is used without any changes but converted to boolean feature by using each flag as individual feature. From *Optional header*, except for *magic* field, all other standard fields are used as raw features in the proposed work whereas 13 windows specific fields are used as raw features while *DLLCharacteristics* field's value were used similar to the *Characteristics* field of *file header*. For a complete list of raw features used in the proposed work please refer Appendix 1. In following section, all the *derived features* are listed and a detail explanation is given for each feature in respective section.

3.4.2 Derived features

Derived features are features that are derived from the raw value of PE header by validating with the set of rules and the result of this process is taken as the feature value. Selection of headers' field for derived features are done on the basis of availability of a rule which confidently decides a value or set of value for the field.

Unlike *raw features* the value for the *derived features* are not the value extracted from header's field; instead, these are values that were amalgamated as result through comparing field's values against logical and documented rules. The derived features are either Boolean or integer. In the proposed work, the structural and logical specifications of PE file are adopted from a widely used document explaining structure of PE file (Pietrek, 1994) and online Microsoft's Microsoft Developer Network (MSDN) document [2].

[2] `https://msdn.microsoft.com/en-us/library/windows/desktop/ms680198(v=vs.85).aspx`

Figure 3.3: Example of valid and invalid TimeDateStamp value

For example, in *TimeDateStamp* field, the raw value will be simply an integer indicating a number of seconds since 1969 or if an encoding has used by extracting package then it will be in date format. Figure 3.3 shows that *ValidFile.exe* has a valid date where as *InvalidFile.exe* do not has a valid date, now by using the *TimeDateStamp* as raw value will not provide a discriminative feature so the proposed work do not use this value directly as feature. In this work *TimeDateStamp* field's value is converted to date and is compared with a range of valid date (From December 31st, 1969, at 4:00 P.M. to the date of the experiment) and resulted in Boolean output taken as features. Table 3.2 summaries all the considered derived values with it counterpart raw values.

In the following sections, all the *derived features* are explained in detail with reference to related works and example.

3.4.2.1 Entropy

Entropy can be defined as *measure of efficiency of information storage* (Shannon, 1948). Entropy directly relates to the packing of the file, packed file will have high entropy hence the high efficiency of information storage. The *pefile* module has method *get_entropy()* that calculate entropy of a given section data. This method gives result in the form of the quantity of bits per byte and thus the maximum entropy will be 8.0. With the inherent properties of measuring the degree of compression, entropy is a very reliable feature to check for packer and malicious behavior. Some modern malware or malicious Packers try to reduce entropy by inserting *zero bytes* in data to avoid detection (because many AV software only reacts to high entropy files) from anti-virus software. In the proposed work, the entropy of different section of PE and whole file were also calculated and treated as features. As malware has many different names for different sections, only standard section names were considered as the feature and if these sections name are present then their respective entropy was added to the feature

set else negative one were assigned as entropy value. In total three features (E_text, E_-data, E_file) were used based on entropy values. A recent work has used file entropy as two boolean feature *HighEntropy* and *LowEntropy* (Markel, 2015; Markel and Bilzor, 2014). A threshold value 7 were set to decide the value for these two boolean variables, entropy greater than 7, set *HighEntropy* to 1 while lesser value set *LowEntropy* to 1.

Table 3.2: Raw and Derived features

Features	Raw Value	Derived Values	
		Type	Values
Entropy	Binary value	Integer	[-1,0-8]
Compilation Time	Integer	Boolean	[0,1]
Section Name	String	Integer	–
Packer Info	NA	Boolean	[0,1]
FileSize	Integer	Integer	–
FileInfo	String	Boolean	[0,1]
ImageBase	Integer	Boolean	[0,1]
SectionAlignment	Integer	Boolean	[0,1]
FileAlignment	Integer	Boolean	[0,1]
SizeOfImage	Integer	Boolean	[0,1]
SizeOfImage	Integer	Boolean	[0,1]

3.4.2.2 File Creation Year

File Creation Year or Compilation time is very useful in identification of the malware from the benign program because malware has the very suspicious year of creation such as year early than 1980 or year beyond the current year, while the benign programs have the very genuine year of creation. 1980 is considered as a starting year because in the same year first DOS operating system was launched [3]. File creation year can be calculated by using *TimeDateStamp* field value of File header for a given PE file. In the proposed work, the calculated file creation year was checked with a valid range of year

[3] `http://windows.microsoft.com/en-IN/windows/history\#T1=era0[AccessedMay27, 2016]`.

i.e. 1980-2015, if the year is within the range than *1* were assigned as a feature value else *0* were assigned which indicates that sample has a suspicious year of creation.

3.4.2.3 Suspicious Sections Name

Suspicious Sections Name is also a very good indicator of the malicious file. The standard compiler gives a well-defined name to each section like *.data*, *.text*, *.rdata* etc. The non-standard or attackers build personal tools or packer gives either random name or a very different section name than the standard compilers which is considered as suspicious sections. The section names of each sample were extracted by using sections field value of PE header and these names of sections were compared with a set of standard section name (Pietrek, 1994) count of the match and non-matching section were assigned to two different features variable *TotalSuspiciousSections* and *TotalNonSuspiciousSections*. An earlier work has also used *number of suspicious sections* name as features for detecting packed and non-packed file (Perdisci et al., 2008).

3.4.2.4 Packer Info

Packer Info would be a very good indicator of malware and benign program because, in our analysis, it was found that nearly 18% of malware sample was packed whereas only 13% of the benign sample were packed. Benign and Malware both can be packed for different purposes. Benign programs are being packed to protect copyright and license key breaking attempt and also to stop reverse engineering of programs. Attackers pack their programs to bypass the signature-based detection because packed file change the byte structure of program and it becomes easy to surpass the signature. In the proposed experiment, *packer* is a Boolean feature and will have *0* if file is not packed and *1* if packed with any packer. Packer information of each sample is collected by using *PEiD* [4] signatures database with *yara* [5] (a Python module for signature matching).

3.4.2.5 File Size

File size is also used as a feature. The initial assumption was that malware and benign file will have a larger size difference. The malware writers try to keep file size

[4]http://www.aldeid.com/wiki/PEiD [Accessed May 27, 2016].

[5]http://plusvic.github.io/yara/ [Accessed May 27, 2017]

minimum that helps them to distribute over the web and hide within another program. Malware samples result smaller in size due to avoidance of Graphical User Interface (GUI). Benign programs are free from such size requirement and have genuine size as needed. All libraries and external resources are used as needed to run the program efficiently and effectively. The malware writers truncate the unused function and other external resources from their malicious program, only selective API functions and DLL are called within programs. In previous works, file size has not been considered as the feature but in the proposed work, it has taken as the feature of type integer that will has the file size in bytes.

3.4.2.6 File Information

FileInfo has set of strings, which contains the metadata about the PE file such as *FileVersion*, *ProductVersion*, *ProductName*, *CompanyName* etc. Benign programs incorporate a rich metadata while malware writers avoid putting metadata. In the proposed work, metadata information of all samples was tried to extract but it was found that many of malware sample does not have *FileInfo* field or other sub-section of this field. Set of all extracted strings about metadata were large so it was not feasible to consider these strings as features instead of that *FileInfo* feature were considered as Boolean and *1* was assigned if a sample has FileInfo metadata else *0* were assigned. A more comprehensive feature set of metadata information can be considered but the proposed work is limited to presence and absence of the *FileInfo* metadata apart from its value.

3.4.2.7 Image Base

ImageBase is a field in the optional header, which has the preferred address of the first byte of the image when loaded into memory (Pietrek, 1994). The condition applied to this field is *must be a multiple of 64k (64X1024)*. Default value for DLL is *0x10000000* (268435456 in decimal), Window CE exe is *0x00010000* (65536 in decimal) and *0x00400000* (4194304 in decimal) for other Windows OS like Windows XP. To make this field more useful, a pilot study is carried on the samples and it was found that 94.55% malware samples have specified default value while only 77.84% of the benign sample have specified default value and all the value from both classes follow

the condition of multiple of 64K. The value of ImageBase field is very specific to the type of file (DLL, exe etc.) so malware authors do not tamper this field often which is obvious from the aforementioned percentage. With the aforementioned result, it was decided to consider this field as Boolean and value were checked for default values and multiple of 64 conditions.

3.4.2.8 Section Alignment

SectionAlignment is the field in the optional header and it is the alignment (in bytes) of sections when they are loaded into memory. According to the Microsoft specification (Pietrek, 1994), it must be greater than or equal to *FileAlignment* and the default is the page size of the architecture. In the proposed work, this field is treated as Boolean and was compared against the specifications. If extracted value follows the specification then this field will have a value *1* else will be assigned value *0*.

3.4.2.9 File Alignment

The value of **FileAlignment** field of the optional header is the factor (in bytes) that is used to align the raw data of sections in the image file. Microsoft specification (Pietrek, 1994) states that this value should be power of 2 (between 512 and 65536). Default value is 512 and if the *SectionAlignment* is less than the architecture's page size, then this value must match with *SectionAlignment* value. The proposed work follows the given specification and considered this as the Boolean feature, which has value *1* if the extracted value matches the specification else *0*.

3.4.2.10 Size of Image

SizeOfImage gives the size (in bytes) of the image, including all headers, as the image is loaded in memory (Pietrek, 1994). It must be a multiple of *SectionAlignment*. The malware can manipulate this field to hide their malicious code. To verify this assumption, the specified condition was checked for malware and benign samples. It was found that almost all benign samples follow the specification but approx 4% (3.87%) malware sample does not follow the specification. To benefit the classification process, this is a significant proportion and hence this field is also considered as Boolean and assigned value *1* if match the specification otherwise *0*.

3.4.2.11 Size of Headers

SizeOfHeaders has the value that is equal to combined size of an MS-DOS stub, PE header, and Section headers and rounded up to a multiple of *FileAlignment*. This specification was also validated for malware and benign samples. Only 16% of benign samples do not follow the specification while 78% of malware samples do not follow the specification. In the proposed work, this field is considered as Boolean and assigned *1* if extracted value follows the specification else *0* were assigned. (David et al., 2016) have also presented and discussed the structured analysis of many of aforementioned fields and argue that considering these as the feature can enhance the detection rate.

This section starts with explaining about the proposed *integrated feature set* which is the combination of the set of *raw*, *expanded raw* and *derived* features. In further sections, *raw* and *derived* features are explained in details. The details of the experiments carried out to compare *raw* and the proposed *integrated* are presented and discussed further in section 3.5.3.

3.5 Performance comparison of Integrated versus Raw feature set

This section presents the performance comparison between the proposed *integrated feature set* and commonly used *raw feature set* on the basis of various experiments. The methods and process for creating these feature set are explained in aforementioned sections. This section provides details about the dataset and experimental system along with the pre-processing steps which are taken to generate both the feature set. The results of each experiment are presented and explained in detail in the respective sections. The following section 3.5.1 provides details about the dataset used to create both the feature sets.

3.5.1 Dataset

To create raw and integrated feature set, malware and benign samples were collected. The malware was collected from *virusshare* [6] [7] and benign samples were taken from freshly installed Window XP and Window 7 program files. Some of the benign samples were also collected from online free software archive [8]. Apart from this dataset, a separate test dataset (explained in detail in subsection 3.5.3.3) is also created for validation purpose which has 129 malware samples of different type and 30 benign samples.

3.5.1.1 Pre-processing

Collected malware and the benign samples can have the duplicate sample (same sample with the different name) and that will affect the training. Using filename to identify and remove duplicate sample may lead to errors, hence Message Digest (MD)[9] technique (MD5) was used to retain only the unique sample from both malware and benign group.

The proposed *integrated feature set* is based only on PE files such as EXE, DLL etc., so it was necessary to select and pass only PE files for next stage. There are many methods available for file type detection, among them, Linux's *file* utility was used to get the file type of each sample and a Python script was used to filter and retain only PE files. After duplicate removal and PE file selection, the final dataset has 2488 benign and 2722 malware samples.

3.5.1.2 Class Labelling

For the supervised learning, it is must to have accurate class label to each of the sample in the dataset. For labeling dataset for malware detection two methods are used in literature, *by locally installed AV engines* or through *online multiple AV engines*. This work has selected online multiple AV engines mode for labeling. *VirusTotal* provides sample scanning through multiple AV engines over Internet and it also offers API based service which helps to automate the scanning process. This thesis work used *Virustotal*

[6] http://virusshare.com/ [Accessed May 27, 2017]

[7] VirusShare.com is a repository of malware samples to provide security researchers, incident responders, forensic analysts, and the morbidly curious access to samples of live malicious code.

[8] http://download.cnet.com/ [Accessed May 27, 2017]

[9] Message digest is technique to create unique hash for the given data. Any change in data will change the hash.

service to verify the class label of each sample of the dataset. VirusTotal provides scan result of multiple anti-virus engines running in parallel but this work only considered and fetch top n AV result based on AV-test[10] result. The final decision on the class label was made using Eq. 3.3.

$$CL(M|B,S) = \begin{cases} M, & If(E_1(S) \vee E_2(S) \vee \ldots E_n(S)) \\ B, & elseIf(!(E_1(S)) \wedge !(E_2(S)) \wedge \ldots !(E_n(S))) \end{cases} \tag{3.3}$$

From Eq.3.3 it's very clear that a malware label is assigned to the sample S, if any of the scanning engines among E_1 to E_n returns positive whereas a benign label is only assigned to the sample S if all of the scanning engines returns negative.

3.5.1.3 Feature extraction

For supervised learning feature extraction is performed on the pre-processed and labeled dataset. All pre-processed and labeled samples (2488 benign and 2722 malware samples) were passed to feature extractor where the different header's fields value were extracted from the each data sample by using *pefile* [11] Python module. It is very efficient and popular module for PE file processing.

The raw feature set was created by just appending class label with each set of values extracted from the sample. The set of raw values and set of derived values were collected for creating integrated features set. Raw values were retained from the previous step and for derived values, a python script was written which has methods to check selected field's value with pre-defined rules (based on MSDN document) and return the result as feature's value. As explained in Algorithm presented and discussed in the earlier sections, all extracted fields value were passed and only selected fields' were checked for derived features. Appending selected raw with returned derived values and class label creates the integrated feature set. Some of sample header's fields' values were not readable which was the result of obfuscation and hence such sample were neglected by the extractor.

This section explained the process of dataset creation and also discussed about adopted pre-processing steps for cleaning the collected malware and benign samples

[10] https://www.av-test.org/en/antivirus/home-windows/

[11] https://github.com/erocarrera/pefile [Accessed May 27, 2017]

for the dataset. This section also explained about class labeling and feature extraction process. In the following section 3.5.2 details about the experimental system which was used to perform various experiments.

3.5.2 Experimental System

To create *raw* and *integrated* feature sets and perform all the experiments, an experimental environment was created with Ubuntu Operating system running on Intel(R) Core (TM) 2 Duo CPU E7400@2.80GHz processor and 4GB of primary memory and 320GB of secondary memory. Ubuntu system help to stop infecting experimental system because the sample was targeted for Windows based OS. Due to static analysis computation cost were very low and hence all experiments were carried on the normal end-host system. *Scikit-learn* (Pedregosa et al., 2011), Python based machine learning framework was used to run all the experiments. It was used to build and test the performance of different classifiers with raw and integrated feature set. Train-test split, supply of test dataset, and 10-folds cross validation testing method of *Scikit-learn* were used to perform various experiments for training and testing. It helps in comparing the performance of various models on many metrics such as accuracy, recall, precision, and f1-measure.

Scikit-learn has the implementation of many of well-recognized machine learning algorithms such as Decision Tree (DT), Logistic Regression (LR), Random forest etc. This work selected one algorithm from each group, these are grouped on the basis of underlying working theories such as tree-based, Bayesian or probability-based, instance-based, dimensionality reduction-based and ensemble-based algorithms. Finally, six algorithms Logistic Regression (LR), Linear Discriminant Analysis (LDA), Random Forest (RF), k-Nearest Neighbours (kNN), Decision Tree (DT) and Gaussian Naive Bayes (NB) were selected and used for all the experiments. All the six aforementioned algorithms were trained and tested with various configurations and different dataset for raw and integrated feature.

This section explained about the experimental system used for carry out various experiments. In the following section 3.5.3.1 to section 3.5.3.5 results of various experiments are presented and explained.

3.5.3 Results

Various experiments were conducted by training classifiers with Raw Feature (RF) set (only have PE headers field value as the feature) and Integrated Feature (IF) set (which have few selected PE fields' value and derived features). Their performance were measure on various metrics and a comparison were done fo find out best performing feature set.

The section 3.5.3.1 presents and explains experimental details and result for popular train-test split testing method.

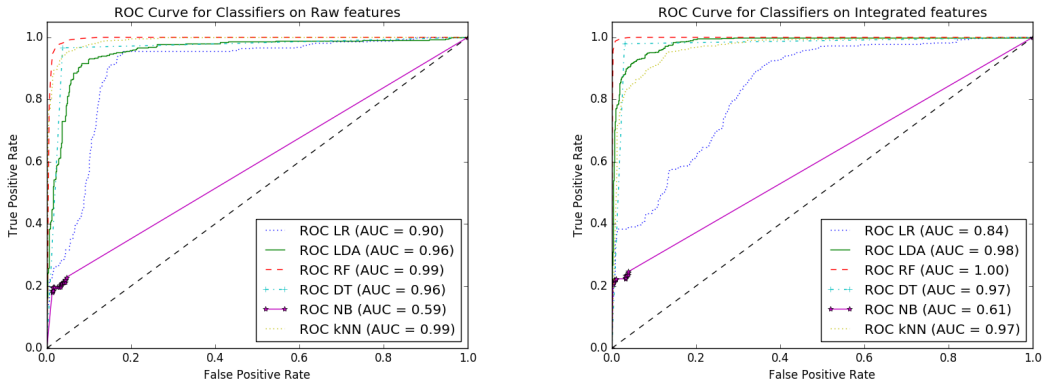
3.5.3.1 Train-Test split

The aim of this experiment is to study the performance of raw and integrated feature set with train and test method. The train-test split is a method of splitting the dataset into two part training and testing. It takes the percentage as splitting threshold. The training dataset is used for training the machine learning algorithm while testing dataset test the classifier's performance. This work used 70-30 ratio for splitting the raw and integrated feature set and hence trained algorithms on 70% training dataset and tested with remaining 30% test dataset.

Table 3.3: Classifiers result on train-test(70-30) split

Classifier	Accuracy		Precision		Recall		F1-score	
	RawF	IntF	RawF	IntF	RawF	IntF	RawF	IntF
LR	77.06	78.12	0.81	0.80	0.77	0.78	0.76	0.77
LDA	91.71	92.45	0.92	0.93	0.92	0.92	0.92	0.92
RF	97.43	98.78	0.97	0.99	0.97	0.99	0.97	0.99
DT	96.47	97.12	0.96	0.97	0.96	0.97	0.96	0.97
NB	56.04	50.09	0.74	0.77	0.56	0.58	0.48	0.51
kNN	94.73	90.79	0.95	0.91	0.95	0.91	0.95	0.91

Table 3.3 summarize the performance of all the six classifiers on selected metrics with the train-test split. It can be observed that NB (accuracy $50-60\%$) performance is minimum while Random Forest (accuracy 97.47% and 98.78% on raw and integrated feature set respectively) performance is best among all classifiers on both dataset and



(a) Raw features

(b) Integrated features

Figure 3.4: ROC curves for different classifiers with train-test split method

integrated feature set is performing better ($accuracy + 1.31\%$) than raw feature set. The integrated feature set with Random Forest is performing better than raw feature set on all metrics while other classifiers are also better on all metrics except kNN which is giving better result on raw features than integrated feature. Figure 3.4 shows the ROC curve for classifiers trained and tested on the raw and integrated feature set. The ROC and AUC values are also indicating similar performance as other metrics that are integrated feature set have better AUC value than raw feature set. Random forest performance is best among all another classifiers.

This section explained and presented results of train-test split method for all the six classifiers on various performance metrics. In the following section 3.5.3.2 10-folds cross-validation method is explained and results of selected six classifiers are presented.

3.5.3.2 10-fold cross validation

The aim of this experiment is to test performance of raw and integrated feature set in more robust testing environment, so this work has used 10-fold cross validation method for testing. It is robust again over-fitting and under-fitting which is common in machine learning algorithms. The train-and-test method of validation has two main limitations. First, if a single train-and-test experiment get a split which does not represent each class sample proportionally then the holdout estimate of *error rate* will be misleading. Second, it reduces the training sample which would degrade the performance of the classifier. To overcome these limitations and have a more robust method for validation, this

work performed this experiment with cross-validation method. The Cross-validation is represented as $k|n$-fold cross validation where k or n represent the number of fold the dataset will be split and the number of times the training and testing will be done. For example, in a *k-fold cross-validation*, the dataset will be split in k folds and for k times the $k-1$ folds will be used for training and left out one fold will be used for testing. The final result will be given as the average of all k folds. The k repeats and random selection of fold for testing makes cross-validation method robust.

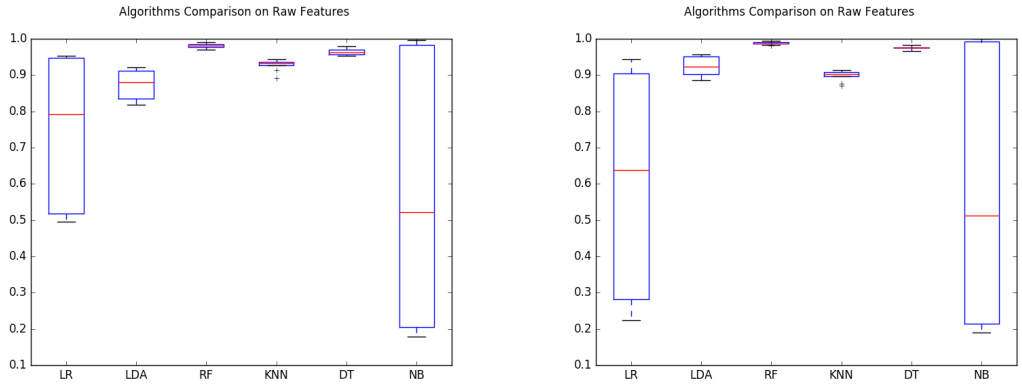
Table 3.4: Accuracy comparison of classifiers with 10-fold cross-validation

Classifier	Accuracy		Difference
	Integrated	Raw	
LR	0.600	0.742	-0.142
LDA	0.923	0.874	+0.049
RF	0.984	0.977	+0.007
DT	0.974	0.960	+0.014
NB	0.583	0.578	+0.005
kNN	0.899	0.928	-0.029

This work performed a 10-fold cross-validation of all the classifiers on raw and integrated feature set. For the comparison, the accuracy of each classifier was measure for both raw and integrated feature set. Table 3.4 shows the 10-fold cross-validation output of all the classifiers. It can be observed that accuracy of all classifiers is decreased by nearly 1% than the accuracy achieved under test-split validation but difference column clearly shows that integrated feature set is performing better than raw feature set. Four classifiers (LDA, RF, DT, and GaussianNB) have higher accuracy while other three classifiers (LR and kNN) are very close in the accuracy of the raw feature set. Figure 3.5 shows the box plot output of classifier's accuracy collected during 10-fold cross validation. It is observed that Random forest and Decision Tree have high accuracy and low variation where others have more variation and low accuracy.

