

MALWARE DETECTION USING HYBRID ANALYSIS

by

VENKATESH P 2015103051
NARAYANAN MR 2015103040
HARIISH N 2015103592

A project report submitted to the

**FACULTY OF INFORMATION AND
COMMUNICATION ENGINEERING**

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

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**

ANNA UNIVERSITY, CHENNAI – 25

APRIL 2019

BONAFIDE CERTIFICATE

Certified that this project report titled **MALWARE DETECTION USING HYBRID ANALYSIS** is the *bonafide* work of **VENKATESH P (2015103051)**, **NARAYANAN MR (2015103040)** and **HARIISH N (2015103592)** who carried out the project work under my supervision, for the fulfillment of the requirements for the award of the degree of Bachelor of Engineering in Computer Science and Engineering. Certified further that to the best of my knowledge, the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or an award was conferred on an earlier occasion on these or any other candidates.

Place: Chennai

Dr.S.Valli

Date:

Professor

Department of Computer Science and Engineering
Anna University, Chennai – 25

COUNTERSIGNED

Head of the Department,
Department of Computer Science and Engineering,
Anna University Chennai,
Chennai – 600025

ACKNOWLEDGEMENT

We express our deep gratitude to our guide, Dr S.Valli for guiding us through every phase of the project. We appreciate her thoroughness, tolerance and ability to share her knowledge with us. We thank her for being easily approachable and quite thoughtful. Apart from adding her own input, she has encouraged us to think on our own and give form to our thoughts. We owe her for harnessing our potential and bringing out the best in us. Without her immense support through every step of the way, we could never have it to this extent.

We are extremely grateful to Dr. S.Valli, Head of the Department of Computer Science and Engineering, Anna University, Chennai²⁵, for extending the facilities of the Department towards our project and for her unstinting support.

We express our thanks to the panel of reviewers Dr. Chitrakala, Dr. Angeline Gladston and Dr. Renugadevi for their valuable suggestions and critical reviews throughout the course of our project.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

Venkatesh P

Narayanan MR

Hariish N

ABSTRACT

The open source Android platform allows developers to take full advantage of the mobile operating system, but also raises significant issues related to malicious applications. Mobile malware is the highest threat to the security of IoT data, users personal information, identity, and corporate/financial information.

We considered static, dynamic, and hybrid detection analysis. In this performance analysis, we compared static, dynamic, and hybrid analyses on the basis of data set, feature extraction techniques, feature selection techniques, detection methods, and the accuracy achieved by these methods. Therefore, we identify suspicious API calls, system calls, and the permissions that are extracted and selected as features to detect mobile malware. This will assist application developers in the safe use of APIs when developing applications for industrial IoT networks.

We propose to combine permission and API (Application Program Interface) calls and use machine learning methods to detect malicious Android Apps. In our design, the permission is extracted from each App's profile information and the APIs are extracted from the packed App file by using packages and classes to represent API calls. By using permissions and API calls as features to characterize each Apps, we can learn a classifier to identify whether an App is potentially malicious or not.

ABSTRACT

TABLE OF CONTENTS

ABSTRACT – ENGLISH	iii
ABSTRACT – TAMIL	iv
LIST OF FIGURES	viii
LIST OF TABLES	ix
LIST OF ABBREVIATIONS	x
1 INTRODUCTION	1
1.1 Problem Domain	1
1.2 Problem Description	2
1.3 Scope	2
1.4 Contribution	3
1.5 SWOT Analysis	5
1.5.1 Strengths	5
1.5.2 Weaknesses	5
1.5.3 Opportunities	6
1.5.4 Threats	6
1.5.5 Organization of Thesis	6
2 RELATED WORK	8
2.1 Identifying Significant Permissions	8
2.2 Malware threats and Detecting Methods	9
2.3 Detecting Malware based on AndroidManifest file	10
2.4 Extracting and selecting features	11
2.5 Detecting Malware using Permissions	12