Among the classifiers, with same configuration Random forest (RF) has consumed highest training time on both raw and integrated features set which are 9.51 & 8.25 seconds respectively, whereas Decision Tree (DT) took lowest training time for both raw and integrated features set, 0.83 and 0.6 second respectively. It was also observed

that training time is less on integrated features set, for example, both RF and DT took less time than raw feature set.



(a) Raw features

(b) Integrated features

Figure 3.5: Accuracy box plot of classifiers with 10-fold cross validation

This section explained the 10-fold cross-validation testing method and presented the results of all six classifiers which were trained with raw and integrated feature set. In section 3.5.3.3 the results of classifiers are presented with respect to a new *test dataset*. The new test dataset has all the unique samples which were not present in the raw and integrated feature set.

3.5.3.3 Testing with new test dataset

The aim of this experiment is to test the performance of raw and integrated feature set with new samples which represent the real world scenario where often *unknown samples* are given to classify. The test dataset is created with samples which are not included in training dataset and used to validate or test the performance of the trained classifier. After testing and comparing the performance of classifiers on the raw and integrated feature set with train-test and cross-validation method, this work also performed an experiment to validate the efficiency of integrated feature set with new test dataset. For creating the new test dataset 129 unique malware samples (which are not present in earlier collected samples) of different type were collected and mixed with 30 benign samples randomly selected from earlier collected samples. The feature extraction and feature set generation were kept same as explained earlier.

Table 3.5: Type of malware samples and count in test dataset

Malware Type	Virus	Worm	Trojan	Bot	Spyware	Downloader	Backdoor	**Total**
Initial Count	20	20	20	9	20	20	20	129
After extraction	5	18	19	7	20	13	20	102

Table 3.5 shows the count of each malware type that was included in the test dataset. All the samples were downloaded from *openmalware* [12] public malware repository and the same class label was kept as given by the aforementioned archive. The MD5 hash comparison was done to ensure that none of the training dataset samples should be present in the new test dataset. After pre-processing and feature extraction step, the final test dataset had totally 102 malware and 30 benign samples.

All the six selected learning algorithms were trained with the feature set created from a total of 5180 samples from both malware and benign class. The classifiers performance was validated with the *new test dataset* which have 132 samples. The result of raw and integrated feature set with the new test dataset on various metrics is summarized in Table 3.6.

Table 3.6: Classifiers performance on test dataset

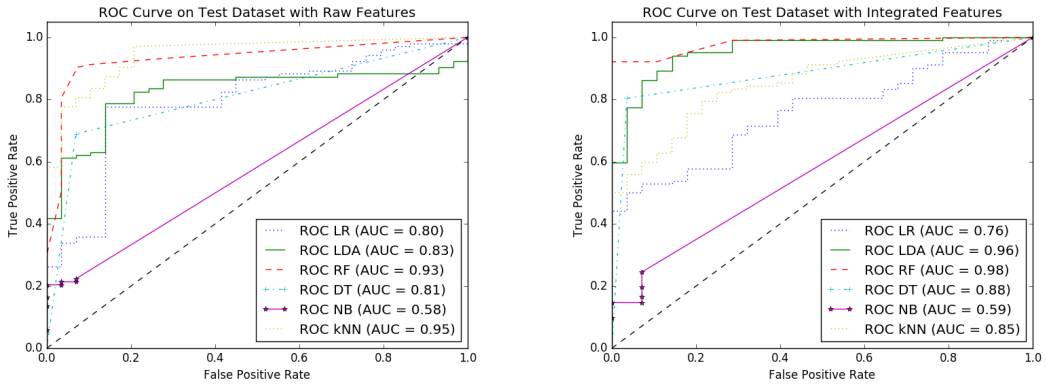
Classifier	Accuracy		Precision		Recall		F1-score	
	RawF	IntF	RawF	IntF	RawF	IntF	RawF	IntF
LR	73.48	70.00	0.77	0.77	0.73	0.70	0.75	0.72
LDA	81.82	88.46	0.85	0.90	0.82	0.88	0.83	0.89
RF	74.24	89.23	0.87	0.93	0.74	0.89	0.76	0.90
DT	74.24	83.85	0.86	0.90	0.74	0.84	0.76	0.85
NB	28.79	31.54	0.83	0.84	0.29	0.32	0.21	0.26
kNN	79.55	76.15	0.88	0.83	0.80	0.76	0.81	0.78

It can be observed from Table 3.6 that raw feature set based classifiers performed

[12]http://oc.gtisc.gatech.edu:8080 [Accessed May 27, 2016]

poorly on the new test dataset. The raw feature set has an accuracy range $73\% - 81\%$ for classifiers low to high. The performance of integrated feature based classifier also reduces and best classifier Random forest has only 89.23% accuracy which is 9.17% less than the 10-folds cross validation result. With comparison to raw, integrated feature set performance is very high. Random forest and Decision tree have only 74.24% accuracy on raw features while 89.23% and 83.85% accuracy with Random forest and Decision tree respectively on integrated features.

Figure 3.6 shows the ROC curve and AUC value of different classifiers on raw (Ref. Figure 3.6a) and integrated (Ref. Figure 3.6b) feature set. It is clearly evident that integrated feature set is performing better than raw feature set and have 5% more AUC value for Random forest and other classifiers also have high AUC values for integrated than raw feature set.



(a) Raw features

(b) Integrated features

Figure 3.6: ROC of classifiers on test dataset

This section explained the method of testing with a new dataset which has all unique samples than the training dataset. The section also presented the results of six selected classifiers on various metrics for raw and integrated feature set. The section 3.5.3.4 presents the comparison of the proposed integrated feature set with the earlier similar works which have used feature set based on PE files.

3.5.3.4 Comparison with previous works

In the previous three experiments, it was found that the proposed integrated feature set is performing better than the raw feature set.

To further validate the accuracy performance of proposed integrated features set, it is important to compare the proposed work with earlier works based on PE files and its header information. Best of our knowledge, except Markel and Bilzor (2014), no other malware classification work has been carried out based on only headers' fields values but David et al. (2016) have discussed the structured analysis of various fields of PE header and argue that considering these as the feature can enhance the detection rate. There are few works close to the proposed work but most of them have used PE header's based feature as complementing feature and have used them with other features set such API & DLL calls, byte-n-gram and opcode-n-gram.

Bai et al. (2014) have used API & DLL calls along with various raw values of PE headers. Comparison with (Bai et al., 2014) work will not give a perfect merits or demerits of the proposed work but can be tested for the nearby result. For comparing the proposed integrated feature set, Bai. et al. work was reproduced Bai et al. (2014). The selection of previous work was based on the similarity of work i.e. learning algorithms and used feature set.

Table 3.7: Result comparison of the proposed work with the earlier work

Works	DT	RF
Proposed work	97.4	98.4
Bai et al. (2014)	98.7	98.9
Markel and Bilzor (2014)(F-score)	97.0	–

Table 3.7 summarizes the comparison result of the proposed work with the previous works. From Table 3.7 it can be observed that Random forest with proposed feature set has an accuracy of 98.4% which is only .5% less than the accuracy of Bai et al. (2014) 98.9%. The improvement of .5% would be due to using of API & DLL calls along with PE headers' fields values.

This section presented the comparative result of the proposed integrated feature set with the earlier works. In section 3.5.3.5 performance of classifiers are tested with selected features and results are presented.

3.5.3.5 Testing with selected Features

The feature selection is the process of selecting a subset of relevant features for use in model construction [13]. Generally, feature selection is performed before training and testing and reduces the feature set dimension by selecting only features with high discriminative value. The focus of the proposed work is to test the classifiers performance with raw and integrated feature set hence feature selection were avoided on both the feature set.

This work is interested in knowing the importance of features' rank in raw and integrated feature sets which can further help to understand the difference and importance of features among both sets. Among model based feature selection methods (wrapper methods) tree based methods are easiest to apply and without much tuning, it can also model for non-linear relations.

In this work *Extra Trees Classifier* with 250 trees was used to get feature importance for the raw and integrated feature set. Except number of trees all other settings kept to default. It is an ensemble classifier and by default uses *Gini impurity* to measure the quality of a split. Table 3.8 list out the top 10 features from the raw and integrated feature set. In top 10 ranked features of integrated feature set, it can be observed that 3 features are derived features, 6 features are boolean which are expansion of *Characteristics* and *DLLCharacteristics* raw features and *SubSystem* is common in both which clearly indicates the importance of derived features.

[13] https://en.wikipedia.org/wiki/Feature_selection

Table 3.8: Tree method based selected top 10 features from raw and integrated feature set

Raw		Integrated	
Features	Value	Features	Value
Characteristics	0.202	FH_Char12	0.213
ImageBase	0.107	OH_DLLChar2	0.082
DLL Charaicteristics	0.079	fileinfo	0.080
MajorSubSystemVersion	0.064	OH_DLLChar0	0.072
SubSystem	0.056	FH_Char0	0.064
MajorOSVersion	0.037	SubSytem	0.031
MinorSubSystem	0.030	E_data	0.027
e_Ifanew	0.029	E_file	0.025
SizeOfStackRes	0.028	OH_DLLChar6	0.0248
CreationYear	0.026	FH_Char3	0.0241

To verify the suitability and efficiency of proposed integrated feature set, five subsets of features were selected from raw and integrated feature set. The selection was made by considering aforementioned feature ranking and five subsets were created as top5, top10, top15, top20 and top25 having top 5, 10, 15, 20 and 25 features respectively. On this five feature sets Random forest and Decision tree classifiers were trained with 10-fold cross validation and their accuracy was recorded and compared. Figure 3.7 shows the accuracy of RF and DT on the raw and integrated feature set. From Figure 3.7 is evident that performance of raw feature is better till top 15 features above that accuracy is constant and more features does not improve accuracy any further where as integrated feature set have lower or equivalent performance below top 15 features but get improved over top 15 and have greater accuracy than raw feature set. The reasons for this behavior is very obvious, *Characteristics* and *DLLCharacteristics* is in top 5 features in raw feature set whereas it is separated in various boolean features in integrated feature set hence top 15 features provides all accuracy for the raw feature set. While in the case of integrated feature set more than 15 top features improves accuracy and overtake the accuracy of raw feature set because of other informative features apart from *Characteristics* and *DLLCharacteristics* based boolean features. Top 20 features

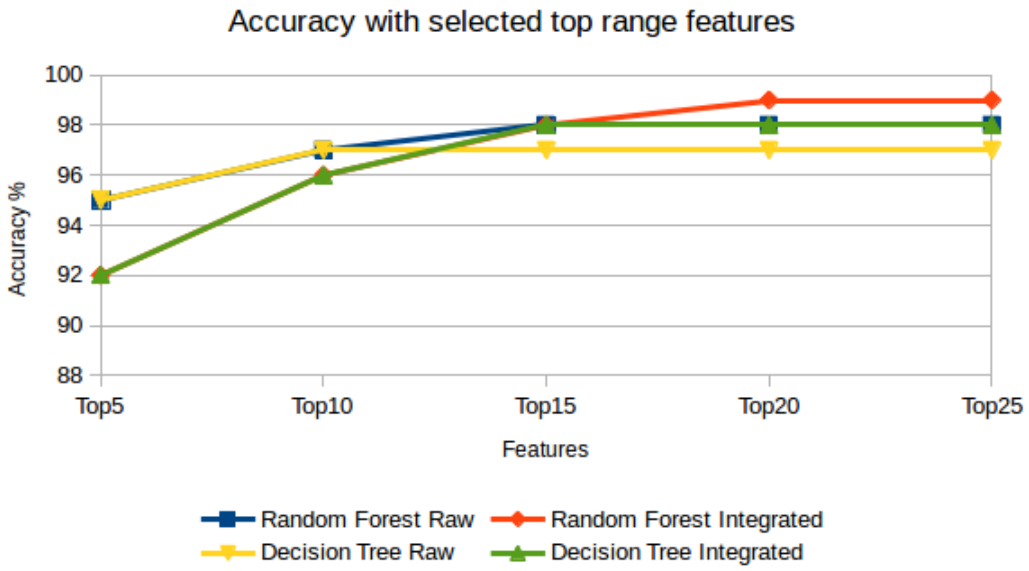


Figure 3.7: Decision tree and random forest accuracy on top N features

of integrated features set have 6 out of 14 derived features.

This section presented the experimental proof of performance improvement of the classifiers with the proposed integrated feature set. With the help of five experiments performance comparison of raw and integrated feature set is presented and the improvements are noted and discussed with the valid reasons. The following section 4.3 presents details of the dataset, experimental system and results of various classifiers for the proposed *section name-based feature set*. The section presents results for showing the effect of feature selection on the proposed *section name-based feature set* which helps to understand the discriminative potential of the proposed feature set.

3.6 Summary

This Chapter proposed an *Integrated feature set* which is combination of the set of *derived features* and the set of selected and *expanded raw* features based on Portable Executable (PE) file format. The Chapter has also provided details about building *raw feature set* which is created by using PE header fields' value without any change, and were adopted in many earlier works as explained in Chapter 2. This Chapter gave an overview and discussed all the steps of building machine learning based malware detector from the raw malware and benign sample. An algorithm for generating feature

set from malware and benign sample is also presented and discussed which was used to generate *raw* and the proposed *integrated* feature set. Various selected ML algorithms are trained and tested on both feature set and their performance is measured. This Chapter also presented the details about dataset, experimental system, and results of training and testing. In the forth coming Chapter 4, a *section name* based binary feature set is proposed which uses Portable Executable (PE) file's section name as features.

CHAPTER 4

Malicious Portable Executable Detection Using Section Name

In Chapter 3, Portable Executable (PE) headers' fields are used to create the *integrated feature set* which is a combination of the set of *raw* and *derived* features. In this Chapter, *section name* of PE files are used as boolean features to create the proposed *section name based feature set*. The objective of the Chapter is to test the discriminative potential of *section name* as the boolean feature. To achieve this objective, various machine learning algorithms are trained with the proposed *section-name based feature set* and their performance was tested on various metrics.

This Chapter explains in details about PE *sections* and *section name*. Various common *sections*[1] present in PE files are also listed and explained in detail. With *Static analysis*, sections name can be extracted by all types of PE files such as EXE, DLL and COM. *Static analysis* is *simple* and *cost effective* so building feature set from *section name* is fast and suits malware classification task. Various experiments confirmed the usability of *section name* as features. Malware classifiers built with *section name* based feature set achieved a good detection accuracy and have low False Positive Rate (FPR).

In section 4.1, *PE sections*, *section table* and a few frequently used *sections name* with their characteristics are explained in detail. During statistical study, a significant difference in *sections name* frequency in malware and benign sample provides motivation to use it as feature for malware detection. This Chapter has also listed the selected top 20 *sections name* with their *information gain* value.

This Chapter presents the method for building the proposed *section name-based feature set* from PE sections. The details about the dataset, experimental system and result

[1]This is decided based on traditional programming naming conventions and use frequency.

of training & testing various machine learning algorithms are presented and explained towards the end of chapter under section 4.3.

4.1 Portable Executable Sections

PE is a file format used in Windows OS to construct, organize and execute different file types such as executable (.exe), Dynamic Library (.dll) etc. All the aforementioned type of PE files are organized and structured in various *sections*. So the *section* is very important unit of a PE file because it holds the contents of the file such as *code*, *data* and other *resources*. All the available sections have a respective section header which are stacked in sequence and these headers succeed the two other PE headers namely, *DoS header* and *NT header*(explained in detail in previous Chapter 3). Fig. 4.1 visualize the arrangement of *section headers* within the *section table* with its respective section. As stated earlier, left part of Fig. 4.1 shows all the sections (*n*) and its respective section header along with other two PE headers while right part of Fig. 4.1 lists out all the fields of the *section header*. The interest of this thesis is in the *section name* which is first field of *section header* which is termed as *Name1* [2] in the figure.

There is a standard *naming convention* for section name such as *.text*, *.rsrc* etc. but these names vary due to the linker and other development tools[3]. The standard section name also indicates about the nature of the section (explained in detail in further section 4.1.2, for example, *.text* or *.code* usually has executable code, while *.data* or *.idata* has initialized data [4].

The *number of sections* (represented as *n* in Fig. 4.1) in PE file is not fixed and it vary from file to file. The total number of sections present in a PE file depends upon functionality and compilation and the count of total available sections can be known by accessing *NumberOfSections* field's value of the *File header*. The *Section name* is an eight-byte array of ASCII characters, and it is meant for human i.e. loader and linker do not use section name during the execution. The deceptive presence of this memory space opens scopes for misuse and so is the motivation for the proposed *section name*

[2]This field would has referred as "name" but Microsoft Macro Assembler (MASM) also has one keyword "name" so different words such as *Name1* is being used for the field.

[3]`http://wiki.osdev.org/PE#Section_header`

[4]`https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files`

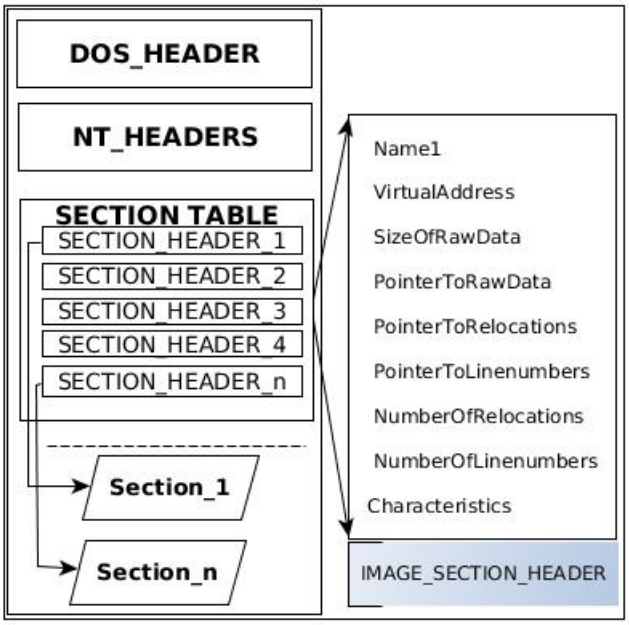


Figure 4.1: Section header and section of the portable executable format

based feature set.

This section explained about PE file in brief, a detailed explanation is provided in the previous Chapter 3. This Chapter also provides a brief introduction about sections, section table, section name and section header. In following section 4.1.1, section table is explained in details which has data structure to hold the *section header* for all the respective *sections* of the PE file.

4.1.1 Section Table

A *Section table* is actually an *array of structure* which is positioned immediately after the *optional header* which is part of *NT header* in a PE file. The structure for the *section table* is called *IMAGE_SECTION_HEADER* i.e. each row of the *section table* is a section header. The number of entry in the *Section table* is determined by *NumberOfSections* field in the file header (*IMAGE_FILE_HEADER*) structure. The location of the *section table* is determined by calculating the location of the first byte after header by using the size of the optional header.

Each *section header* has at least 40 bytes of entry. Few important entries i.e fields of the *section header* are explained in details in following sections 4.1.1.1 to 4.1.1.6.

4.1.1.1 Name

This thesis has used the *section name* as binary feature to build malware detector. The *section name* is extracted from the *Name* field of *section header*. The *Name* field of the *section header* is an 8-byte null-padded Unicode Transformation Format (UTF)8 encoding string and contains the *name of the section* and it even can be *null*. This field would has been referred as "name" but MASM also has one keyword "name" so different words such as *Name1* is being used for the field.

The maximum length of the *name* field is 8 bytes. The *section name* works like a label for the respective section and has no further use in loading or execution of PE file. Any name can be used for this field or even it can be left blank. It is also worth to note that the guideline does not mention anything about the terminating null i.e. the *section name* field is not an ASCIIZ[5]string so it is not expected that it must terminate with a null. A string of exactly 8 characters long does not has a terminating null. The section name longer than 8 characters is achieved by using the *string table* in which case the *section name* field has a slash (/) followed by the ASCII representation of a decimal number that serves as an offset into the string table. *Section name* can not be longer than 8 characters for the executable image file because it does not use a string table. In the case of *object* file, longer names are truncated if the file is going to be an executable file.

From this section, it can be observed that the *section name* field of the *section header* is very deceptive in its structure and uses. Such deceptive nature of the *section name* widen the malicious opportunity for the attacker and make it complex for static analysis to decide nature of PE file based on the *section name*. This motivates this thesis work to devise binary features based on *section name*. The proposed *section name based feature set* can be used with machine learning algorithms to detect malicious PE files.

In following sections other fields such as *virtual size*, *virtual address* etc. of *section header* are explained which help to understand the use of *sections* and its *header*.

[5]ASCIIZ means that the string is terminated by the \0 (ASCII code 0) NULL character.

It also called *C strings*. In computing, a C string is a character sequence terminated with a null character (\0, called NUL in ASCII).

4.1.1.2 VirtualSize

The *Virtual size* field of section header has actual size (in bytes) of the section's data when loaded into memory. This size can be less than the size of the *section on disk*. The *Virtual size* value greater than *SizeOfRawData* indicates that the section is *zero-padded*. For only *executable images* this field is valid and should be set to zero for *object files*.

4.1.1.3 VirtualAddress

This is the Relative Virtual Address (RVA) of the section. The PE loader examines and uses the value in this field when it maps the section into memory. In executable images, this field has the address of the first byte of the section which is relative to the image base when the section is loaded into memory. Thus if the value in this field is 1000*h* and the PE file is loaded at 400000*h*, the section will be loaded at 401000*h*. In the case of *object files*, it has the address of the *first byte* before relocation is applied.

4.1.1.4 SizeOfRawData

This field holds the *size of the section* for object file and the *size of the initialized data* on the disk for the image file. The size of the section's data is rounded up to the next multiple of the *file alignment*. For the executable image, this value must be a multiple of the value of the *File Alignment* field of the *optional header*. If this field's value is less than the value of the *Virtual Size* field then it indicates that the section is *zero-filled*. The PE loader examines the value in this field to know, *how many bytes of the section* it should map into the memory during loading. For a section which has only *uninitialized data*, this field should be zero.

4.1.1.5 PointerToRawData

This field is used to find the starting address of the respective section. The value of *PointerToRawData* field is the file offset of the beginning of the section. This is very useful because the PE loader uses the value in this field to find the location of the *section data* within the file.

4.1.1.6 Characteristics

The characteristics field of the *section header* has set of flags which describe the characteristics of the respective section. It has flags which indicates that the section contains *executable code*, *initialized data*, *uninitialized data* or has *read/write* permission etc. Due to holding such sensitive information about the particular section, this field is very important for malware detection. In many cases, it has been observed that the attacker hide malicious code inside sections by manipulating the flags of the characteristics field. Table 4.1 lists out some of important flags of *Characteristics* field of the *section header* along with their decimal equivalent value and with the short description. These flags could be potential binary features for malware detection but are out of scope for the proposed *section name based feature set*.

Table 4.1: Important flags of section header's characteristics field

S.N.	Flag	Value	Description
1	CNT_CODE	32	contains executable code.
2	CNT_INITIALIZED_DATA	64	contains initialized data.
3	CNT_UNINITIALIZED_DATA	128	contains uninitialized data.
4	LNK_NRELOC_OVFL	16777216	contains extended relocations.
5	MEM_DISCARDABLE	33554432	can be discarded as needed.
6	MEM_NOT_CACHED	67108864	cannot be cached.
7	MEM_NOT_PAGED	134217728	not pageable.
8	MEM_SHARED	268435456	can be shared in memory.
9	MEM_EXECUTE	536870912	can be executed as code.
10	MEM_READ	1073741824	can be read.
11	MEM_WRITE	2147483648	can be written to.

Previous section presented important fields of *section header* and provided a detailed explanation for each of the fields. In section 4.1.2, sections which are more frequent and present in the PE file are presented with a detailed explanation.

4.1.2 Common Sections of Portable Executable (PE) file

As stated earlier that PE file is organized and structured in various *sections* and *each section* has a respective *header* comprising many important fields used by loader and linker. Usually, linker and loader process section with code or data without any explicit knowledge of the section contents. The contents of the section are relevant only to the application that being linked or loaded. However, some sections have explicit meanings with their presence in object file or image file. The explicit meanings of the section is recognized by various tools and loaders. Each section indicates a special function which can be identified by checking the *set flags* of *Characteristics* field of the respective *section header*, special locations in the image and even the *section name* (by convention). For example, an application in the *Windows NT* typically has nine predefined sections, such as *.text*, *.bss*, *.rdata*, *.data*, *.rsrc*, *.edata*, *.idata*, *.pdata*, and *.debug*. Because these *section names* are more frequent for *Windows NT* application, many other supporting tools also try to follow these *section names* as convention but these names are not mandatory.

Further, frequently used *section names* have presented with discussion about their conventional functionality by relating them with required set flags of *Characteristics* field listed in Table 4.1.

4.1.2.1 .text/.code

The *.text* section of the PE file usually has the *executable code* which also named as *.CODE* or *.code*. As Windows uses a page-based virtual system, having one large code section is easier to manage for both the OS and the application developers. This section set three important flags of *Characteristics* field, *IMAGE_SCN_CNT_CODE*, *IMAGE_SCN_MEM_EXECUTE*, and *IMAGE_SCN_MEM_READ*.

4.1.2.2 .idata

The *.idata* section contains various information about imported functions, including the import directory and import address table of the PE file. This section has *read* and *write* permission by enabling *IMAGE_SCN_MEM_READ* and *IMAGE_SCN_MEM_WRITE* flag of *Characteristics* field in its header.

4.1.2.3 .rdata

The *.rdata* represents the read-only data on the file system, such as *strings* and *constants*. To achieve desired properties, this section set the *IMAGE_SCN_MEM_READ* and *IMAGE_SCN_CNT_INITIALIZED_DATA* flags of *Characteristics* field.

4.1.2.4 .edata

The *.edata* section contains the export directory for an application or DLL. When present, it contains information about the names and addresses of exported functions. It also set the *MEM_READ* and *INITIALIZED_DATA* flags of *Characteristics* field.

4.1.2.5 .rsrc

The *.rsrc* is a resource section, which contains resource information of a PE file. In a Graphical User Interface (GUI) application and many other cases it has *icons* and *images* that are the part of the PE file resources.

4.1.2.6 .bss

This section has the uninitialized data for the application in free format. It has read and write access permission and respective flags (IMAGE_SCN_MEM_READ and IMAGE_SCN_MEM_READ) are set for the same.

Apart from these common section names, there are many others names which are used and accepted among developers and recognized by many tools & applications. But it must be understood that these *section names* are just used by convention and are not mandatory or bound to any PE specification rules.

This section discussed about PE file, *section table*, *section header* and listed few of the frequently used *section name*. The frequently used *section name* has an explicit nature of data which are also discussed in this section. In the following section, motivations for using *section name* as binary features are explained and the process and method for creating feature set based on *section name* are also explained. At the end of the following section, an example feature set is shown which mimic the proposed binary *section name based feature set*.

4.2 Section Name as features

The deceptive nature of *section name* and its importance as features are explained in the previous section. This section explains the process and method of creating feature set based on the *section name*. Creating and testing a feature set based on *section name* as boolean features are novel in terms that in almost all of earlier works it has been used as dependent feature set. The *section names* or other related features are always used with the combination of other features in a feature set. Despite wide scope of possible malicious activities the use of *section name* as the feature is limited. This limit the scopes and possibilities of *section name* as features. This work has explored the potential of *section name* as boolean features and registered the possible outcome of such features in the building of machine learning based malware detector.

4.2.1 Motivations

The *section name* is stored in the *name* field of the *section header* which is an 8 bytes field and can be extended by using *string* table. The *name* field of a section header serves as an unnoticed memory space to the attacker which can be exploited in many ways. The attackers disturbed the normality of *name* field for hiding their malicious modification in the PE file by using various off-shelf tools. The observations on these facts motivate to carry out this work which utilizes *section name* as the binary feature for building malware detector by using machine learning algorithms. Considering the *section name* to use as the binary features for training machine learning algorithms is based on various comparative statistical data and observing the difference in *section name* between malicious and benign sample. Some of earlier works have also used *section name* as feature with the combination of other features (Perdisci et al., 2008; Yonts, 2012; David et al., 2016). On the basis of standard and non-standard name, *sections name* has classified as *suspicious* and *non-suspicious* respectively, then only the *total number* of suspicious sections name are used as an integer feature to classify packed and non-packed PE files (Perdisci et al., 2008). The comparative analysis about *NumberOfSections* in malware and benign shows that nearly 50% of malware sample have less than or equal to 3 sections whereas only 25% of benign files fall in this range (Yonts, 2012). The presence of section *without name* or if the name has *other than*

alphabetic character is seen as indication of malicious PE file (David et al., 2016). Further, all the points which are motivation for this work are listed and summarized.

- Each section name has 8 byte space, a PE file can have many sections, some file type can also use *string table* to store longer *section name*, summing all this provide ample space which can be used by malware without having any effect on the program normal execution.

- Different compilers and packers use the varying naming convention for section name which results in large number of *standard* and *non-standard* section name. The lack of any common standard for naming *section name* opens scope for any values in those *name* field of each section header.

- *Section name* works as the label for human only because linker and loader do not use it for any purpose which opens it for use and misuse. Any contents can be stored in the *name* field in place of *section name* and the work of loader and linker will be unaffected.

- There is a large difference between a number of sections in malware and benign files (Yonts, 2012).

- Use of *non-standard*, *blank* or *name with non-alphabetic character* as section name is indicator of malicious activities (Perdisci et al., 2008; David et al., 2016).

With the reference of aforementioned works, motivational points and the deceptive nature of the *section name* motivate this work to create and verify the potential of feature set based on the *section name*. In the following section 4.2.2, process and method of creating the binary feature set by using *section name* as the feature is explained in details.

4.2.2 Process and Method

This section explains the process and method used for creating the feature set based on *section name* as the boolean feature. For all the available sections in a PE file the *name field* of all the respective *section header* holds the *section name*. As explained in the previous chapter 3, feature extraction and creating feature set is third step after data

collection and pre-processing, both of which is explained further in section 4.3. This section explained the feature set generation process assuming a well-processed dataset having raw malware and benign sample. The details of *dataset* preparation, experimental system and machine learning algorithms training and testing are also presented further in section 4.3.

As explained in earlier sections, every section present in a PE file has a respective *section header* which is stored as the array of structure as *section table*. Each *section header* has a set of fields including *Name* field which store name of the section. Count of total sections present in a file can be known by accessing *NumberOfSections* field from *File header*. Each field of every individual *section header* can be accessed by knowing the total sections and then looping through the *section table*. After looping through every *section header* of the PE file value stored in the *Name* field are extracted. Each extracted *name* field value is processed and only those values are kept which satisfy our conditions to be a feature, for example, alphabetic versus non-alphabetic character, maximum size etc.

Algorithm 2 Generate SectionsName based feature set

procedure GENERATESECTIONSFEATURESET(Ψ, Ω)

▷ Ψ : Malicious PE files

▷ Ω : Benign PE files

$features[...] \leftarrow$ "$sectionName$" ▷ To store selected section name as feature.

$sectionsCount[...][...] \leftarrow$ "$sectionName$", 0 ▷ Array to store section name and frequency

▷ Multi-dimension array to store feature set.

$featureSet[count(\Psi) + count(\Omega)][len(features) + 1] \leftarrow 0$

▷ Extract all section names from each sample (malware and benign)

for $\kappa \in \Psi + \Omega$ **do**

$names[] \leftarrow extractSectionsName(\kappa)$

$sectionsCount[][] \leftarrow StoreAndUpdate(names)$

▷ Select top N section name as feature based on count

$features[] \leftarrow SelectTopSection(sectionsCount[][])$

function UPDATEFEATURESET($names, features, class$)

$featuresValues[len(features) + 1] \leftarrow 0$

for $name \in features$ **do**

if $name \in names$ **then** $featuresValues \leftarrow append(1)$

else $featuresValues \leftarrow append(0)$

$featuresValues \leftarrow append(class)$

return featuresValues[]

for $\kappa \in \Psi$ **do** ▷ Create features value for Malicious PE files

$Class \leftarrow 0$

$names[] \leftarrow extractSectionsName(\kappa)$

$featureSet \leftarrow$ UPDATEFEATURESET($names, features, class$)

for $\kappa \in \Omega$ **do** ▷ Create features value for Benign PE files

$Class \leftarrow 1$

$names[] \leftarrow extractSectionsName(\kappa)$

$featureSet \leftarrow$ UPDATEFEATURESET($names, features, class$)

return($featureSet$)

Algorithm 2 summarizes the overall process of feature set generation from raw malware and benign sample. The Ψ and Ω is the set of malware and benign PE files which are input to the algorithm and the proposed feature set (indicated as variable *featureSet*) is the output of the algorithm. Variable *features* and *sectionCount* is used to store *section name* and *section name with their frequency* respectively. Line 4 in Algorithm 2 initialized the total rows and columns of the proposed feature set. Total rows will be equal to the sum of total malware (Ψ) and total benign (Ω) sample in the dataset. Total columns will be one more than the size of the *feature* array, the one extra column will have the class label of each sample. By looping through the dataset, all the *section name* will be extracted and will be stored with their frequency in the respective variable. The final features will be selected on the basis of frequency ranking and top features will be selected to create the feature set.

Further the algorithm will update the feature set by using the function *UpdateFeatureSet()* and looping through all the sample from both malware and benign class. *UpdateFeatureSet(names,features,class)* function takes *extracted section names* from each sample, the *features* list and *class label* as input parameter. All the *section names* in parameter *features* which is also in the parameter *names* will be assigned value $'1'$ and rest will be assigned $'0'$. By this process, a set of values (1 and 0) for each sample will be created which constitute a single row in the feature set after appending the class label. Every step of Algorithm 2 is supplied with appropriate comment for better understanding. The generated feature set will be passed to various machine learning algorithms for training and testing and their performance result is presented and explained in section 4.3.

With the aforementioned method, a global feature list is created by extracting a total of 464 unique *section names* from each sample of the dataset. As explained earlier the proposed feature set is created by extracting *sections name* from each sample and comparing them with the global feature list. The different subsets of the feature set are also created on the basis of *information gain* feature selection method. Table 4.2 lists out top 20 selected *section name* along with their *information gain* score and their frequency in malware and benign sample. The details about the various subset of feature set and experiments with results are presented and explained in section 4.3.

This section explained the process and method of feature set generation from the

Table 4.2: Top 20 features with score and frequency

Top 20		Frequency	
Features	Score	Malware	Benign
RELOC	0.3065	1001	2109
RSRC	0.0905	2033	2452
TEXT	0.0823	2138	2240
IDATA	0.0808	499	706
BSS	0.0603	460	43
DATA	0.0597	2575	2071
PDATA	0.0515	116	11
RDATA	0.0499	1676	969
UPX	0.0243	496	22
CRT	0.0151	119	32
TLS	0.0149	322	73
ITEXT	0.0116	54	7
CODE	0.011	326	27
NDATA	0.01	8	20
EDATA	0.007	218	29
ADATA	0.0063	18	**0**
OTI	0.0044	46	**0**
ORPC	0.0041	2	27
TEXTF	0.0038	9	**0**
XURI	0.0032	30	**0**

Table 4.3: An example for feature set based on section name as features

RELOC	RSRC	TEXT	IDATA	BSS	DATA	PDATA	RDATA	UPX	CRT	Class
1	0	0	1	0	1	0	1	0	1	Benign
1	0	1	1	1	1	1	0	0	1	Benign
0	1	0	1	0	0	1	0	0	1	Benign
0	0	1	1	0	0	0	0	0	0	Malware
0	0	1	0	1	0	0	1	1	1	Malware
0	1	1	1	0	1	1	0	1	0	Benign
0	1	0	0	1	0	0	1	1	1	Malware
1	1	0	1	1	0	1	0	1	0	Benign

raw samples. In the following section 4.2.3, an example feature set which symbolized the original feature set.

4.2.3 An Example of the Feature set

In this section, a representative feature set is presented by using selected top 10 *section name* as features and features values are filled by mock data. Table 4.3 visualizes the essence of feature set created during the experiment. Top 10 *section names* (refer Table 4.2) are used as features and their presence and absence in each sample is represented as '1' and '0' respectively. The last column of the table indicates the class label of each sample they belong to malware or benign. This labeling of the sample is must for supervised ML algorithms. These all information can be stored in many ways. In this work, the feature set is stored as a Comma Separated Value (CSV) file and later read and process to pass to ML algorithms.

4.3 Discriminative capacity of Section Name as features

This section presents the results of various experiments which were performed to test the discriminative capacity of *section name* as features. The details about the section, section table, section header, rational behind considering section name as feature, and method for feature set generation is presented and explained in earlier sections. This section provides details of steps taken for dataset preparation and about the experimental system along with the experiments and their respective results.

4.3.1 Dataset

This work collected 2724 malware and 2529 benign sample for conducting various experiments to test the proposed *section-name feature set*. The malware sample were collected from *virusshare*[6] and benign sample were collected from system and program folder of freshly installed Windows OS. To make collected sample suitable for further processing such as feature extraction and feature set generation various pre-processing steps were taken which are explain in following section.

4.3.1.1 Pre-processing

This work is based on *section name* of PE files so the dataset only can has PE files. So during pre-processing step, non-PE files and duplicate samples were removed from the dataset. This work also removed files which were only supported by *DOS* and hence do not have PE signature which indicates the presence of NT headers. The final dataset has 2724 malware and 2501 benign sample.

The packer change the section name of PE files so dataset was checked for packed and unpacked files. The *packer* information of each sample was checked using **PEiD**[7] signature database (a total of 1660 signature) with **Yara**[8] tool. Table 4.4 shows the count of the packed and unpacked sample of malware and benign sample.

Table 4.4: Packed and unpacked sample in dataset

PackerInfo	Malware	Benign	Total
Packed	1039	353	**1392**
Unpacked	1685	2148	3833
Total	**2724**	**2501**	**5225**

4.3.1.2 Class Labelling

After cleaning the dataset during pre-processing step, the dataset is passed for class labelling. For supervised learning each sample of dataset must be labelled with proper class label. This work adopted the same technique to label the sample as explained

[6] `https://virusshare.com/`,[Last Accessed:10 January 2017]

[7] PEiD detects most common packers, cryptors and compilers for PE files.

[8] `http://virustotal.github.io/yara/`,[Last Accessed:10 January 2017]

in Chapter 3. Each of the sample was labelled by online anti-virus scanning service *virustotal*[9] which provides a API service to scan sample with multiple parallel running Anti-virus engines.

4.3.1.3 Feature extraction

This work has proposed the *section name* of PE files as feature so cleaned and labelled dataset was passed for feature extraction. The Features i.e. *section names* from each PE file were extracted by using *pefile* module by implementing a Python script. All the feature processing task were also carried out by different Python scripts. The details explanation of feature extraction and feature set generation process is explained in section 4.2.2.

This section explained about the dataset, pre-processing steps, labelling and feature extraction for the *section-name based feature set*. The following section 4.3.2 explains about the experimental system used for performing the various experiments to test the performance of section-name based feature set.

4.3.2 Experimental System

The proposed feature set is created by using static analysis. This work has used Ubuntu OS running on Intel(R) Core (TM) 2 Duo CPU $E7400@2.8-GHz$ processor with 4GB of primary and 500GB of secondary memory. The Python programming Language and modules were used for scripting and for the experiments. The *Pefile* was used for PE processing and *scikit-learn*Pedregosa et al. (2011) was used for performing machine learning task.

This section presented the details of the experimental system used for carry out various experiments to verify the discriminative potential of the *section name based feature set*. The following section 4.3.3 presents and discusses the results of various experiments.

[9] `https://virustotal.com/`,[Last Accessed:10 January 2017]

4.3.3 Results

In further sections, results of different experiments are presented with explanation and discussion. With feature selection method different sub-sets of original feature set were created and which are used in different experiments. Six machine learning algorithms were selected which are Logistic Regression (LR), Linear Discriminant Analysis (LDA), Random Forest (RF), Decision Tree (DT), Gaussian Naive Bayes (NB), and k-Nearest Neighbors (kNN). The four performance metrics Accuracy, Precision, Recall and F1-score were used for compare the ML algorithms. The experiments were design on the basis of feature selection such as performance without and with feature selection. In following section 4.3.3.1 results of experiments without feature selection is presented and discussed.

4.3.3.1 Classifiers performance without feature selection

In this section, all the six classifiers performance is evaluated with all the 464 features. The evaluation result is presented and compared with Accuracy, Precision, Recall and F1-score. Table 4.5 list out the values of these metrics for all the six classifiers. From Table 4.5, it can be observe that Random forest has an accuracy of 87.12% which is highest among all other classifiers.

Table 4.5: Classifiers result on various performance metric

Classifiers	Accuracy	Precision	Recall	F1-score
LR	83.67	0.84	0.84	0.84
LDA	83.99	0.84	0.84	0.84
RF	**87.12**	0.87	0.87	0.87
DT	86.86	0.87	0.87	0.87
NB	52.42	0.75	0.52	0.42
kNN	86.16	0.86	0.86	0.86

Figure 4.2 visualize the ROC of all classifiers with all features. Again, from Figure 4.2, it can be observe that Random forest has highest AUC value (0.93) among all classifiers.

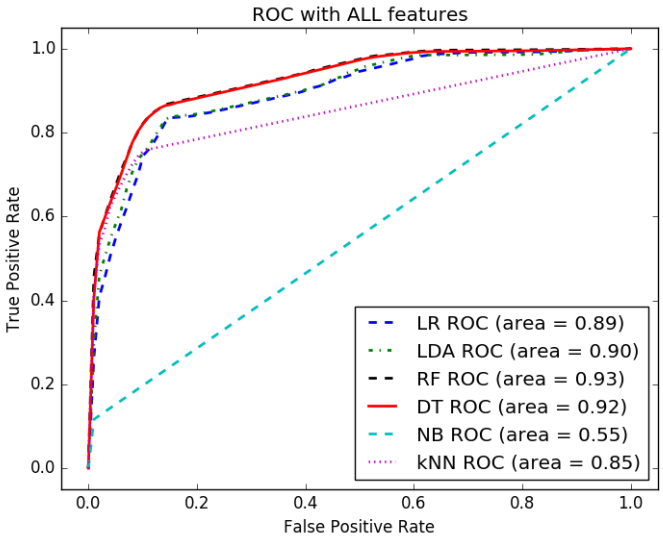


Figure 4.2: ROC for all classifiers with ALL features

This section presented the result of selected classifiers on various metrics without any feature selection. The following section 4.3.3.2 presents and discusses the results of experiments with feature selection.

4.3.3.2 Classifiers performance with feature selection

The feature selection is a method for constructing and selecting subsets of features that are useful to build a good predictor. The feature selection benefits in many ways, such as data visualization, in tackling *curse of dimensionality*, reducing storage requirements, training and utilization time Guyon and Elisseeff (2003).

This work has used wrapper methods with *Decision Tree* for feature selection and used *entropy* value for ranking the feature. Two different feature sets were created with the importances score of features, one having all features with *non-zero* scores and other having top 20 features. The non-zero feature set has 158 features, which is nearly one-third of all the features. Table 4.6 presents the top 20 selected features (i.e. section name) with their score and frequency in malware and benign sample.

Table 4.6: Top 20 features with score and frequency

Top 20		Frequency	
Features	Score	Malware	Benign
RELOC	0.3065	1001	2109
RSRC	0.0905	2033	2452
TEXT	0.0823	2138	2240
IDATA	0.0808	499	706
BSS	0.0603	460	43
DATA	0.0597	2575	2071
PDATA	0.0515	116	11
RDATA	0.0499	1676	969
UPX	0.0243	496	22
CRT	0.0151	119	32
TLS	0.0149	322	73
ITEXT	0.0116	54	7
CODE	0.011	326	27
NDATA	0.01	8	20
EDATA	0.007	218	29
ADATA	0.0063	18	**0**
OTI	0.0044	46	**0**
ORPC	0.0041	2	27
TEXTF	0.0038	9	**0**
XURI	0.0032	30	**0**

Figure 4.3 depicts the ROC of all six classifiers on the feature set with top 20 features. Decision tree and Random forest performance are equal and are the best among all the classifiers.

In Table 4.7, it can be observe that Random forest performance does not change with feature selection whereas all other classifiers performance with only top 20 features get reduced by 1% and performance of Naive Bayes get boosted by nearly 20%.

From Table 4.7, it can be derive that Random forest performance slightly (reduced 1% with top 20 features) changed while kNN performance varies with the different set

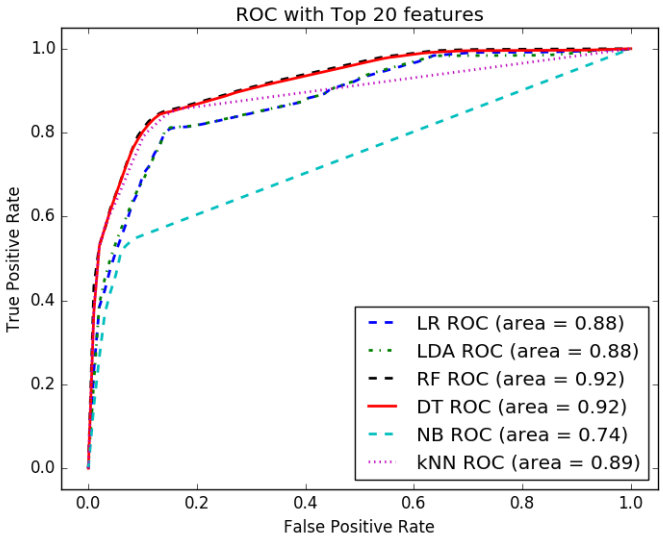


Figure 4.3: ROC for all classifiers with Top20 features

of features. With all non-zero score features, it achieved 90% AUC value which is 5% more than the AUC with all the features.

Table 4.7: AUC of classifiers with set of selected features and 10 folds cross-validation

Classifiers	ALL	Non-Zero	Top20
LR	0.89	0.89	0.88
LDA	0.90	0.89	0.88
RF	**0.93**	**0.93**	**0.92**
DT	0.92	0.92	0.92
NB	**0.55**	0.54	**0.74**
kNN	0.85	0.90	0.89

This section explained about dataset, experimental system and presented results of various experiments which was performed to test the potential of *section name base feature set*. The results of various classifiers were grouped on the basis of with feature selection and without feature selection. The proposed feature set achieved 93% accuracy with just top 20 features and extracting section names for feature set is also simple and computationally cheaper.

The following section 5.5 presents and explains the performance of the proposed weighted permission based feature set. The results of proposed feature set are compared

with binary permission based feature set which are extensively used in the literature. The proposed feature set is prepared from the Android's apk files.

4.4 Summary

This Chapter has proposed a *sections name based feature set* which has a set of selected *sections name* as boolean/binary features. It provides details about *section table*, *section header* and *sections* along with the process of building the boolean *feature set* from the malware and benign PE files. With the help of Algorithm 2 whole steps involved in building *sections name based feature set* is summarized. Various selected machine learning algorithms are trained and tested on proposed *sections name based feature set* and their performance are measured. This chapter also presented the detail and the results of training and testing of various feature set.

CHAPTER 5

Malicious Android Applications Triaging Using Weighted Permission

In the previous two Chapters (Chapter 3 and Chapter 4), two feature sets are proposed for the detection of malicious Portable Executable (PE) files. PE file format is for Windows Operating System (OS). This Chapter and following Chapter 6 proposes feature sets for detection of malware for Android which is a smartphone OS and holds largest share in terms of users.

This Chapter explores the potential of Android's permission as feature for malicious app detection using machine learning algorithms. This thesis work has used permission as *integer* feature than traditional use as *boolean/binary* feature. Each permission has been assigned an *effective weight* and used as feature's value instead of just using binary value for presence and absence as feature's value.

This Chapter explains the details of *permission weighting process* which assigns weight to each of the permission. Later, the weighted permission is used as feature to train various machine learning algorithms to build a classifier to detect malicious and benign Android applications. Using permission weight instead of boolean value as feature improve the classification accuracy.

This Chapter also present and discuss about the proposed tool, *FAMOUS* (**F**orensic **A**nalysis of **MO**bile devices **U**sing **S**coring of application permissions) which is machine learning based triaging tool for forensic analysis. Among six trained classifiers the best performing classifier were selected and used as back-end to build the proposed tool *FAMOUS* which scan Android device and a triage installed apps into suspicious and benign.

This Chapter starts by explaining in detail about Android security and its permis-

sion system. Further, details about Android apps' *manifest* file are discussed and presented. This Chapter provides details about permission extraction, permission weighting process with the help of scoring engine and feature set generation process. The introduction of proposed tool *FAMOUS* is also presented. The details about dataset, experimental system, and results of various experiments for machine learning algorithms performance is presented further in section 5.5.

5.1 Andriod Security

This section provide a brief introduction about Android Security and the following section explain about Android's permission system. The Android platform has been designed with multi-layered security measures such as Linux kernel at OS level, mandatory application *sandboxing* at application level, secure inter-process communication at process level, application signing at developer level and application-defined and user-granted permissions at end-user level [1]. Fig. 5.1 illustrates Android multi-layered security measures. Among all the aforementioned security mechanisms, the effectiveness of application's permission layer outcome is fully dependent upon the end-user which would be treated as either empowering the user or burdening, from different perspectives. This 100% user dependent security check is the weakest link in the Android's security pipeline. To perform different tasks on the device such as accessing Internet, using camera, reading contacts etc., each application has to explicitly display and acquire the required permissions during the installation and user has to decide either granting requested permissions and proceed with installation, or denying it and cancelling the installation. As most of the end-users are not technically aware to make an informed decision, they grant permissions and install application without understanding the malicious intentions of application. This weak spot of Android multi-layered security has been picked by attackers and hence many malicious apps are intruding the end-user devices through various third-party malicious app stores.

This section gave a brief introduction of Android security and in the following section 5.2, a detailed explanation of Android application permission system is provided along with the discussion about the *manifest* file.

[1] https://source.android.com/security/

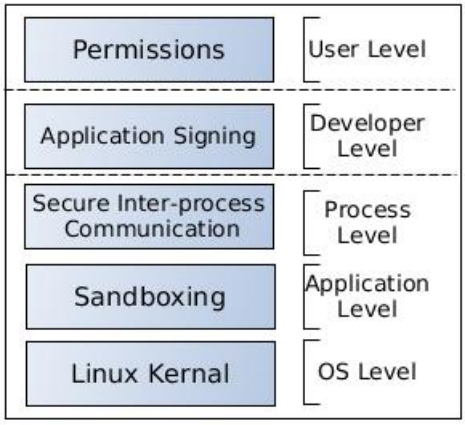


Figure 5.1: Multi-layered security measures of android

5.2 Application's Permission

This section explains about permission system of Android OS which is an user level measure to limit the access of applications. The permissions are declared in *AndroidManifest.xml* file explicitly by all the application developers during packaging of application as *APK*. The detail about *manifest* file is explained in following section. Android uses the *permission system* to provide security to user where every application requires to declare respective permissions to access a service. During installation all permissions required by an application is shown to the user and installation will proceed only if user accepts and allow else installation will be aborted. All the required permissions by the application is listed in a default system file called manifest file named as *AndroidManifest.xml*. All the official Android permissions are categorized into four types: *Normal*, *Dangerous*, *Signature* and *SignatureOrSystem*.Fig. 5.2 shows a part of an example manifest file which is listing six permissions required by the example application.

Permissions requested by the application clearly reveals the nature and behaviours of the application. Studying permissions pattern for malware and benign will be helpful to find methods and threshold to build classification system. In the following section 5.2.1, Android *manifest* is explained with an example. To understand the difference in permission request by malicious and benign few statistical tests were conducted which is explained in further sections.

5.2.1 Manifest file

This section provides detail about Android manifest file which has all the declared permissions require by the application to provide its functionalities. Every application must have an *AndroidManifest.xml* file in its root directory[2]. It presents essential information about the application to the Android system, information the system must has before it can run any of the application's code. In this file permissions which are required by the application and the others relevant details such as information about *intents* and *activities* are declared. Fig. 5.2 shows a part of an example manifest file listing required permissions by the application. It acts as a declaration file and here only the activity which should start first is declared. It also lists the libraries that the application must be linked and all the other components i.e. activities, services, broadcast receivers, and content providers should also declared in the manifest file. Fig. 5.3 shows a part of an example manifest file listing *activity* required by the application.

```
<?xml version="1.0" encoding="utf-8"?>

<manifest android:versionCode="10023" android:versionName="1.0.2"
package="com.wia.ucgepcdvlsl" xmlns:android="http://schemas.android.com/apk/res/android">
...
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
<uses-permission android:name="android.permission.READ_PHONE_STATE"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"></uses-permission>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-permission>
...
</manifest>
```

Figure 5.2: A snapshot of AndroidManifest file showing *uses-permission* elements

```
<activity android:name="com.agmart.in.Login_activity" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.NotAUser" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.Otp_activity" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.Sale_CategoryActivity" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.NewSalePost_activity" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.Buy_CategoryActivity" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.BuyModule.FilterCrops" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.Change_password" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.Change_MobileNumber" android:screenOrientation="portrait"/>
<activity android:name="com.agmart.in.BuyModule.ListingBuy" android:screenOrientation="portrait"/>
```

Figure 5.3: A snapshot of AndroidManifest file showing *activity* elements

This section provided detail about the Android *manifest* file and in the following sections details of various statistical tests is explained. Results of these initial tests

[2]The file name *AndroidManifest.xml* is fixed so must have same name.

motivated this thesis work to use permissions as integer feature by assigning weight according to its frequency in malware and benign class.

5.2.2 Statistical test

This section discusses the statistical tests which were carried out to understand the differences between malicious and benign Android application. The results of the statistical tests are very helpful to understand the relationship between various variables. To understand the differences and similarities between malware and benign, three candidate variables: *filesize*, *total files in apk* and *total permissions* were selected and an overlay histogram of each variable were plotted. The overlay histogram provided visual clues about the variables of both classes. In the following sections details about these tests are explained.

5.2.2.1 File size

File size is the requirement of total bytes on disk to store all contents of a file. File size is very important in study of understanding difference between malware and benign applications. It is common assumptions and understanding that to propagate itself, malware try to be as smaller as possible while benign applications do not impose such restriction on file size. File size of each sample present in dataset were calculated to plot the overlay histogram shown in Fig. 5.4.

Form Fig. 5.4 it can be observed that malware and benign samples have a different file size patterns, malware seems to have smaller in size whereas benign samples are larger in size. The mean of sample size for benign category was observed as 6.37 with a Standard Deviation (SD) of 8.52 whereas mean of file-size for malware was recorded as 1.31 with SD 2.02.

With these mean and SD values, it can be concluded that benign apk has an average size of $7MB$ where as malware is much smaller in size and has average size of $2MB$. Larger variations in benign also suggest that benign sample file sizes spans across a larger spectrum (i.e. $0-20MB$ in Fig. 5.4) where as malware file size does not vary much and $8MB$ size seems to be largest size.

This section explained about file size and presented differences observed in file size among malware and benign Android applications. The following section 5.2.2.2,

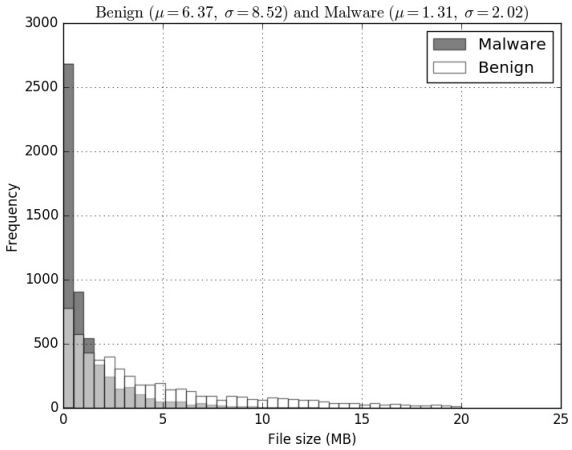


Figure 5.4: Histogram of file size for malware and benign samples

explains about *total number of files* and its pattern among malware and benign samples.

5.2.2.2 Total files

Android's apk is a compressed (very similar to zip compression) form of many files present in an Android application which includes all resource files under *res* directory, *dex* files, *AndroidManifest* and many other required files. Every file present in the *APK* was considered as a candidate variable and counted for all samples available for the experiment. Each *APK* was decompressed and all files inside the *APK* were counted after which file count value is plotted as an overlay histogram as shown in Fig. 5.5. An overview of the histogram in Fig. 5.5, reveals that malware uses very less number of files whereas benign samples have more number of files. The mean of *number of files* for benign category was observed as 342 whereas the mean of *number of files* for malware category was observed only as 90. Number of files present in benign and malware category correlated with the size of file i.e. malware have fewer number of files resulted in small size where as benign have more number of files resulted in large size.

This section explained about *total files present in the APK* and its pattern among malicious and benign Android application. The following section 5.2.2.3, presents study about permissions count among malware and benign sample.

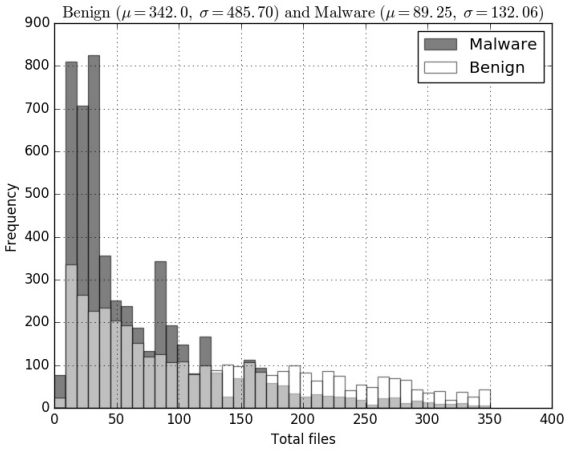


Figure 5.5: Histogram of file count for malware and benign samples

5.2.2.3 Total Permissions

As explained in earlier section 5.2, Android uses a permission system to provide a user level security and the permissions required by the application is listed in the default system file called *AndroidManifest.xml* file. All the permissions from each of the *APK* were extracted and counted. The total permissions of each *APK* in malware and benign category were used to create an overlay histogram shown in Fig. 5.6. Fig. 5.6 shows the comparison of permission count for malware and benign category. The mean of permissions count for benign category was observed as 6.73 with a SD of 5.66 whereas mean of permission count for malware category was observed as 12.43 with a SD of 7.96. It can be inferred that malware permission count is double than that of benign category. The graph vividly shows that permission count for benign is much higher in range of $0-15$ and very low towards higher bin sizes. Malware samples show its presence at lower bin sizes but have a significant high presence at higher bin sizes, where benign presence is negligible.

This section explained about permission and shown the difference in *total number of permissions* among malware and benign application. Motivated by the different permission patterns among malware and benign sample, this work proposed a weighted permissions based feature set for malware detection. In the following section 5.3, details about permission extractor, scoring engine and the process of feature set generation is explained.

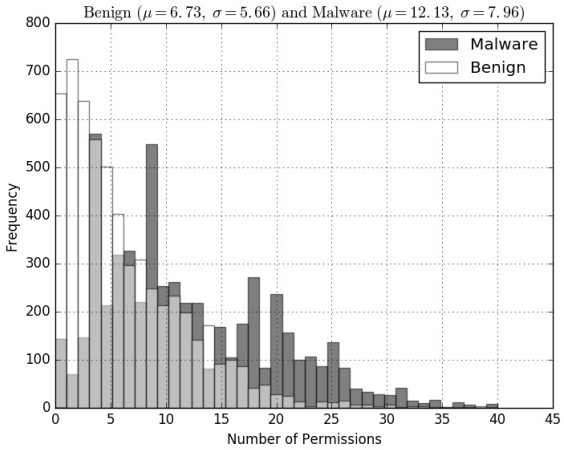


Figure 5.6: Histogram of total permissions requested by malware and benign samples

5.3 Weighted Permission as Feature set

This section explains about the process of creating the proposed feature set from *raw* malware and benign Android applications. The process of creating a feature set using *weighted permission* as feature has three main components: (1) Permission Extractor, (2) Scoring Engine and (3) Feature Set Generator. *Permission Extractor* extracts all requested permissions present in an *APK*, *scoring engine* works in back end and update the malicious and benign score of each permission and *feature set generator* use scoring engine's output and gives score to each permission present in a given android application. Fig. 5.7 depicts all three components and their interaction and control flow.

The system takes malware and benign Android samples as input and produce a feature set in a multi-dimensional vector representation. Each column represents a feature except last that store class label and each row represents a vector representation of permissions present in sample.

In the following section 5.3.1, process of permission extraction is explained, in section 5.3.2, details about scoring engine is presented and in section 5.3.3, process of feature set generation is explained.

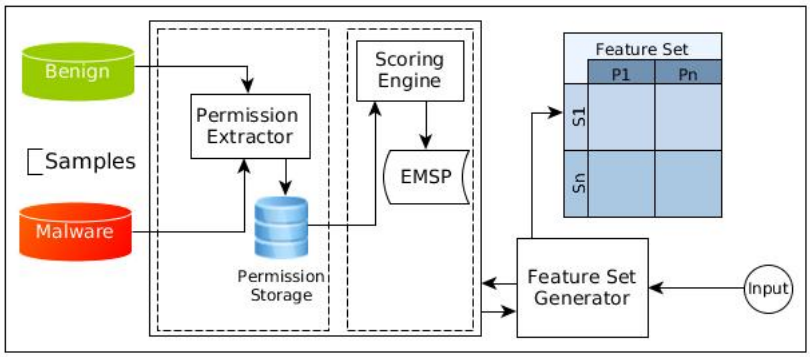


Figure 5.7: Feature extraction and scoring system

5.3.1 Permission Extractor

Android application packed as an *apk* file has all required files to run the application. Among other files, every apk must have a manifest file, which is a binary *XML* file and must be named as *"AndroidManifest.xml"* as per the Android developer guidelines [3]. Among other essential information, it declares which permissions the application must have in order to access protected parts of the API and interact with other resources. These permissions are declared as *uses-permission* and *uses-feature* element in manifest file. Fig. 5.2 shows a list of permissions requested by a sample application.

Permission extractor module extracts all permissions declared in manifest file and will write them to an external storage. In this work, *Comma Separated Value (CSV)* file format is used to store all extracted permissions and pass these to further module for processing. Permission extractor do not differentiate between Android's standard permissions and other user defined custom permissions, so it extracts all permissions requested by the Android application. Android standard permissions are declared as *android.permission.PERMISSION-NAME* while other custom permissions can have any structure but mostly follow the package name format such as *com.android*, *com.motorola* and *org., hr.*.

```
1 <uses-permission android:name="android.permission.BLUETOOTH" />
```

Program 5.1: Manifest file showing BLUETOOTH permission

For example in Program 5.1, the permission in an application tell that it can control *Bluetooth* which includes sending and receiving data from nearby devices.

[3] https://developer.android.com/guide/topics/manifest/manifest-intro.html

This section explained about *permission extractor* which extracts all permissions from the application and is first and must component of the system. The output of permission extractor module is served as input to the *scoring engine* which is explained in the following section.

5.3.2 Scoring Engine

This section explains working of scoring engine which provides an effective score to each of the permission. The Scoring engine takes extracted permissions file as input, and in first iteration it extracts only standard Android's permissions. After cleaning the permission format, it counts the occurrences of each permission in benign and malware category. The Scoring engine calculates values for different variables (B, M, PuB and PuM) that is used to find the BSP, MSP and $ESMP$. B and M represent the total sample used from benign and malware class respectively. Permission used in Benign (PuB) and Permission used in Malware (PuM) variables are used to represent the frequency of each permission in benign and malware class respectively. After calculating aforementioned preliminary variables, Benign Score of Permission (BSP) and Malicious Score of Permission (MSP) for each permission is calculated by using Eq.5.1 and Eq.5.2 respectively. In Eq. 5.1, PuB is the total number of times a permission is used in benign group, and B is total number of benign samples.

$$BSP = PuB/B \tag{5.1}$$

In Eq.5.2, PuM is the total number of times a permission is used in all malware sample, and M is total number of malware sample.

$$MSP = PuM/M \tag{5.2}$$

The Effective Maliciousness Score of Permission ($EMSP$) is the subtraction of BSP value from MSP that will normalize the permission use in benign and will have value which will represent the *maliciousness weight* of the permission. After calculating MSP Eq.5.2 and BSP Eq.5.1 of a permission, the Effective Maliciousness Score of Permission $EMSP$ can be calculated by using Eq. 5.3.

$$EMSP = MSP - BSP \tag{5.3}$$

The *MSP* (Eq.5.2), *BSP* (Eq.5.1) and *EMSP* (Eq.5.3) scores were calculated using the *Score Engine* for all the permissions (includes uses-permissions and uses-features) on the processed dataset. Table 5.1 shows top 25 permissions from each group and their *MSP*, *BSP* and *EMSP* score.

To understand the permission's scoring process, an example is presented and explained. Suppose, sample present in example (Ref. Fig. 5.2) is a malware, so *MSP* for INTERNET permission can be calculated as $(5979/5553 = 1.07671)$ by using Eq.5.2, where *PuM* for INTERNET is 5979 (Ref. Table 5.1) and value of *M* i.e. total malware sample is 5553. Similarly, if sample (Ref. Fig. 5.2) is benign, then *BSP* for for INTERNET permission can be calculated as $(5536/5818 = 0.9515)$ by using Eq.5.1, where *PuB* for INTERNET is 5536 (Ref. Table 5.1) and value of *B* i.e. total benign sample is 5818. Once *MSP* and *BSP* is calculated for INTERNET permission, the *EMSP* can be calculated as $(1.07671 - 0.9515 = 0.1251)$ by using Eq. 5.3.

Maliciousness Score (MS) of an Android's application is sum of *EMSP* value of each permission present in the given application. The *MS* of an app will be calculated by using Eq. 5.4, where *EMSP* is a pre-calculated value for each standard Android's permission based on dataset, n is total permission present in the given Android application and p_i is individual *EMSP* of each present permission.

$$MS = \sum_{i=1}^{n} EMSP(p_i) \tag{5.4}$$

Continuing the earlier example for sample apk (Ref. Fig. 5.2), the total maliciousness score can be calculated as $(0.1251 + 0.6048 + (-0.0222) + 0.4951 + 0.1694 + 0.1646)$ by using Eq.5.4, which sums *EMSP* value of all 6 permissions (refer Table 5.1 for all other permission's *EMSP* values).

EMSP score can be used in two ways, first to calculate total maliciousness score of a sample and second to create a feature set to train machine learning algorithm (as explained in section 5.3.3).

MS can be used to build a light weight classifier, for which it has to be compared with a threshold value to decide class label of a given test apk. Threshold finding can be achieved by different ways, one of the methods can be similar to (Talha et al., 2015)'s work which used Logistic regression to find a threshold. The proposed work is not using *MS* scoring to build classifier, hence no further investigation in this direction was

Table 5.1: PuB, PuM, BSP, MSP and EMSP values of top 25 permissions (sorted based on malware)

Permission	PuM	PuB	BSP	MSP	EMSP
INTERNET	5979	5536	0.9515	1.0767	0.1252
READ_PHONE_STATE	5463	2205	0.379	0.9838	0.6048
ACCESS_NETWORK_STATE	4440	4781	0.8218	0.7996	-0.0222
WRITE_EXTERNAL_STORAGE	4146	3386	0.582	0.7466	0.1646
SEND_SMS	3058	274	0.0471	0.5507	0.5036
RECEIVE_BOOT_COMPLETED	2755	6	0.001	0.4961	0.4951
ACCESS_WIFI_STATE	2563	1700	0.2922	0.4616	0.1694
WAKE_LOCK	2193	1780	0.3059	0.3949	0.089
RECEIVE_SMS	2151	169	0.029	0.3874	0.3584
READ_SMS	2098	91	0.0156	0.3778	0.3622
ACCESS_COARSE_LOCATION	1920	1591	0.2735	0.3458	0.0723
ACCESS_FINE_LOCATION	1780	1684	0.2894	0.3205	0.0311
VIBRATE	1659	1	0.0002	0.2988	0.2986
READ_CONTACTS	1334	443	0.0761	0.2402	0.1641
WRITE_SMS	1246	44	0.0076	0.2244	0.2168
CHANGE_WIFI_STATE	1001	201	0.0345	0.1803	0.1458
INSTALL_PACKAGES	829	19	0.0033	0.1493	0.146
GET_TASKS	821	417	0.0717	0.1478	0.0761
RESTART_PACKAGES	756	82	0.0141	0.1361	0.122
CALL_PHONE	742	664	0.1141	0.1336	0.0195
WRITE_SETTINGS	686	208	0.0358	0.1235	0.0877
ACCESS_LOCATION _EXTRA_COMMANDS	618	270	0.0464	0.1113	0.0649
WRITE_APN_SETTINGS	564	8	0.0014	0.1016	0.1002
WRITE_CONTACTS	546	237	0.0407	0.0983	0.0576
SET_WALLPAPER	530	240	0.0413	0.0954	0.0541

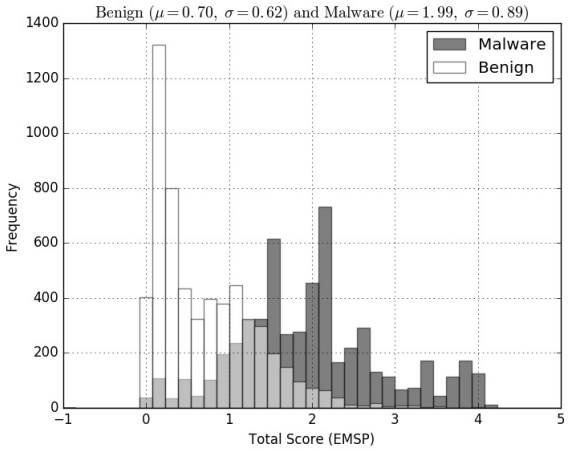


Figure 5.8: Histogram of total EMSP for malware and benign apks

done.

The calculated *EMSP* for each permission is passed to the next component *Feature Set Generator* that create a vector representation for all samples based on permissions present in each sample and their respective *EMSP* score. In the following section 5.3.2.1, the study about distribution of *EMSP* score is explained.

5.3.2.1 Score Distribution

The value of *EMSP* and *BSP & MSP* were calculated and their distribution were observed to know the potential of *permission weighting*. The effect of these scores were study for malware and benign category by considering histogram as the visualization tool. Individual permission's score for each sample was calculated as explained in section 5.3.2, total EMSP and BSP & MSP were calculated according to Eq. 5.4. Total score for both (EMSP and BSP & MSP) were plotted for malware and benign category. Fig. 5.8 shows histogram based on EMSP based scoring and Fig. 5.9 shows histogram based on BSP and MSP based scoring.

In Fig. 5.8, it is visible that malware has high clustering at higher value bins whereas benign samples values are clustered towards lower value bins. The benign samples have a mean value 0.70 where as malware have mean value 1.99 and bin size 2 can be seen as threshold which clearly separate benign and malware values.

This section explained about the working of *scoring engine* and presented various equations which were used to calculate value for different variables. This section also

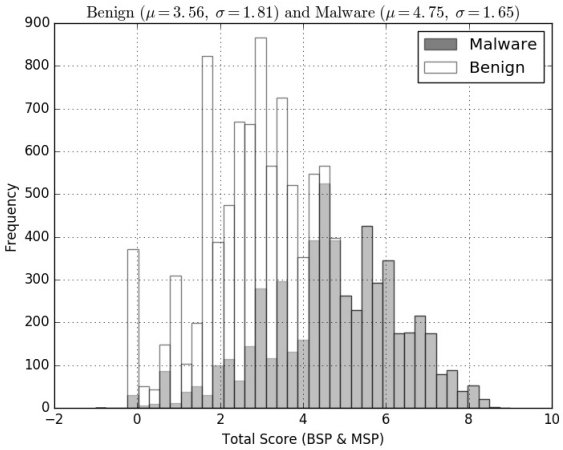


Figure 5.9: Histogram of total BSP and MSP for malware and benign

shown the distribution of calculated scores for malware and benign samples. In the following section 5.3.3, the process of feature set generation is explained which takes extracted permissions and their scores as input and produce the proposed feature set.

5.3.3 Feature Set Generator

Feature Set Generator module is dependent on previous two modules i.e. *Permission Extractor* and *Scoring Engine*. It takes *apk* files as input and call permission extractor to extract permissions and uses scoring engine output to assign value to each permission. Fig. 5.10 shows a row from feature set based on permissions presents in sample *APK* file (Ref. Fig. 5.2). It can be observe that all the available permissions have got value same as their respective EMSP value, whereas all other permissions (features) get assigned zero as value. It is in contrast with earlier permission based feature set that were considered as boolean feature. In such feature set, present permissions of a manifest get 1 and all others were assigned 0 as values. Eq. 5.5 is used to perform the mapping of score to the feature.

$$I(x,p) = \begin{cases} Score(p), & \text{(if the application x have permission p)} \\ 0, & Otherwise \end{cases} \tag{5.5}$$

The process of dataset to feature set generation is illustrated in Algorithm 3 and it consolidate all actions performed at three aforementioned steps i.e. permission extraction, scoring and feature set generation. In Algorithm 3, Ψ and Ω is collection of

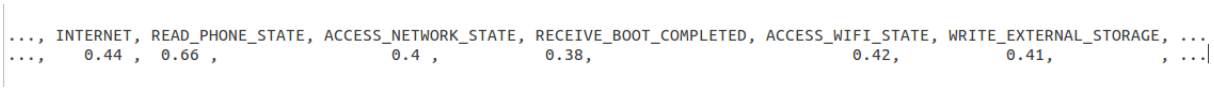


Figure 5.10: A snapshot of feature set generated based on EMSP and permissions

benign and malware sample respectively and it is supplied as input to the *GenerateFeatureSet()*. The *FetchPerm()* and *UpdatePermCount()* are methods used for extracting permissions from sample and updating each permission count respectively. The χ is feature set's data and λ is set for respective label of each instance in χ, χ and λ is output of *GenerateFeatureSet()*.

This section explained the process of feature set generation and presented the Algorithm 3 which summarized all the steps taken to create the proposed feature set from raw malware and benign Android applications. The following section 5.4, presents and explains the proposed forensic tool built by selecting best performing machine learning algorithm on the proposed weighted feature set.

5.4 FAMOUS

FAMOUS (**F**orensic **A**nalysis of **MO**bile devices **U**sing **S**coring of application permissions) is a forensic analysis tool built to triage Android applications and to assist analyst in selection of applications for further in-depth or manual analysis. Screenshots of main window and result window of *FAMOUS* are as illustrated in Fig. 5.11 and Fig. 5.12 respectively. The motivation behind *FAMOUS* is to overcome the limitations of signature-based triaging forensic tool. The main functions of *FAMOUS* is to assign a proper class label (among benign and malware / suspicious) to every selected Android applications by underlying classification engine. Each classification engine is built by training and testing different machine learning algorithms on proposed *permission's score* based feature set that are extracted from a large dataset. Currently, in the proof-of-concept implementation it has only best performing classifier but it can be easily extended with more classifiers.

In the further section architecture of *FAMOUS* and its components are explain in detail.

Algorithm 3 The algorithm for the feature set generation

1: **procedure** GENERATEFEATURESET(Ψ, Ω)
2: $\rho_B[perm, count] \leftarrow permString, 0$
3: $\rho_M[perm, count] \leftarrow permString, 0$
4: $\alpha \leftarrow count(\Psi)$
5: $\beta \leftarrow count(\Omega)$
6: **for** $\kappa \in \Psi$ **do**
7: $P \leftarrow FetchPerm(\kappa)$
8: $PuB \leftarrow UpdatePermCount(\rho_B[P, count])$
9: **for** $\kappa \in \Omega$ **do**
10: $P \leftarrow FetchPerm(\kappa)$
11: $PuM \leftarrow UpdatePermCount(\rho_M[P, count])$
12: **for** $\eta \in \rho$ **do**
13: $BSP[\eta[permString]] \leftarrow \eta[PuB]/\alpha$
14: $MSP[\eta[permString]] \leftarrow \eta[PuM]/\beta$
15: $EMSP[\eta[permString]] \leftarrow MSP - BSP$
16: $Initilaize(\chi[\alpha+\beta][count[\rho_{B+M}[perm]]], 0)$
17: **for** $\mu \in \{\Psi, \Omega\}$ **do**
18: $\gamma \leftarrow FetchPerm(\mu)$
19: $\lambda \leftarrow updateLabel(\gamma)$
20: **if** $permString \in \gamma$ **then**
21: $\chi[permString] \leftarrow Fetch(EMSP[permString])$
22: **return**(χ, λ)

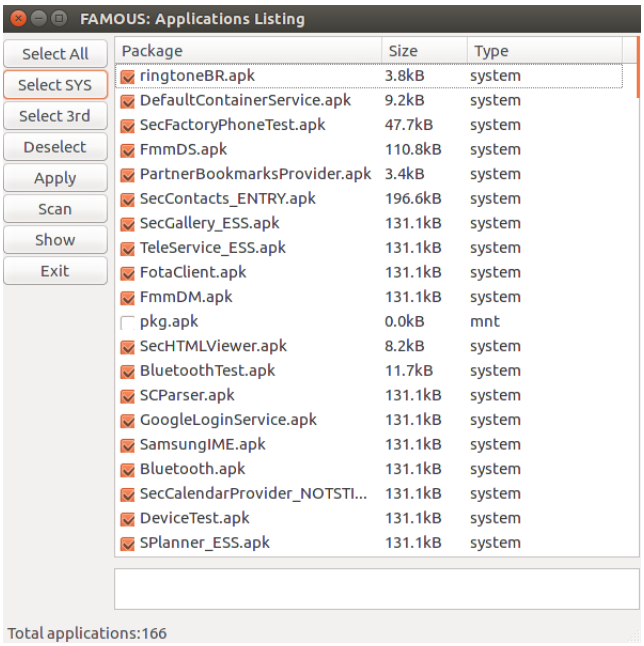


Figure 5.11: Main Window of FAMOUS: Listing all applications of the attached device

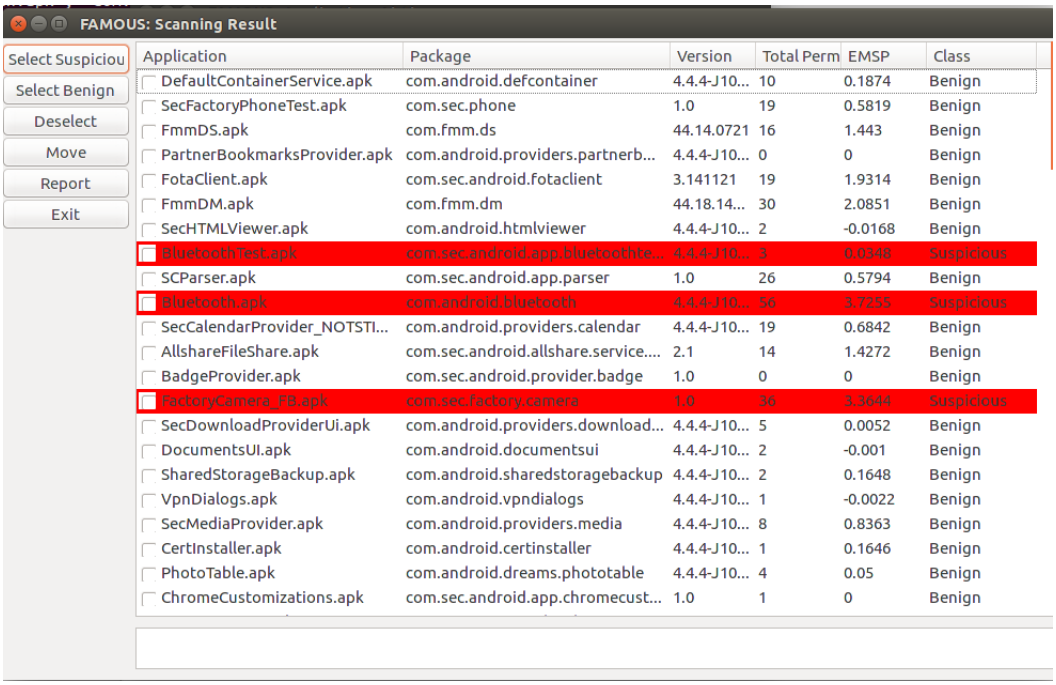


Figure 5.12: Scan Result Window of FAMOUS: Showing predicted class label of all the selected applications

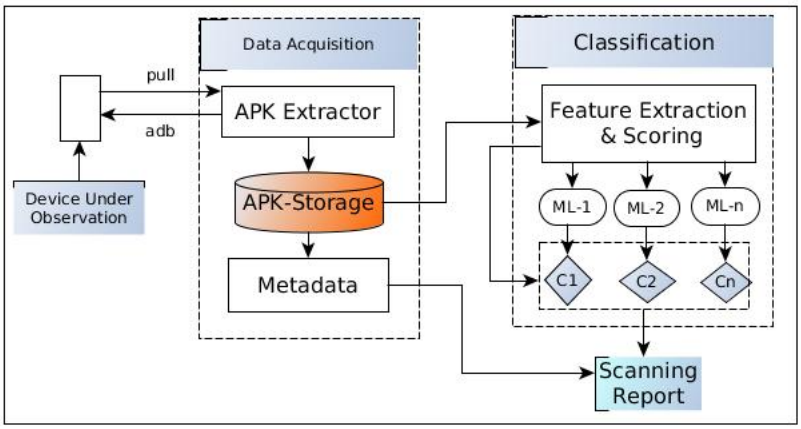


Figure 5.13: Block diagram of FAMOUS' architecture (ML-1 to ML-n are different machine Learning algorithms and C1 to Cn are classifiers which are output of ML training

5.4.1 FAMOUS' Architecture

The *FAMOUS* architecture has two main modules: *Data acquisition* and *Classification*. The main task of data acquisition module is to extract *apk* files from attached Android device. It uses Android Debug Bridge (ADB) protocol to connect Android device to analyst's system. It pulls and lists out all installed applications with size and type of application (system and third party). *FAMOUS* does not require a rooted device because "apk" can be pulled out with ADB protocol without root access on device.

Once all the "apk" files are pulled out from attached device, access of pulled "apk" storage is passed to classification module that do further pre-processing and label each apk either with benign or malware. The core of classification module is the Feature Extraction and Scoring (FES) component which create a feature set from given data samples. The output of FES is used to train machine learning algorithms and build a classifier. Many pre-processing tasks such as (permissions extraction, scoring and feature set generation) are carried out on pulled apk to generate the feature set. Fig. 5.13 shows a block diagram of **FAMOUS**' architecture.

5.5 Performance of Weighted Permission based Feature Set

This section presents and discusses the results of various experiments which were performed to compare the performance of the proposed *Weighted permission* and *boolean permission* based feature set. This section also presents the working result of the proposed forensic tool *FAMOUS*. This section explains about the dataset, experimental system and results of various experiments.

5.5.1 Dataset

To test the effectiveness of *EMSP* or *weighted permission* feature set and usability of *FAMOUS* two main experiments were carried out. In the first experiment (Experiment-I) the accuracy of different machine learning algorithms with different configuration was tested. In the second experiment (Experiment-II), *FAMOUS* was tested with real user's device. For the purpose of experiment-I, two datasets were amalgamated. The first dataset (dataset-1) has a total of $11,371$ Android applications for the experiment. The dataset has samples from both classes i.e. 5553 malware and 5818 benign. For this dataset, malware samples were adopted from *DREBIN* project (Arp et al., 2014) and benign samples were downloaded from *PlayDrone* archive (Viennot et al., 2014). The second dataset (dataset-2), has total 4317 samples, Malware samples were gathered from multiple online public archives such as *Contagio dump* (Parkour, 2016), *AndroMalShare* (Team et al., 2014) and *Andrototal* (Maggi et al., 2013). The benign samples were obtained from *PlayDrone* collection (Viennot et al., 2014) which has a sorted list of Google Play apks based on download count. This work downloaded top 999 and bottom 979 application from sorted list by pipelining Linux's *head* and *tail* output to *grep* and *wget* respectively. Along with this this work also collected 755 samples from different third party app stores and a torrent collection [4] of 1380 Google Play's paid apps and games. Table 5.2 illustrates the source and various categories of benign samples.

For Experiment-II, end-users' devices were required. So 4 users with the smartphone having different hardware with Android OS (different OS versions) were ran-

[4] https://kat.cr/1380-paid-android-apps-and-games-apk-t5344319.html

domly selected for the dataset. Users were guided to activate the developer options [5] on their phone and users' permission were obtained to activate Universal Serial Bus (USB) mode debugging option on each phone.

5.5.1.1 Pre-processing

All third party applications were verified with VirusTotal (VirusTotal, 2004) and it was found that 46 (7%) applications are detected as malware by minimum one antivirus engine. For this work the MD5 hash was used to get the scan reports and hence 130 applications do not have scanned result which was eliminated from our dataset along with the malicious applications. So the benign dataset has a total of 587 third party applications. The MD5 hashes and Secure Hashing Algorithm (SHA) hashes along with package names were used to identify samples uniquely.

Duplicate samples in dataset would skew the accuracy of the experiment, hence all duplicate samples were removed by using MD5 hash and package name and only the unique samples were in each dataset and each category. Dataset-1 was used for scoring, feature set generation and training whereas dataset-2 was used only for testing. It was made sure that samples present in dataset-2 are not available in dataset-1 and hence it is not used for score calculation. This separation represents the real-world scenario where new apps appear with new permission patterns.

[5] https://developer.android.com/studio/run/device.html

Table 5.2: Android apks collected from third party app stores

Category	9apps	fdroid & others	Total
Bussiness	63	2	65
Education	73	98	171
Entertainment	50	0	50
Games	79	78	157
Lifestyle	64	16	80
Multimedia	20	75	95
Personalisation	82	55	137
	Total	755 + 1380[†]= 2135	

[†] Torrent collection Google Play's paid Apps and Games

5.5.1.2 Class Labelling

All the samples in both datasets were scanned with VirusTotal (VirusTotal, 2004) and the proper class label was given accordingly. A decision on the class label was made separately for malware and benign based on the outcome of total positive results given by VirusTotal (VirusTotal, 2004).

For malware, *if any* criteria was adopted i.e, if any of the scanning engine flag positive (detects a sample as malware) for a given sample that considered as malware while for benign *if all* criteria was adopted i.e. if and only if, all engines pass a given sample as clean then a benign label is attached to the sample. Eq. 3.3 represents the class labeling (CL) process adopted based on positive score of VirusTotal's scanning engine E_i.

$$CL(M|B,S) = \begin{cases} M, & If(E_1(S) \vee E_2(S) \vee \ldots E_n(S)) \\ B, & elseIf(!(E_1(S)) \wedge !(E_2(S)) \wedge \ldots !(E_n(S))) \end{cases} \tag{5.6}$$

5.5.1.3 Feature extraction

Features i.e. requested permissions by app which are declared in *manifest* file were extracted with the help of *Androguard* module and a Python script. After extracting all permissions from each sample of the dataset, further processing such as permission frequency count, weight calculation etc. was done by using different Python scripts.

This section presented the details of the dataset used for both the experiments along with the pre-processing steps and class labelling process. In the following section 5.5.2 details of experimental system is presented.

5.5.2 Experimental System

An experimental system was prepared to carry out various kinds of experiments for the proposed feature set. The experimental system had Ubuntu 14.4, 64-bit OS running on Intel Core 2 Duo CPU $E7400@2.80GHzx2$ processor with $4GB$ primary memory and $500GB$ secondary memory. Python programming language with various modules was used for all experiments. APK processing was done using Androguard (Desnos, 2012), which is a Python based tool to perform different kinds of processing on an Android application. After collecting raw dataset (apk files), class of each *apk* was verified by using VirusTotal (VirusTotal, 2004) web service, which scanned each sample by nearly 55 parallel anti-virus engines. Scikit-learn (Pedregosa et al., 2011), a Python library was used for the purpose of machine learning algorithms training and testing. All the statistical calculation and graph plotting was also done using various Python based modules. To implement *FAMOUS* as the forensic tool, along with other modules *wxPython* module was used for developing Graphical User Interface (GUI).

This section explained about the experimental system used for various experiments. The following section 5.5.3 presents and discusses the results of the experiments on various performance metrics.

5.5.3 Results

In this section results of experiment-I and experiment-II are presented and the observed findings are listed out and explained.

5.5.3.1 Experiment-I: Machine Learning classifier performance test

In experiment-I, the performance of all six selected machine learning algorithms are compared by splitting dataset-I into training and testing set with a ratio of 70% and 30% respectively. Fig. 5.14 shows the ROC and AUC values of all selected classifiers. It shall be observed that Random Forest (RF) performance is best among all other classifiers. This result of RF is achieved by using 100 estimators during training. To find the optimum value for a number of estimators, Random Forest was again trained with the same training and testing dataset by adjusting estimator's value (10, 50, 100, 150, 200).

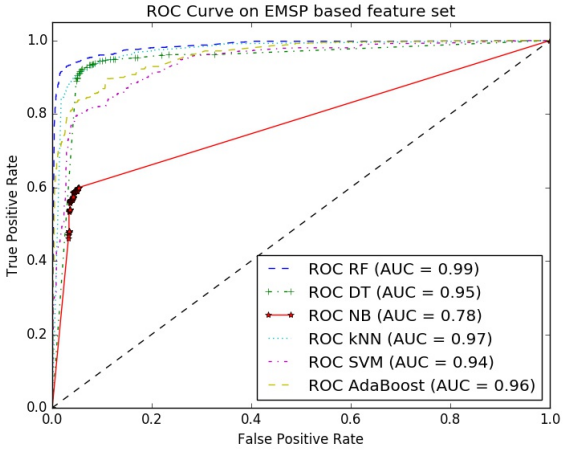


Figure 5.14: ROC for six different classifier on EMSP based feature

Fig. 5.15 shows the *ROC* and *AUC* value of *RF* with different numbers of estimators. It can be observed that with 100 estimators RF achieves its maximum AUC value i.e. 99.0% and there is no significant change in AUC value above 100 estimators. So, the *FAMOUS* has used aforementioned configuration for RF based classifier.

This work has compared the performance of proposed feature set with boolean permission based feature set. Fig. 5.16 shows the ROC and AUC values of all classifiers on boolean features. Table 5.3 shows the comparison of accuracy, precision, recall and F1-score for EMSP and boolean based feature set for all six selected algorithms. From Table 5.3 it is evident that EMSP based feature set is performing better than boolean based features set. It is also observed that kNN and SVM are giving better result on the boolean feature set.

After experimenting with train and test split of dataset-1 on EMSP (weighted fea-

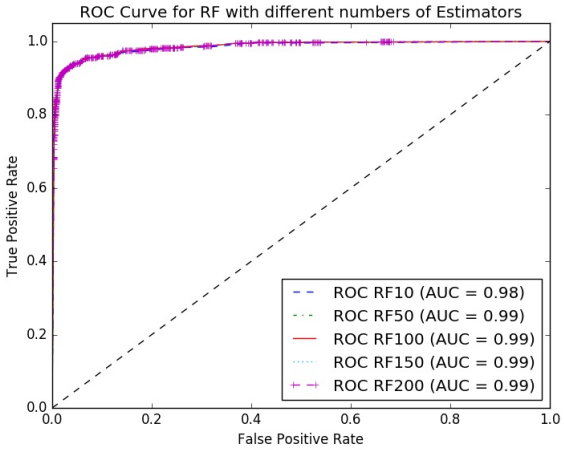


Figure 5.15: ROC for five different value for number of trees in random forest

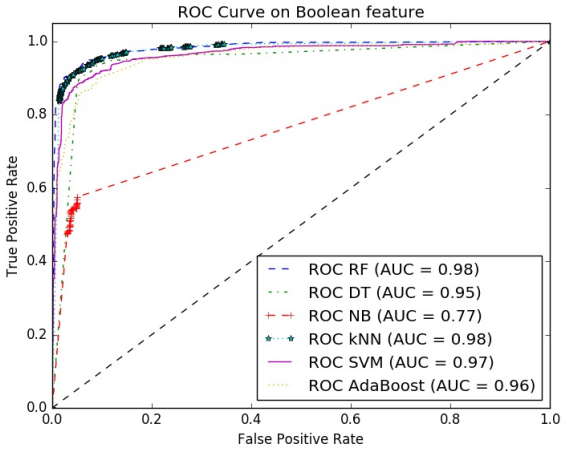


Figure 5.16: ROC of six classifiers on boolean features

ture set) and boolean feature, experiments were carried out to test the performance of aforementioned machine learning algorithms with dataset-2 (test dataset). As explained in Sec. 5.5.1, dataset-1 is used to calculate the EMSP and so even after the train-test split of dataset-1, there are chances of over-fitting due to the presence of samples on which score is calculated. So, to make classifiers robust and accurate on unseen data, testing algorithms with a separate dataset is recommended in the literature. Dataset-2 have samples that are not included in dataset-1. The dataset-1 was used for training and dataset-2 for testing. The performance of both EMSP and boolean feature set were tested.

Table 5.3: Classifiers performance on (70%-30%) dataset split with EMSP and boolean feature

Classifiers	Accuracy		Precision		Recall		F1-Score	
	EMSP	Boolean	EMSP	Boolean	EMSP	Boolean	EMSP	Boolean
RF	94.84	93.70	0.95	0.93	0.95	0.93	0.95	0.93
DT	93.17	92.20	0.93	0.92	0.93	0.92	0.93	0.92
NB	77.22	75.26	0.81	0.80	0.77	0.75	0.76	0.74
kNN	92.44	93.23	0.92	0.93	0.92	0.93	0.92	0.93
SVM	86.48	91.44	0.87	0.92	0.86	0.91	0.86	0.91
AdaBoost	91.32	90.27	0.91	0.90	0.91	0.90	0.91	0.90

Table 5.4: Classifiers performance on test dataset with EMSP and boolean feature

Classifier	accuracy		precision		recall		f1-score	
	EMSP	Boolean	EMSP	Boolean	EMSP	Boolean	EMSP	Boolean
RF	91.52	91.31	0.94	0.94	0.92	0.91	0.93	0.92
DT	88.95	88.21	0.94	0.94	0.89	0.88	0.91	0.9
NB	89.34	89.83	0.93	0.91	0.89	0.9	0.91	0.9
kNN	87.4	91.15	0.94	0.94	0.87	0.91	0.9	0.92
SVM	84.73	89.16	0.94	0.94	0.85	0.89	0.88	0.91
AdaBoost	86.5	89.53	0.94	0.94	0.86	0.9	0.89	0.91

Fig. 5.17 illustrates the ROC and AUC for classifiers with test dataset (dataset-2) on EMSP feature set while Fig. 5.18 show the ROC and AUC for classifiers with test dataset (dataset-2) on boolean feature set. Table 5.4 list out the performance of algorithms on different metrics. Values of all selected metrics are mentioned for both EMSP and boolean feature set.

5.5.3.2 Experiment-II:FAMOUS performance test

In Experiment-II, the performance of *FAMOUS* is tested with live Android devices. This work acquired Android based smartphone from random users and performed the scanning of their devices with *FAMOUS*. In this section, various aspects of experiment

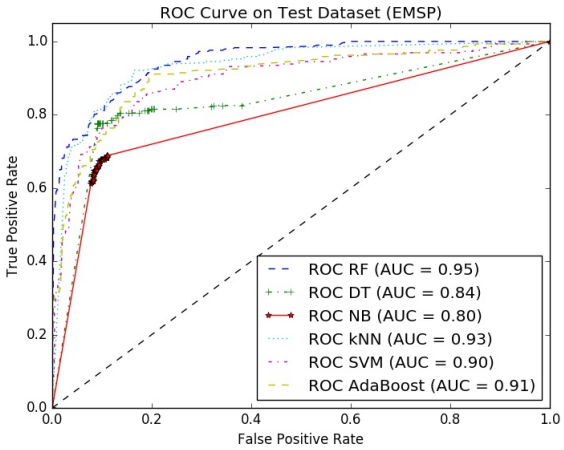


Figure 5.17: ROC on test dataset with EMSP features

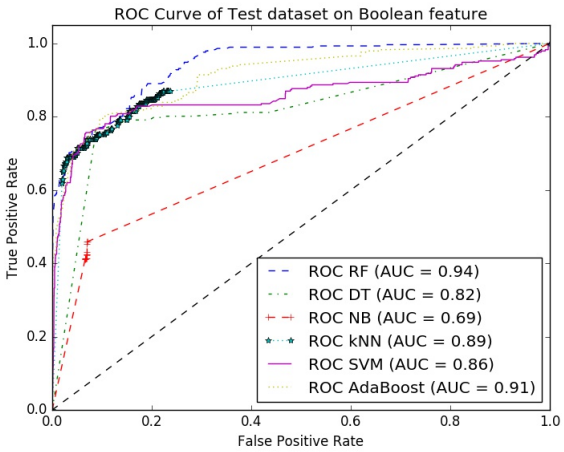


Figure 5.18: ROC on test dataset with boolean features

and result are presented and discussed. Before scanning the devices, users of the device were requested to activate the *developer options* on their smartphone. The *USB enabled debugging* mode with *ADB* was used to connect end user's devices to our experimental system. Once the connection was up and running, *FAMOUS* was executed with various configurations.

5.5.3.3 FAMOUS: GUI Interface

Aforementioned proposed approach was implemented as a forensic tool named as *FAMOUS*. Fig. 5.11 is showing the initial screen of *FAMOUS*, it lists out all the installed applications of attached device by showing its size and type of application. Type of

application is decided on the basis of installed location i.e. if an application is installed on *system* partition it considered as system application else in all other it is considered as 3^{rd} party application. The initial screen of *FAMOUS* has options to select individual applications or according to its type.

Once user/analyst selects the application/s, then with scan option all the selected applications will be scanned with underlying classifiers. The scan result of all selected applications will be displayed along with other metadata such as package, version, total permissions, and EMSP score. Fig. 5.12 shows the output of applications selected from one of the attached experimental device. The applications with the suspicious label are those which are flagged malicious by the underlying classifier and so these can be triaged for further analysis. Output window have the option to select listed applications with their class label and after selection, those can be moved to a separate folder on analyst's system which will help to go for further analysis.

5.5.3.4 FAMOUS: Operational result

FAMOUS is built with the objective to assist forensic analyst by triaging the applications into a category which enables them to make further decision. To achieve the objective, along with accuracy it is also important to be quick in extraction and scanning of applications. The accuracy of *FAMOUS* mainly depends upon the underlying classifiers that are explained in Section 5.5.3.1. This section explains the time taken by *FAMOUS* on different experimental devices. Table 5.5 list out all the devices with their make and model, applications pulling time and scanning time taken by *FAMOUS*. Pulling and scanning time is an average of five runs which is done to overcome any biasness.

Table 5.5: FAMOUS:Scanning results of four devices

Devices (Model&Version)	Total Apks	Pulling time(Mins.)	Scanning time(Sec.)	Suspicious Apks
GalaxyJ1-4.4.4	166	13.58	2.67	13
Micromax-A096-5.0.2	153	12.05	2.54	17
Lenovo-K50-5.0	249	15.62	4.25	26

It can be observed from Table 5.5 that *FAMOUS* is fast in pulling and scanning

the applications from attached devices. It took an average of 14 minutes to pulled all applications from attached device and 4 seconds to scan them. Time given in Table 5.5 is calculated by averaging all the applications size and total time was taken, so it can vary with devices due to installing applications' size. Table 5.5 also shows the number of suspicious apks which were identified as suspicious applications by *FAMOUS*. These apks have a high probability of being quarantined as malware with further triaging with a human expert.

This section explained about dataset, experimental system and results of various experiments carried out to show the performance comparison of the proposed weighted permissions (based on EMSP score) and boolean/binary permissions based feature set. This section also explained about *FAMOUS* and shown the performance result of its pulling and scanning time. The section 6.3 presents and explains the dataset, experimental system and result of various experiments which were carried out to show the performance of image based representation and image feature set for Android malware detection.

5.6 Summary

This Chapter proposed an *Weighted permission based feature set* which used each permission weight as feature value, oppose to the traditional boolean value based permission feature set. The Chapter provided details about creating *weighted permission feature set* which is accomplished by using three components: *Permission extractor*, *Scoring engine* and *Feature set generator*. The feature set is novel because boolean value were used as permission based feature set which were adopted in many earlier works as explained in Chapter 2. Various selected ML algorithms are trained and tested on both boolean and weighted feature set and their performance are measured. This chapter also presented the detail and result of training and testing. The best performing classifier is also used to create a forensic tool named *FAMOUS* which can be used to scan any Android device and is able to group all installed applications as *suspicious* or *benign*.

CHAPTER 6

Malicious Android Applications Detection using Multimodal Image Representations

In the previous Chapter 5, a weighted permission based feature set is proposed to detect Android malware by using machine learning algorithms. The Chapter explained the process of feature generation from malware and benign Android applications samples.

This Chapter presents a new approach which involves visualizing the Android application (here onward referred as apps) into various images format and uses machine learning algorithms to classify the given apps as benign or malware with the help of GIST features extracted from each image. The objective of this work is to test the discriminative potential of the image representation of the Android application and study the performance of different image format for malware detection.

This work has considered four image formats based on *Grayscale*, *RGB*, *CMYK*, and *HSL* color channels. Every sample from malware and benign set was converted into each of four image formats which finally resulted into four image dataset. The GIST features were extracted from each dataset and four feature set were created to train and test various machine learning algorithms on performance metrics such as *precision*, *recall*, *f-measure*, FPR and classification accuracy.

This Chapter discusses the process of converting Android applications to *image representations* and extracting features from those images for training and testing selected machine learning algorithms. The details about the dataset, experimental system and results of machine learning algorithms are presented in section 6.3.

6.1 Image Representation of Android Applications

This section explains about image representation of Android applications and discusses the four color formats used in this work. This section also explains the process of converting Android applications to the selected image formats.

This work has focused on the fact that "every computer program written in any high level language can be represented as machine code equivalent" i.e. in the form of '0s' and '1s'. The earlier works have considered the High-level representation of Android applications for feature extraction and creating anti-malware solutions. In this work, the binary equivalent of each Android apps is retrieved from disk and converted into four different image format based on selected color channels. As one can see in the Fig. 6.1, collected samples from both malware and the benign group were pre-processed and *stream of binary* representing each Android apps file was given as input for the *image conversion algorithm* (explained in further section), where each byte was represented as a color pixel on screen. The size of the each application was used to decide the height of image based on the given width which was selected as 256 in our all experiments. The Fig. 6.1 shows the steps involved from *raw sample* to *training* the machine learning algorithms. In the following section, the four image formats based on selected color channels are explained in details.

6.1.1 Color Channels

Visualizing binary file as the image has been used in past for different purposes such as bypassing file type restriction and classification of computer malware into it families (Kancherla and Mukkamala, 2013). The conversion of byte stream of the Android application (i.e. .apk [1] file format) into different color formats is chosen because it has not experimented and it is easy to compare the features of two images rather than analyzing the code of the application. This work has selected 4 color channels i.e. Grayscale, RGB, CMYK, and Hue, Saturation, Lightness (HSL) and converted each Android sample into an image according to the respective color channel. In the following sections, each color channel is explained in detail.

[1] Android packaging system packages application as apk which is used to share and install the application. The *apk* can be symbolized as *.exe* file on Windows OS.

6.1.1.1 Grayscale

In grayscale image format, each pixel carries only intensity information by having a single value in the range of $0-255$. Grayscale images are composed exclusively of shades of gray varying from *black at the weakest intensity* to *white at the strongest* one [2].

6.1.1.2 RGB

The RGB color format is based on three colors Red(R), Green (G), Blue (B). Within computer graphics or image processing applications, each color component is typically represented by 8 bits. Thus, a color value needs 24 bits to define a single color out of 16 million possible colors. For the higher color accuracy, 10 bits (or even 12 bits) are used for each color component. The RGB color format is very popular in computer graphics and image processing [2].

6.1.1.3 CMYK

CMYK is similar to RGB in the way of representation but each color is described by the color components Cyan (C), Magenta (M) and Yellow (Y), the additional component black (K) is used for gray and black color representation [2]. Visualizing both on the computer screen will not show any difference, CMYK is used for printing purposes while RGB is used in the digital display.

6.1.1.4 HSL

The components of HSL color model is Hue (H), Saturation(S) and Luminance (L). Hue defines the pure color, tone out of the color spectrum, saturation defines the mixture of the color tone with gray and finally, luminance defines the lightness of the resulting color [2].

This section presented and explained the four selected color channel by using which each of the Android application was converted to four different image formats. In the following section 6.1.2, the process of converting the Android application to the image format is explained in details.

[2] `http://www.equasys.de/colorformat.html`. [Accessed: 12-May-2017]

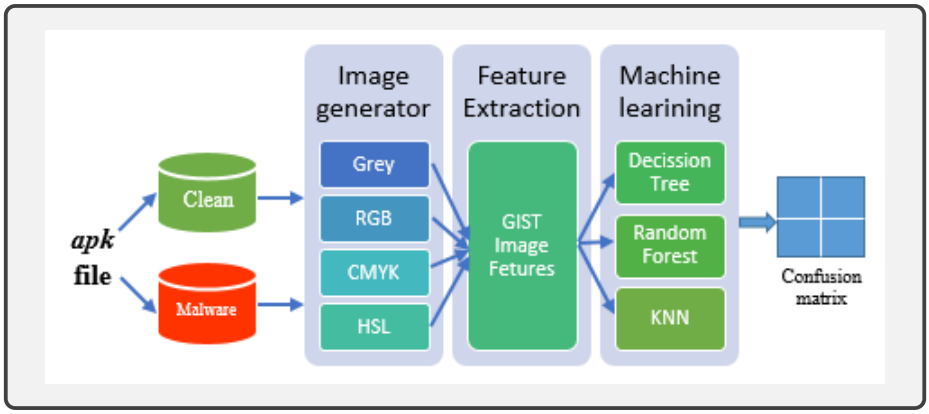


Figure 6.1: Workflow of image-based android malware detection system

6.1.2 Android application to Image Conversion

As it was stated in previous sections the binary stream of each apk file was converted into different color formats. The *image height* of each apk file was calculated by dividing the bytes size of apk by the given width. The *Image Generation algorithm* takes an apk file as input and outputs respective image format based on the called function respective to the color channel. For converting according to each color channel, a separate function is implemented and was used by the algorithm. Algorithm 4 and Algorithm 5 presented in following section summarizes the whole process of converting malicious and benign apps dataset to equivalent image dataset for feature extraction and machine learning algorithms training & testing.

6.1.2.1 Process and Method

Converting processed Android dataset having malicious and benign sample was carried out by using Algorithm 4. It takes malicious and benign Android applications along with required color channel as input parameters and after converting all the sample into image format output a image dataset of requested color channel. The algorithm is very simple, it initialized the *height* (α) and *width* (β) variable which pass as parameter to other called function. It also create two folders *MApkImages*, *BApkImages* for storing converted malicious and benign images respectively. It loop through malicious and benign apps folder and internally, uses *ConvertToImage()* function call passing *each app* (κ),*color channel*, *height* (α) and *width* (β). *ConvertToImage()* is presented and

explained as Algorithm 5.

Algorithm 4 Algorithm for converting Apps to Image Dataset

procedure CONVERTAPKTOIMAGE($\Psi, \Omega, ColorChannel$)

▷ Ψ : Malicious Android applications

▷ Ω : Benign Android applications

$\alpha \leftarrow$ getHeight() ▷ Calculate height for Image

$\beta \leftarrow$ width ▷ Get width for Image

▷ To store converted images of malicious apks

$MApkImages \leftarrow CreateFolder()$

▷ To store converted images of benign apks

$BApkImages \leftarrow CreateFolder()$

▷ Loop and convert malicious apks

for $\kappa \in \Psi$ **do**

$image \leftarrow ConvertToImage(\kappa, ColorChannel, \alpha, \beta)$

$MApkImages \leftarrow MoveFile(image)$

▷ Loop and convert benign apks

for $\kappa \in \Omega$ **do**

$image \leftarrow ConvertToImage(\kappa, ColorChannel, \alpha, \beta)$

$BApkImages \leftarrow MoveFile(image)$

return($MApkImages, BApkImages$)

Converting an Android application to an image is an iterative process where bits/bytes are read and grouped on the basis of target color channel. Algorithm 5 capture the essence of app to image conversion process. It takes four input parameters i.e. *each app* (κ),*color channel*, *height* (α) and *width* (β) and gives an image as output. In the proposed work, the width of the image was fixed to 256 and so depending upon the size of the app the height of each output image were calculated. After initializing the required variables, Algorithm 5 loop through the bits/bytes of the input application and group these according to the requested color channel requirement. These group bits/bytes are then converted to equivalent *Integer* value and store as in matrix format to create an image.

The next section shows the output of Algorithm 5 by taking a sample from the

Algorithm 5 Algorithm for converting APK to image

procedure CONVERTTOIMAGE($\kappa, ColorChannel, \alpha, \beta$)

▷ κ : An Android application

▷ ColorChannel:Target color channel (Grayscale,RGB,CMYK,HSL

▷ α: Image height

▷ β: Image width

▷ Loop through the app bytes

for $byte \in \kappa$ **do**

$bytesmatrix[\alpha][\beta] \leftarrow IntegerValue(byte)$

$image \leftarrow getPixelValue(bytesmatrix)$

return($image$)

malicious and benign group and converting them into image format based on the four selected color channel. The steps which are involved in carrying out this work are listed as:

1. **Data collection**: To carry out this work, as the first step malware and benign *apk* files were collected from the wild, online archives and various apps stores. The details about the dataset are presented further in section 6.3.1.

2. **Pre-processing**: As the second step, under preprocessing many actions are taken such as removing duplicate and unmatched samples. During preprocessing step the class label to each sample was also assigned. The *MD5 hashing* and *VirusTotal* service were used for duplicate removal and labeling respectively. The details about the pre-processing and experimental system are presented further in section 6.3.1.1 and section 6.3.2 respectively.

3. **Apk to Image conversion**: This is the third and the important step for this proposed work. By using bits/bytes stream each *apk* file was converted to four image format based on the selected color channel. In the previous section 6.1.2 this conversion process is explained in detail.

4. **Feature extraction**: The feature extraction is the fourth step and GIST image feature was used for this work, so GIST image descriptor of each image was extracted for all four image format from both malware and benign classes. The

details of GIST and few other popular image descriptors are explained further in section 6.2.1 .

5. **Training**: At fifth step, all the four generated feature set were used to train three selected machine learning algorithms i.e. Decision Tree (DT),Random Forest (RF) and k-Nearest Neighbors (kNN). The *10-fold cross-validation* method has used during training to achieve a robust result.

6. **Testing** : At sixth and final step, performance of all three machine learning algorithms have measured based on *recall*, *precision*, and *detection accuracy*. The result of training and testing is explained and presented in further section 6.3.3.

This section explained the process of Android application to image conversion with the help of Algorithm 4 and Algorithm 5. The section also listed out all the six steps taken to achieve the objective of this work. In the following section 6.1.2.2, the output of the Android application to four image format is shown by taking one sample each from malware and benign class.

6.1.2.2 Example of Application to Image output

To visualize and understand the earlier explained method of *app to image conversion*, this section shows the sample output of Algorithm 5 by passing two sample Android application (apps or apk) one each from malware and benign group. The Fig. 6.2 shows the output of both sample in four images different according to selected color channel. The leftmost image is a *grayscale* proceed by RGB, CMYK and HSL and the upper row is for the benign sample and bottom row is representing the malicious application.

It can be observed that different grouping of bits/bytes resulted in different image representation and it can be assumed that it also has an effect on the feature extraction and hence will affect the performance of machine learning algorithms. This assumption is tested by training three different machine learning algorithms and their results are presented further in section 6.3.3.

In the following section 6.2, the fourth step i.e. feature extraction is explained with listing and discussing few other image based features. A feature set generation algorithm which takes *set of images* as input and output a feature set is also presented in the following section 6.2.2.

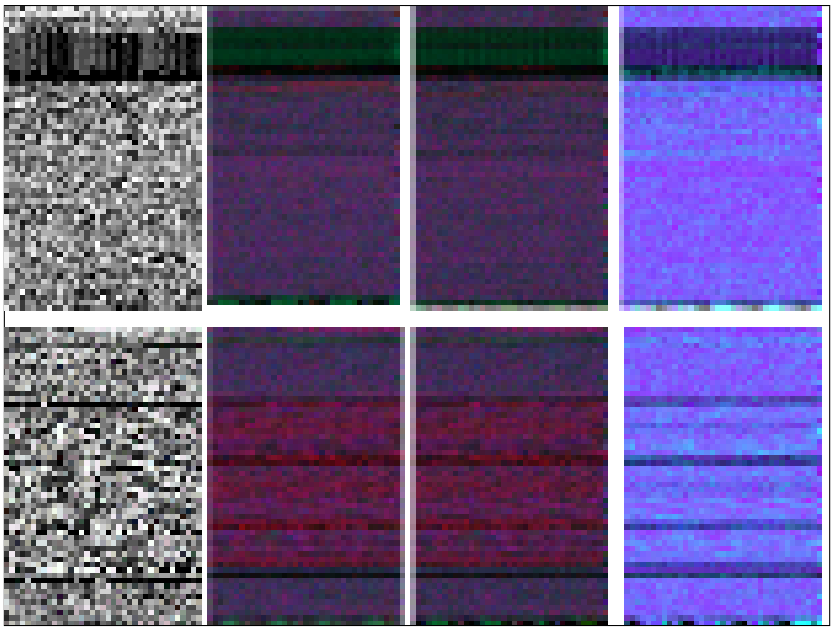


Figure 6.2: A benign and malware sample in all four image formats (grayscale, RGB, CMYK and HSL / from left to right)

6.2 Image features based Apps classification

This section explains various image based features and the feature extraction process which was used in this work. The process of feature extraction starts after the Android applications are converted to its equivalent image representation which also makes it is easy to apply techniques available for *image classification* for malware detection. The focus of this work is on the machine learning based techniques which required to extract features from the converted image dataset. The extracted features then pass as a feature set to build image classifiers which in fact will work as Android malware detector.

The following section lists out and explains some of the very popular *image features* which are used with machine learning algorithms.

6.2.1 Image Features

The features in the domain of *image classification* is known as feature descriptor. The feature descriptor is a representation of an image or an image patch that simplifies the image by extracting useful information and throwing away extraneous information. From the literature, it was observed that several types of image features are defined and

used in the various research domain. These features help in finding the uniqueness of an image as well as used for similarity check with other images. Some of the very popular *image features* i.e. SIFT, HoG and SURF are listed and explained in following section.

6.2.1.1 Scale Invariant Feature Transform (SIFT)

SIFT (Lowe, 1999) is one of the most popular feature descriptor among other image-based features. It is invariant to *scale*, *rotation* and *translation*. The popularity of SIFT comes due to lots of efficient implementations which have good results in many real applications. Although it is also true that there are newer image's feature descriptors which are proving to be better in terms of performance and simplicity.

A SIFT feature is a selected image region (also called *keypoint*) with an associated descriptor. Keypoints are extracted by the *SIFT detector* and their descriptors are computed by the *SIFT descriptor*. A SIFT keypoint is a circular image region with an orientation. It is described by a geometric frame of four parameters: the keypoint center coordinates x and y, its scale (the radius of the region), and its orientation (an angle expressed in radians). The SIFT detector uses as keypoints image structures which resemble "blobs" [3].

6.2.1.2 Histogram of Oriented Gradients (HoG)

HoG (Dalal and Triggs, 2005) is an another popular descriptor which is simple and faster than SIFT. The basic difference between these two is that HoG uses a "global" feature to describe an object rather than a collection of "local" features as used by SIFT. The HoG uses a sliding detection window which is moved around the image. At each position of the detector window, a HOG descriptor is computed for the detection window. HoG counts occurrences of gradient orientation in localized portions of an image. The working principle of HoG is that the target image is divided into small connected regions called *cells*, and for the *pixels* within each cell, a histogram of gradient directions is compiled and the concatenation of these histograms create the HoG descriptor of the image. *Contrast-normalization* for the local histograms is done by calculating a measure of the intensity across a larger region of the image, called a *block*. This normalization improves the accuracy and results in better invariance to changes in illumination

[3] `http://www.vlfeat.org/api/sift.html` [Accessed: 12-May-2017]

and shadowing.

6.2.1.3 Speeded Up Robust Features (SURF)

SURF (Bay et al., 2008) is very similar to SIFT but it is faster and it is proven to performance better than SIFT. With improved performance yet SURF is not as popular as SIFT. SURF can be seen as approximation of SIFT. SURF replace computing the Gaussian scale-space and the histograms of the gradient direction by fast approximations. To detect interest points, SURF uses an integer approximation of the determinant of Hessian blob detector, which can be computed with 3 integer operations using a precomputed integral image. Its feature descriptor is based on the sum of the *Haar wavelet* response around the point of interest. These can also be computed with the aid of the integral image.

6.2.1.4 GIST

GIST descriptors are the representation of an image in the low dimension that contains enough information to identify the scene. GIST descriptor was proposed by (Oliva and Torralba, 2001). GIST is features of an image which uses a wavelet decomposition of the image and being used widely for scene classification and object classification (Nataraj et al., 2011; Douze et al., 2009; Moudni et al., 2013). The gist descriptor is built from 8 oriented edge responses at 4 scales with 4×4 non-overlapping windows. The Gist feature is computed by convolving an oriented filter with the image at several different orientations and scales. This way the high- and low-frequency repetitive gradient directions of an image can be measured. The scores for the filter convolution at each orientation and scale are stored in an array, which is the Gist feature for that image. GIST descriptors were calculated using a *Gabor filter* at 8 orientations at 4 scales. This work has used GIST descriptor which was extracted from the image and passed to the machine learning algorithms for training and testing.

This section explained about *image features* and listed out few very popular image features along with the GIST which is used in this work. In the following section 6.2.2, the process of feature set generation from the *set of converted images* is explained.

6.2.2 Process for Feature set Generation

The feature set generation is important and the prerequisite for training and testing machine learning algorithms. It is shown in Fig. 6.1 (presented at the beginning of this Chapter), that feature extraction is the next step after converting Android apps dataset to equivalent image dataset. The section 6.1.2 explained the process of converting apps dataset to four equivalent image dataset based on the four selected color channel. Generating feature set which is the vector representation of image descriptor along with the class label is an iterative process. The extraction process loop through each image of a particular class i.e. malicious or benign and extract GIST descriptor and store in a file along with the adequate class label. The same process is repeated for both classes and for the all four image formats. The iterative extraction process outputs four labeled feature set which further used to train and test three selected machine learning algorithms.

The Algorithm 6 presents the steps used for generating the four feature sets from the converted image datasets. The converted images were organized in the folder based on the *color channel name* and each of the *color channel named folder* has malicious and benign folder having malware and benign images respectively. The algorithm takes these four folder name as input and returns four feature set based on these color channel name as CSV files. Each of these *color channel named folder* was processed one after other and under each of these folders all the malware and benign images were iterated for GIST feature extraction and simultaneously extracted feature with the class label were written to the output file which has the *color channel name* (grayscale.csv, rgb.csv, cmyk.csv and hsl.csv) as the file name. The *ReadFolder()* method was used for reading image and the *GetGIST(image)* method was used for GIST features extraction from an individual image passed as input parameter.

This section explained various popular image features and also explained the process of feature set generation from the *set of converted images*. The Algorithm 6 summarized all the steps taken during the feature set generation process. The all four generated feature sets were used to train and test three selected machine learning algorithms for knowing the potential of the image representation of the Android application for malware detection and the effect of different color channel based image on malware de-

Algorithm 6 The Algorithm for the feature set generation

procedure GENERATEFEATURESET($[Grayscale, RGB, CMYK, HSL]$)

$ColorChannels \leftarrow [Grayscale, RGB, CMYK, HSL]$

▷ Have images of malicious apks

$MApkImages \leftarrow ReadFolder()$

▷ Have images of benign apks

$BApkImages \leftarrow ReadFolder()$

▷ Loop through ColorChannel

for $ColorChannel \in ColorChannels$ **do**

$outputfile \leftarrow ColorChannel.csv$

▷ Loop and extract GIST of malicious image

for $image \in MApkImages$ **do**

$class \leftarrow' malware'$

$features \leftarrow GetGIST(image)$

$outputfile \leftarrow (features + class)$

▷ Loop and extract GIST of benign image

for $image \in BApkImages$ **do**

$class \leftarrow' benign'$

$features \leftarrow GetGIST(image)$

$outputfile \leftarrow (features + class)$

return($[Grayscale, RGB, CMYK, HSL].csv$)

tection. The results of these training and testing are presented further in section 6.3.3.

6.3 Performance of Image features for Android Application Classification

This section presents the details about the experiments which were carried out to test the discriminative capability of feature set created from the image representation of Android Applications. This section gives the detail about dataset, pre-processing and experimental system. This section also presents the experiments' result with discussion and conclusion. The following section 6.3.1 explains about the datset used for the proposed image-feature based feature set.

6.3.1 Dataset

Initially this work started with a total of 275 samples, out of which 136 were collected as benign from *mobile9* [4], *9apps* [5] and *androidapksfree* [6] and 139 are malware.

6.3.1.1 Pre-processing

The duplicate samples were removed by using MD5 hash comparisons. The final dataset has total 246 samples in which 108 were benign and 138 were malicious apk. The class label of each sample of dataset was verified by VirusTotal. The method was similar to the technique explained in section 5.5.1.2. Table 6.1 summaries the number of samples and pre-processing method used for preparing the dataset used in this work.

[4] `http://www.mobile9.com/.[Accessed:20-Jul-2015]`

[5] `http://www.9apps.com/.[Accessed:20-Jul-2015]`

[6] `http://www.androidapksfree.com/.[Accessed:21-Jul-2015].`

Table 6.1: Benign and malware sample statistic

Actions	Benign	Malware
Initial Count	136	139
Without Duplicates	136	138
After labelling	108	138
Average of apk (KBs)	2547	1084
Duplicate removal	MD5	MD5
Labelling	VirusTotal	VirusTotal

6.3.1.2 Class Labelling

The VirusTotal service was used to find out the exact class (malware vs. benign) of the collected samples. the VirusTotal[7] provides API service, which checks any sample against more than 50 Anti-virus engine. Each sample's MD5 were calculated and submitted to VirusTotal, and if the MD5 hash is not available then the apk file itself were uploaded to the service and result were logged. The rule to decide class label of each sample is very simple, apk considered clean by all Anti-virus engine taken as benign else malware. During labeling, it was found that total 28 samples from benign collection term as malware which is the point to ponder. During this work manually 28 benign samples were submitted because for them previous analysis was not available.

6.3.1.3 Feature extraction

Apk to image conversion allows us to use image features, so GIST image descriptor was selected among various other image features. The Python implementation of GIST extractor with the help of the script was used to extract GIST from each image. Details about apk to image converstion, feature set generation and image features are explained in previous sections.

This section explained about dataset, pre-processing, class labelling and feature extraction to create the image based feature set from Android's apk files. In the following section 6.3.2 details of the experimental system is provided.

[7] `www.virustotal.com`

6.3.2 Experimental System

For building classifiers and testing the performance of GIST features, *scikit-learn-0.14.1* implementation of Decision Tree (DT), Random Forest (RF) and k-Nearest Neighbors (kNN) were trained on GIST feature set which were generated from converted images. DT and kNN were used with default configuration whereas in case of RF, 100 estimators were used instead of default. *Scikit-learn* cross-validation and confusion matrix module were used for performance evaluation.

6.3.3 Results

This section presents the results of machine algorithms trained on GIST features extracted from converted images. The value for primitive metrics such as TP, FP etc. and advance metrics such as *accuracy*, *error-rate* etc. are presented in different section.

6.3.3.1 Metadata analysis

This work also calculated and compared the size of each sample and it was found that there is no considerable different in sizes of the application before and after converting into the image. The average size calculated for normal apk files before image conversion is $2547KB$ for benign apk's and $1084KB$ for malware apk's. The Grayscale image files having an average difference in size with normal apk files as $507KB$ for benign and $527KB$ likewise all other formats also have average size difference but those are not large enough to be considerable. In Fig. 6.3 the size range of malware and benign apk files are presented. It has observed that malware apk are normally much smaller in size than the benign sample, out of 138 malware samples 112 (i.e. 81%) are in the range of $1 - 1000$ KBs whereas only 24 (22%) benign samples are in same range. It was also observed that numbers of benign samples sizes are distributed in all size range. These two observations conclude that attack tries to keep malware size low to make distribution and installation easy while it makes detection hard.

6.3.3.2 Basic metrics and Confusion Matrix

The Table 6.2 provides the value for all the attributes of confusion matrix for each trained classifier on all four image set. These values are used to calculate advanced per-

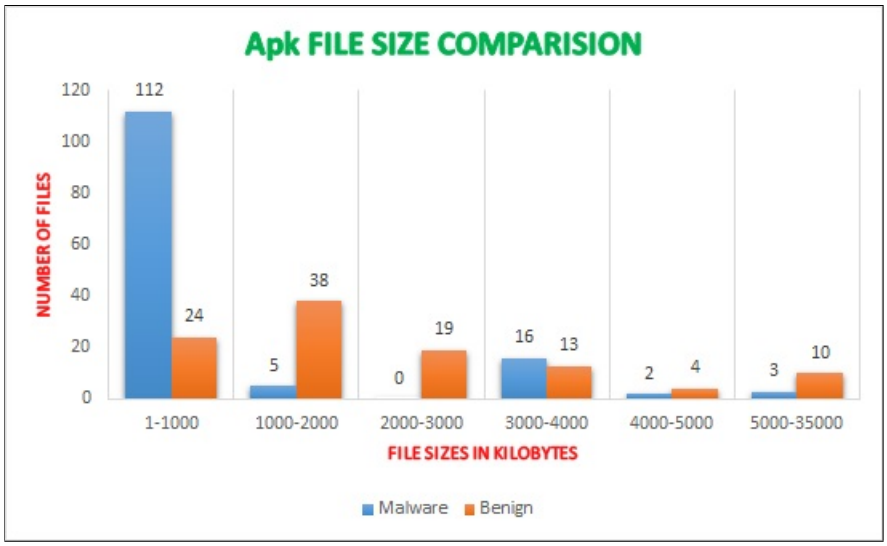


Figure 6.3: Size comparison of malicious and benign apps

formance metrics such as accuracy, error rate etc.. Results of these metrics are presented in next section.

Table 6.2: Confusion matrix values of classifiers

Image Format	Metric	DT	RF	kNN
Grayscale	TP	81%	93%	89%
	FP	12%	10%	10%
	FN	19%	7%	11%
	TN	88%	90%	90%
RGB	TP	69%	96%	82%
	FP	20%	23%	20%
	FN	31%	4%	18%
	TN	80%	77%	80%
CMYK	TP	65%	95%	81%
	FP	20%	25%	17%
	FN	35%	5%	19%
	TN	80%	75%	83%
HSL	TP	72%	95%	84%
	FP	18%	24%	16%
	FN	28%	5%	16%
	TN	82%	76%	84%

This work has also created the heat map for the confusion matrix values for each classifier on all four image format. Fig.6.4 shows the heat map of confusion matrix for Decision Tree (DT).

Fig.6.5 shows the heat map of confusion matrix for Random Forest (RF).

Fig.6.6 shows the heat map of confusion matrix for k-Nearest Neighbors (kNN).

The heat map is the visual way to understand the performance of classifier performance and it simplifies the process of comparison. In the current heat map, higher *reddish* color in first quarter correlated to a better performance.

6.3.3.3 Advance metrics

Through bar chart Fig. 6.7 summarizes the performance of each classifier for all four image feature set. Among three trained classifier it was observed that Random Forest (RF) is performing best, which has an average accuracy 86% for all four color and 91%

(a) Grayscale

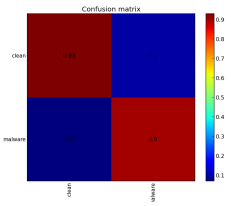
(b) RGB

(c) CYMK

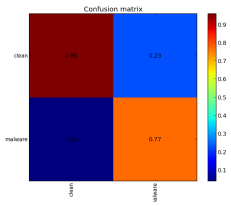
(d) HSV

Figure 6.4: Confusion matrix for decision tree

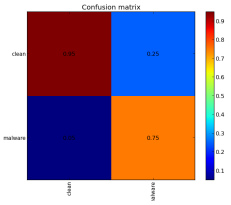
for grayscale image feature set. The error rate for Random forest in average is 12% and for grayscale only 8% which is minimum among all another classifier. Decision Tree (DT) method is delivering worst accuracy percentage in classification having 76% average classification accuracy and 22% average Error Rate. With aforementioned observations, it is concluded that Random forest on grayscale image feature is best among all others. Table 6.3 presents the value for all the advanced metrics for all three classifiers and for all the four image formats.



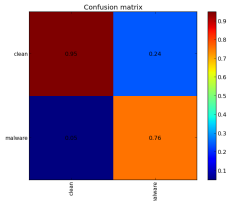
(a) Grayscale



(b) RGB



(c) CYMK

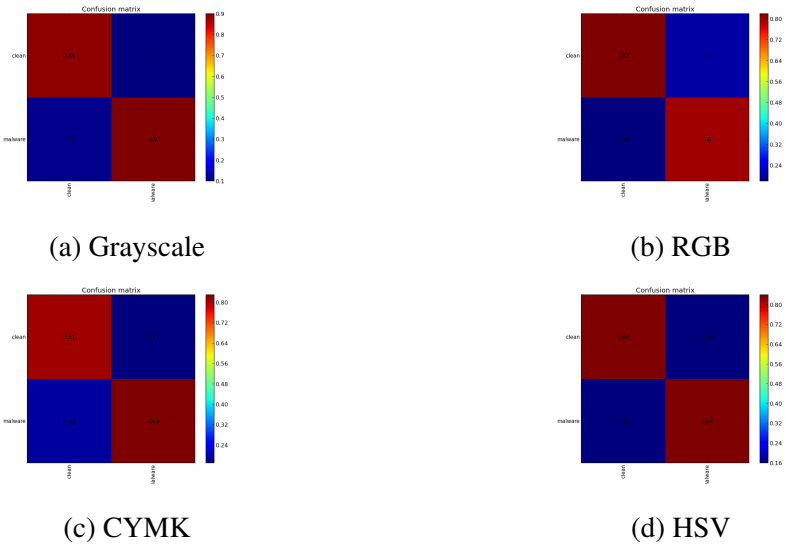


(d) HSV

Figure 6.5: Confusion Matrix for Random Forest

Table 6.3: Classifiers performance on various metrics

Image Format	Metric	DT	RF	kNN
Grayscale	TPR	81%	93%	89%
	TNR	88%	90%	90%
	F-m	83%	91%	88%
	CA	84%	**91%**	89%
	ER	15%	**8%**	10%
RGB	TPR	69%	**96%**	82%
	TNR	80%	77%	80%
	F-m	72%	86%	80%
	CA	74%	86%	81%
	ER	25%	13%	19%
CMYK	TPR	65%	95%	81%
	TNR	80%	75%	83%
	F-m	69%	85%	80%
	CA	72%	85%	82%
	ER	27%	15%	18%
HSL	TPR	72%	95%	84%
	TNR	82%	76%	84%
	F-m	75%	86%	83%
	CA	77%	85%	84%
	ER	23%	14%	16%



(a) Grayscale (b) RGB

(c) CYMK (d) HSV

Figure 6.6: Confusion matrix for nearest neighbour

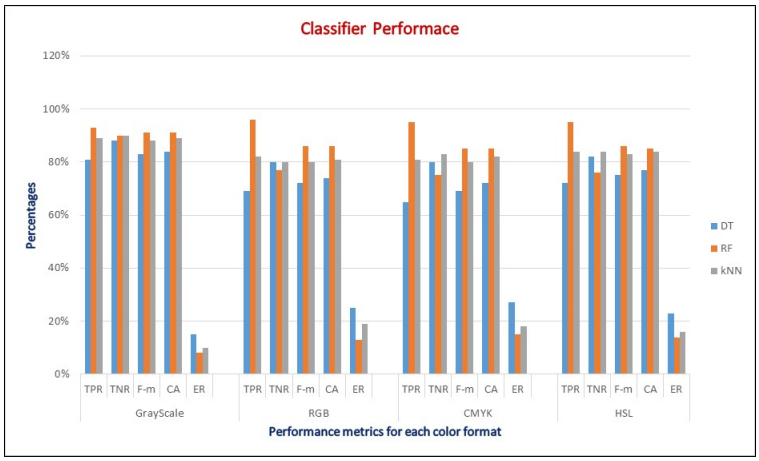


Figure 6.7: Classifier performance comparison with bar chart

6.4 Summary

The main objective of this Chapter was to explore the discriminative potential of the image representation of Android applications for malware detection. By representing each app into four different types of image based on selected color channels, this work also studies the effect of different image representation (based on color channels) on Android malware detection process.

This Chapter proposed a method to use *image features* extracted from converted *im-*

age representation of Android applications. Each of the samples from both malware and the benign group was converted to *image representation* in four different formats based on selected color channel. All the converted images from each individual color formats formed a separate dataset and so four different feature sets were created from the dataset having malware and benign samples. Three selected machine learning algorithms are trained and tested on all the four feature set and their performance is measured on various metrics. This chapter also present the result of training and testing along with dataset, experimental system, and pre-processing methods.

CHAPTER 7

Conclusions and Future scopes

7.1 Conclusions

Under the proposed framework, with static features this work is able to achieve 98.4% and 94.84% accuracy with Random forest for malicious portable executable and Android application classification respectively. A detailed conclusion is presented below:

- With the proposed integrated feature set built from the portable executable headers fields, Random forest achieved the accuracy of 98.4% with 10-fold cross-validation and 89.23% accuracy with test dataset evaluation.
- The boolean feature set built with only section name of the portable executable file, Random forest achieved the accuracy of 93% with features having non-zero information gain score and 92% accuracy with top 20 selected features.
- The proposed weighted permission based feature set performed better than boolean feature set and were able to achieve 94.84% accuracy with Random forest classifier.
- The GIST features extracted from the grayscale representation of Android apps yields the best result i.e. 91% accuracy and only 8% error rate with Random forest classifier.

7.2 Answers of the Research Questions

This thesis explores the research problem and achieved the objectives set for the research problem. In this section the answers of research questions (set for the research

problem) is stated with the available results and observations.

Research Question 1: How the performance of PE headers-based features can be improved?

The experimental results for integrated feature sets shows an improvement in the performance of classifiers for the malware detection. This clearly indicates that deriving new feature and value from the raw header field's values (derived features) by use of domain knowledge and rules applicable to the field can improve the performance of PE headers-based features.

Research Question 2: What is the potential of *section name* as features to build ML based malware detector?

In Chapter 4, the experiment with *section-name* based feature set is achieved the accuracy of 93% with boolean features. This indicates that *section name* has potential to be used as features to build ML based malware detector and the performance can be improved by integrating it with other feature set.

Research Question 3: How weighting the permission of Android applications will affect the detection performance of ML algorithms?

From Chapter 5, It was observed that after giving weight to the permission and then creating feature set improved the performance of classifiers. On the basis of experimental results and observation, it can be stated that weighting the permission of Android applications has a positive impact on detection performance of ML algorithms.

Research Question 4: Do the image representation of Android application can improve the malicious android applications detection?

In Chapter 6, the image based Android application detection has achieved the accuracy of 91% which is better than the baseline. Apart from the accuracy image based Android application detection also address some of issues of malware detection such as code obfuscation, code packing etc. So, on the basis of experimental result and the benefits of image based detection, we can clearly state that it improve the malicious android applications detection.

Research Question 5: How the various feature selection and machine learning algorithms impact the performance of the features?

For integrated feature set, we have observed that classifiers and feature set accuracy changes with the number of selected features. With experimental result, we observed that after a fixed number of features the performance doesn't improve further so there is no need of using all the features, instead we can use selected features with best accuracy. Similar, trend is observed in *section name* based feature set, with top 20 features Random Forest achieved 92% accuracy which 1% lesser than the accuracy achieved with all features having non-zero information gain. In all the experiments, we observed that with same feature set different ML algorithms performed differently which is due to the intrinsic property of various ML algorithms. With the experimental results and observations, we can state that feature selection and different machine learning algorithms impact the performance of the features.

7.3 Future Scopes

This work has utilized the static features for malicious Portable Executable (PE) and *apk* files detection using machine learning algorithms. The extraction of dynamic features is complex and time-consuming but capable of addressing issues of static features. As future scopes of this work dynamic features can be extracted by dynamic analysis and with the help of various experiments its discriminative capability can be measured.

This work has used the dataset with adequate samples but as future works experiments can be performed with the very large dataset. The result with larger dataset will increase the trust of the framework.

This work focused on the individual set of features such as only headers fields, section name, permissions, and GIST, in future work these set of features can be integrated with another set of features to boost the classification performance of algorithms. With enhanced designed experiments, the set of discriminative features can be selected from various sets and will be helpful to build more effective and efficient malware detector.

In future works, the scope of hybrid features i.e. the combination of static and dynamic features can also be explored for malware detection.

Appendix

Table 1: List of header's fields used as raw features

Header	**Fields**
DOS_HEADER(6/19)	e_cblp e_cp e_cparhdr e_maxalloc e_sp e_lfanew
FILE_HEADER(1/7)	NumberOfSections
OPTIONAL_HEADER (Standard Fields (8))	MajorLinkerVersion MinorLinkerVersion SizeOfCode SizeOfInitializedData SizeOfUninitializedData AddressOfEntryPoint BaseOfCode BaseOfData
OPTIONAL_HEADER (Windows Specific (13/22))	MajorOperatingSystemVersion MinorOperatingSystemVersion MajorImageVersion MinorImageVersion CheckSum MajorSubsystemVersion MinorSubsystemVersion Subsystem SizeOfStackReserve SizeOfStackCommit SizeOfHeapReserve SizeOfHeapCommit LoaderFlags

Table 2: DOS header fields

S.N.	Field	Description
1	e_magic	Magic number
2	e_cblp	Bytes on last page of file
3	e_cp	Pages in file
4	e_crlc	Relocations
5	e_cparhdr	Size of header in paragraphs
6	e_minalloc	Minimum extra paragraphs needed
7	e_maxalloc	Maximum extra paragraphs needed
8	e_ss	Initial (relative) SS value
9	e_sp	Initial SP value
10	e_csum	Checksum
11	e_ip	Initial IP value
12	e_cs	Initial (relative) CS value
13	e_lfarlc	File address of relocation table
14	e_ovno	Overlay number
15	**e_res1**	Reserved words
16	e_oemid	OEM identifier (for e_oeminfo)
17	e_oeminfo	OEM information; e_oemid specific
18	**e_res2**	Reserved words
19	e_lfanew	File address of new exe header

Table 3: File header fields

S.N.	Field	Description
1	Machine	Target machine type
2	NumberOfSections	The total number of sections
3	TimeDateStamp	Time when header generated
4	PointerToSymbolTable	Address
5	NumberOfSymbols	Total Symbols
6	SizeOfOptionalHeader	Size of **PE Optional Header**
7	Characteristics	bit flags, each holding some characteristics

Table 4: Optional header fields

S.N.	Field	Description
1	Signature	
2	MajorLinkerVersion	
3	MinorLinkerVersion	
4	SizeOfCode	
5	SizeOfInitializedData	
6	SizeOfUninitializedData	
7	AddressOfEntryPoint	The RVA of the code entry point
8	BaseOfCode	
9	**BaseOfData**	Only in PE32
10	ImageBase	
11	SectionAlignment	
12	FileAlignment	
13	MajorOSVersion	
14	MinorOSVersion	
15	MajorImageVersion	
16	MinorImageVersion	
17	MajorSubsystemVersion	
18	MinorSubsystemVersion	
19	Win32VersionValue	
20	SizeOfImage	
21	SizeOfHeaders	
22	Checksum	
23	Subsystem	
24	DLLCharacteristics	
25	SizeOfStackReserve	
26	SizeOfStackCommit	
27	SizeOfHeapReserve	
28	SizeOfHeapCommit	
29	LoaderFlags	
30	NumberOfRvaAndSizes	
31	DataDirectory	

References

Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42. IEEE, 2004a.

Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. Detection of new malicious code using n-grams signatures. In *PST*, pages 193–196, 2004b.

Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 183–194. ACM, 2016.

Faraz Ahmed, Haider Hameed, M Zubair Shafiq, and Muddassar Farooq. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, pages 55–62. ACM, 2009.

Shadi A Aljawarneh, Raja A Moftah, and Abdelsalam M Maatuk. Investigations of automatic methods for detecting the polymorphic worms signatures. *Future Generation Computer Systems*, 60:67–77, 2016.

Altyeb Altaher, Ammar ALmomani, Mohammed Anbar, Sureswaran Ramadass, et al. Malware detection based on evolving clustering method for classification. *Scientific Research and Essays*, 7(22):2031–2036, 2012.

Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel JG Van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. Measuring the cost of cybercrime. In *The economics of information security and privacy*, pages 265–300. Springer, 2013.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

Saba Awan and Nazar Abbas Saqib. *Detection of Malicious Executables Using Static and Dynamic Features of Portable Executable (PE) File*, pages 48–58. Springer International Publishing, Cham, 2016. ISBN 978-3-319-49145-5. doi: 10.1007/978-3-319-49145-5_6. URL `http://dx.doi.org/10.1007/978-3-319-49145-5_6`.

Jinrong Bai, Junfeng Wang, and Guozhong Zou. A malware detection scheme based on mining format information. *The Scientific World Journal*, 2014, 2014.

Usukhbayar Baldangombo, Nyamjav Jambaljav, and Shi-Jinn Horng. A static malware detection system using data mining methods. *arXiv preprint arXiv:1308.2831*, 2013.

Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer vision and image understanding*, 110(3):346–359, 2008.

Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

Mohamed Belaoued and Smaine Mazouzi. A real-time pe-malware detection system based on chi-square test and pe-file features. In *IFIP International Conference on Computer Science and its Applications*, pages 416–425. Springer, 2015.

John Bethencourt, Dawn Song, and Brent Waters. Analysis-resistant malware. 2008.

Daniel Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844. ACM, 2012.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium on Secure Software Engineering*, pages 37–44, 2006.

Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Usenix security symposium*, page 15, 2011.

Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 34–44. ACM, 2004.

Jianyong Dai, Ratan K Guha, and Joohan Lee. Feature set selection in data mining techniques for unknown virus detection: a comparison study. In *CSIIRW*, page 56, 2009.

Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

Baptiste David, Eric Filiol, and Kévin Gallienne. Structural analysis of binary executable headers for malware detection optimization. *Journal of Computer Virology and Hacking Techniques*, pages 1–7, 2016.

Anthony Desnos. androguard/androguard, 2012. URL `https://github.com/androguard/androguard`.

Francesco Di Cerbo, Andrea Girardello, Florian Michahelles, and Svetlana Voronkova. Detection of malicious applications on android os. In *Computational Forensics*, pages 138–149. Springer, 2010.

Matthijs Douze, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg, and Cordelia Schmid. Evaluation of gist descriptors for web-scale image search. In *Proceedings of the ACM International Conference on Image and Video Retrieval*, page 19. ACM, 2009.

Thomas Dube, Richard Raines, Gilbert Peterson, Kenneth Bauer, Michael Grimaila, and Steven Rogers. Malware target recognition via static heuristics. *Computers & Security*, 31(1):137–147, 2012.

Thomas E Dube, Richard A Raines, Michael R Grimaila, Kenneth W Bauer, and Steven K Rogers. Malware target recognition of unknown threats. *IEEE Systems Journal*, 7(3):467–477, 2013.

Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

Yuval Elovici, Asaf Shabtai, Robert Moskovitch, Gil Tahan, and Chanan Glezer. Applying machine learning techniques for detection of malicious code in network traffic. In *Annual Conference on Artificial Intelligence*, pages 44–50. Springer, 2007.

Zheran Fang, Weili Han, and Yingjiu Li. Permission based android security: Issues and countermeasures. *computers & security*, 43:205–218, 2014.

Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. A permission verification approach for android mobile applications. *Computers & Security*, 49:192–205, 2015.

Mo Ghorbanzadeh, Yang Chen, Zhongmin Ma, T Charles Clancy, and Robert McGwier. A neural network approach to category validation of android applications. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 740–744. IEEE, 2013.

Ibai Gurrutxaga, Olatz Arbelaitz, Jesus Ma Perez, Javier Muguerza, Jose I Martin, and Inigo Perona. Evaluation of malware clustering based on its dynamic behaviour. In *Proceedings of the 7th Australasian Data Mining Conference-Volume 87*, pages 163–170. Australian Computer Society, Inc., 2008.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

You Joung Ham and Hyung-Woo Lee. Detection of malicious android mobile applications based on aggregated system call events. *International Journal of Computer and Communication Engineering*, 3(2):149, 2014.

Kyoung Soo Han, Jae Hyun Lim, Boojoong Kang, and Eul Gyu Im. Malware analysis using visualized images and entropy graphs. *International Journal of Information Security*, 14(1):1–14, 2015.

KyoungSoo Han, Jae Hyun Lim, and Eul Gyu Im. Malware analysis method using visualization of binary files. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 317–321. ACM, 2013.

PW Hodgson. The threat to identity from new and unknown malware. *BT technology journal*, 23(4):107–112, 2005.

Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, page 48, 2007.

Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M. Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers Security*, 68:36 – 46, 2017. ISSN 0167-4048. doi: http://dx.doi.org/10.1016/j.cose.2017.03.011.

Rafiqul Islam, Ronghua Tian, Lynn M Batten, and Steve Versteeg. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 2012.

Kesav Kancherla and Srinivas Mukkamala. Image visualization based malware detection. In *Computational Intelligence in Cyber Security (CICS), 2013 IEEE Symposium on*, pages 40–44. IEEE, 2013.

Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1 (1-2):13–23, 2005.

Jeffrey O Kephart and William C Arnold. Automatic extraction of computer virus signatures. In *4th virus bulletin international conference*, pages 178–184, 1994.

Dmitry Komashinskiy and Igor Kotenko. Using low-level dynamic attributes for malware detection based on data mining methods. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 254–269. Springer, 2012.

Shishir Kumar and Durgesh Pant. Detection and prevention of new and unknown malware using honeypots. *arXiv preprint arXiv:0912.2293*, 2009.

Arun Lakhotia, Andrew Walenstein, Craig Miles, and Anshuman Singh. Vilo: a rapid learning nearest-neighbor classifier for malware triage. *Journal of Computer Virology and Hacking Techniques*, pages 1–15, 2013.

Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.

Felix Leder, Peter Martini, and Andre Wichmann. Finding and extracting crypto routines from malware. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 394–401. IEEE, 2009.

Yibin Liao. Pe-header-based malware study and detection. *Retrieved from the University of Georgia: http://www. cs. uga. edu/˜ liao/PE_Final_Report. pdf*, 2012.

David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

Federico Maggi, Andrea Valdi, and Stefano Zanero. Andrototal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices (SPSM)*, pages 49–54. ACM, November 2013.

Zane Markel and Michael Bilzor. Building a machine learning classifier for malware detection. In *Anti-malware Testing Research (WATeR), 2014 Second Workshop on*, pages 1–4. IEEE, 2014.

Zane A Markel. Machine learning based malware detection. Technical report, DTIC Document, 2015.

Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

Robert Moskovitch, Yuval Elovici, and Lior Rokach. Detection of unknown computer worms based on behavioral classification of the host. *Computational Statistics & Data Analysis*, 52(9):4544–4566, 2008a.

Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. Unknown malcode detection using opcode representation. In *Intelligence and Security Informatics*, pages 204–215. Springer, 2008b.

H Moudni, M Er-rouidi, Mustapha Oujaoura, and O Bencharef. Recognition of amazigh characters using surf & gist descriptors. In *International Journal of Advanced Computer Science and Application. Special Issue on Selected Papers from Third international symposium on Automatic Amazigh processing*, pages 41–44, 2013.

S Murugan and K Kuppusamy. System and methodology for unknown malware attack. In *Sustainable Energy and Intelligent Systems (SEISCON 2011), International Conference on*, pages 803–804. IET, 2011.

Lakshmanan Nataraj, S Karthikeyan, Gregoire Jacob, and BS Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

Nir Nissim, Robert Moskovitch, Lior Rokach, and Yuval Elovici. Novel active learning methods for enhanced pc malware detection in windows os. *Expert Systems with Applications*, 41(13):5843–5857, 2014.

Philip O'Kane, Sakir Sezer, Kieran McLaughlin, and E Im. Svm training phase reduction using dataset feature filtering for malware detection. 2013.

Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic rep-

resentation of the spatial envelope. *International journal of computer vision*, 42(3): 145–175, 2001.

Mila Parkour. contagio, 2016. URL `http://contagiodump.blogspot.com`.

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

Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 300–305. IEEE, 2013.

Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

Matt Pietrek. Peering inside the pe: a tour of the win32 (r) portable executable file format. *Microsoft Systems Journal-US Edition*, pages 15–38, 1994.

Bruce Potter and Greg Day. The effectiveness of anti-malware tools. *Computer Fraud & Security*, 2009(3):12–13, 2009.

Jithu Raphel and P Vinod. Heterogeneous opcode space for metamorphic malware detection. *Arabian Journal for Science and Engineering*, pages 1–22, 2016.

Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

Zahra Salehi, Mahboobeh Ghiasi, and Ashkan Sami. A miner for malware detection based on api function calls and their arguments. In *Artificial Intelligence and Signal Processing (AISP), 2012 16th CSI International Symposium on*, pages 563–568. IEEE, 2012.

Aiman A Abu Samra, Kangbin Yim, and Osama A Ghanem. Analysis of clustering technique in android malware detection. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on*, pages 729–733. IEEE, 2013.

Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Idea: Opcode-sequence-based malware detection. In *International Symposium on Engineering Secure Software and Systems*, pages 35–43. Springer, 2010.

Igor Santos, Felix Brezo, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. Using opcode sequences in single-class learning to detect unknown malware. *IET information security*, 5(4):220–227, 2011a.

Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware

detection. *Information Sciences*, 2011b.

Igor Santos, Javier Nieves, and Pablo G Bringas. Semi-supervised learning for unknown malware detection. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 415–422. Springer, 2011c.

Igor Santos, Jaime Devesa, Felix Brezo, Javier Nieves, and Pablo Garcia Bringas. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS-ICEUTE*, pages 271–280. Springer, 2013.

Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, and Pablo Garcia Bringas. On the automatic categorisation of android applications. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 149–153. IEEE, 2012.

Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, pages 289–298. Springer, 2013a.

Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G Bringas, and Gonzalo Álvarez Marañón. Mama: manifest analysis for malware detection in android. *Cybernetics and Systems*, 44(6-7):469–488, 2013b.

Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001a.

Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001b.

Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*, 14(1):16–29, 2009a.

Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*, 14(1):16–29, 2009b.

Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.

Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1–22, 2012.

M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2009a.

M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2009b.

Raja Khurram Shahzad and Niklas Lavesson. Veto-based malware detection. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 47–54. IEEE, 2012.

Raja Khurram Shahzad, Syed Imran Haider, and Niklas Lavesson. Detection of spyware by mining executable files. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 295–302. IEEE, 2010.

Raja Khurram Shahzad, Niklas Lavesson, and Henric Johnson. Accurate adware detection using opcode sequence extraction. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 189–195. IEEE, 2011.

Syed Zainudeen Mohd Shaid and Mohd Aizaini Maarof. Malware behavior image for malware variant identification. In *Biometrics and Security Technologies (ISBAST), 2014 International Symposium on*, pages 238–243. IEEE, 2014.

CE Shannon. A mathematical theory of communication, bell system technical journal 27: 379-423 and 623–656. *Mathematical Reviews (MathSciNet): MR10, 133e*, 1948.

Shina Sheen, R Anitha, and P Sirisha. Malware detection by pruning of parallel ensembles using harmony search. *Pattern Recognition Letters*, 2013.

Saurabh Anandrao Shivale. Cryptovirology: Virus approach. *arXiv preprint arXiv:1108.2482*, 2011.

Thomas Stibor. A study of detecting computer viruses in real-infected files in the n-gram representation with machine learning methods. In *Trends in Applied Intelligent Systems*, pages 509–519. Springer, 2010.

Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014a.

Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2014b.

G Ganesh Sundarkumar and Vadlamani Ravi. Malware detection by text and data mining. In *Computational Intelligence and Computing Research (ICCIC), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013.

Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

Kabakus Abdullah Talha, Dogru Ibrahim Alper, and Cetin Aydin. Apk auditor: Permission-based android malware detection system. *Digital Investigation*, 13:1–14, 2015.

Botnet Research Team et al. Sanddroid: An apk analysis sandbox. xi'an jiaotong university, 2014.

Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. A malware instruction set for behavior-based analysis. 2009.

Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233. ACM, 2014.

P Vinod, Vijay Laxmi, and Manoj Singh Gaur. Scattered feature space for malware analysis. In *International Conference on Advances in Computing and Communications*, pages 562–571. Springer, 2011.

VirusTotal VirusTotal. Virustotal - free online virus, malware and url scanner, 2004. URL `https://www.virustotal.com/`.

C+ Visual and Business Unit. Microsoft portable executable and common object file format specification, 1999.

Andrew Walenstein, Daniel J Hefner, and Jeffery Wichers. Header information in malware families and impact on automated classifiers. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 15–22. IEEE, 2010.

Jau-Hwang Wang, Peter S Deng, Yi-Shen Fan, Li-Jing Jaw, and Yu-Ching Liu. Virus detection using data mining techinques. In *Security Technology, 2003. Proceedings. IEEE 37th Annual 2003 International Carnahan Conference on*, pages 71–76. IEEE, 2003.

Tzu-Yen Wang, Chin-Hsiung Wu, and Chu-Cheng Hsieh. Detecting unknown malicious executables using portable executable headers. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 278–284. IEEE, 2009.

Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):3, 2016.

Guanhua Yan, Nathan Brown, and Deguang Kong. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013a.

Guanhua Yan, Nathan Brown, and Deguang Kong. Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013b.

Chaitanya Yavvari, Arnur Tokhtabayev, Huzefa Rangwala, and Angelos Stavrou. Malware characterization using behavioral components. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 226–239. Springer, 2012.

Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. An intelligent

pe-malware detection system based on association mining. *Journal in Computer Virology*, 4(4):323–334, 2008.

Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 121–128. IEEE, 2013.

Joel Yonts. Attributes of malicious files. *SANS Institute InfoSec Reading Room*, 2012.

Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300, 2010.

Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 129–140. IEEE, 1996.

Sheng Yu, Shijie Zhou, Leyuan Liu, Rui Yang, and Jiaqing Luo. Detecting malware variants by byte frequency. *Journal of Networks*, 6(4):638–645, 2011.

Hao Zhang, Danfeng Daphne Yao, Naren Ramakrishnan, and Zhibin Zhang. Causality reasoning about network events for detecting stealthy malware activities. *computers & security*, 58:180–198, 2016.

Sami Zhioua. The middle east under malware attack dissecting cyber weapons. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pages 11–16. IEEE, 2013.

List of Publications

List of Publications

Peer Reviewed & Indexed International Journals

1. **Ajit Kumar**, K.S Kuppusamy, and G. Aghila, FAMOUS: Forensic Analysis of MObile Devices Using Scoring of application permissions, Future Generation Computer Systems,2018, Elsevier, https://doi.org/10.1016/j.future.2018.02.001, ISSN: 0167-739X.**Impact Factor: 3.997**

2. **Ajit Kumar**, K.S Kuppusamy, and G. Aghila, A learning model to detect maliciousness of portable executable using integrated feature set, Journal of King Saud University - Computer and Information Sciences,2017, Elsevier, ISSN: 1319-1578.

3. **Ajit Kumar**, K.S Kuppusamy, and G. Aghila,Features for Detecting Malware on Computing Environments, IJEACS Volume : 01, Issue: 02, December 2016 ,Empirical Research Press,UK

4. **Ajit Kumar**, K.S Kuppusamy, and G. Aghila, CASroid: A two level certification framework for building trust in third party android application stores,International Journal of Information Security Science, (provisionally accepted) ISSN: 2147-0030

International Conference Publications

5. **Kumar, Ajit**, and G. Aghila. "Portable executable scoring: What is your malicious score?." Science Engineering and Management Research (ICSEMR), 2014 International Conference on. IEEE, 2014.

6. **Ajit kumar**, G. Aghila, " PoToMAC: A Pre-processing Tool for Malware Classification", In proceedings of ICIIT 2014, Chennai, India.

7. **Ajit Kumar**, Pramod Sagar K, K.S.Kuppusamy,Aghila G.,Machine learning based Malware Classification for Android Applications using Multimodal Image Representations, 10th International Conference on Intelligent Systems and Control (ISCO), 2016

VITAE

Mr. Ajit Kumar, the author of this thesis is a Ph.D full time, research scholar in the Department of Computer Science, School of Engineering and Technology, Pondicherry University. He was born on 7^{th} Feb 1990 at Lakhisarari, Bihar, India.

He has received his Bachelor of Computer Application (BCA) from IGNOU in the year 2009 and Master of Computer Science in the year 2011, from Pondicherry University. With his formal education he has received Post Graduate Diploma in *Statistical and Research Methods* from Pondicherry University in 2015 and Post Graduate Diploma in *Information Security* from IGNOU in 2016. His area of interest includes Information Security, Malware detection and Machine learning.

He qualified UGC-NET for Lecturer exam in 2014, besides UGC-NET he has also qualified three states (Rajasthan, Andhra Pradesh and Tamilnadu) SET (State Eligibility Test) lectureship exam. He is all India first rank holder in Ph.D entrance exam of Pondicherry University in year 2012.

He has published 4 peer-reviewed International Journal papers and presented and published 6 papers in International IEEE and Elsevier conferences.