2.6	Detecting malware Using Permissions and API Calls . . .	12
3	REQUIREMENTS ANALYSIS	14
3.1	Functional Requirements	14
3.2	Non-Functional Requirements	14
3.2.1	User Interface	14
3.2.2	Hardware	15
3.2.3	Software	15
3.3	Dataset Description	15
3.3.1	Constraints and Assumptions	17
3.4	System Models	18
3.4.1	Use case diagram	18
3.4.2	Sequence diagram	19
4	SYSTEM DESIGN	20
4.1	System Architecture	20
4.2	UI Design	23
4.3	Module Design	23
4.3.1	Acquiring APK dataset	23
4.3.2	Preprocessing of data	23
4.3.3	Machine learning and Malware detection	28
5	SYSTEM DEVELOPMENT	30
5.1	Prototype across modules	30
5.2	Static feature extraction Algorithm	31
5.3	Dynamic Feature Extraction Algorithm	32
5.4	Malware Detection Algorithm	33
6	RESULTS AND DISCUSSION	34

6.1	Dataset	34
6.2	Results and Screenshots	34
6.2.1	Static feature extraction	34
6.2.2	Dynamic feature extraction(Generating Events) .	35
6.2.3	Dynamic feature extraction(Extracting Log) . . .	36
6.2.4	Dynamic feature extraction(Log File)	37
6.2.5	Dynamic feature extraction(Extracted System Calls)	38
6.2.6	Dynamic feature extraction(Generating Vector) .	39
6.2.7	Dynamic feature extraction(Generated Vector) . .	40
6.2.8	KNN	41
6.2.9	SVM	44
6.2.10	Final Accuracy	46
6.3	Evaluation parameters	47
6.4	Testcases	49
6.4.1	Comparison between algorithms	50
7	CONCLUSION	52
7.1	Contributions	52
7.2	Summary	54
7.3	Future Work	54
	REFERENCES	56

LIST OF FIGURES

3.1	Use Case Diagram	18
3.2	Sequence Diagram	19
4.1	System Architecture	22
4.2	Static Feature Extraction	25
4.3	Dynamic Feature Extraction	26
4.4	Feature selection	27
4.5	Machine learning	28
6.1	Static Feature Extraction	35
6.2	Dynamic Feature Extraction(Generating events)	36
6.3	Dynamic Feature Extraction(Extraction Log)	37
6.4	Dynamic Feature Extraction(Log File)	38
6.5	Dynamic Feature Extraction(Extracted System Calls)	39
6.6	Dynamic Feature Extraction(Generating Vector)	40
6.7	Dynamic Feature Extraction(Generating Vector)	41
6.8	KNN Accuracy	42
6.9	KNN Accuracy(With PCA)	43
6.10	SVM train test	44
6.11	SVM Accuracy	45
6.12	Final Accuracy	46
6.13	Testcases	50
6.14	Accuracy of different algorithms	51

LIST OF TABLES

6.1	Evaluation Metrics	48
6.2	Testcase results	49

LIST OF ABBREVIATIONS

KNN	K-Nearest Neighbours
SVM	Support Vector Machine
DL	Deep Learning
ML	Machine Learning
API	Application programming
GPU	Graphical Processing Unit
APK	Android application package
PID	Permission identity
PCA	Principal Component Analysis
MD	Malware Detection

CHAPTER 1

INTRODUCTION

1.1 PROBLEM DOMAIN

The open source Android platform allows developers to take full advantage of the mobile operating system, but also raises significant issues related to malicious applications. On one hand, the popularity of Android absorbs attention of most developers for producing their applications on this platform. The increased numbers of applications, on the other hand, prepares a suitable prone for some users to develop different kinds of malware and insert them in Google Android market or other third party markets as safe applications. Malware has become more harmful than in the past as the number of intelligent systems and Internet-connected devices increased dramatically. Therefore one of the most important issues in cyber security has become the detection of previously unknown malware in the shortest time possible in order to stop it from becoming epidemic and from harming users. [3] The most significant issue is the emergence of traditional malware such as viruses, worms, Trojan horses, and rootkits. Malicious software in this context behaves similarly to the same threats on traditional IT networks. In this case malware may be targeted at ex-filtrating sensitive data from the mobile platform or further leveraging the compromised asset to access sensitive industrial/critical infrastructures in IoT networks in which the mobile device has legitimate access to the corresponding IoT devices. Second, developers are beginning to offer software for free and instead

generate revenue with data analytics and marketing which can invade a users privacy. [6]

1.2 PROBLEM DESCRIPTION

The sensitivity of mobile platforms and their potential for abuse, several security issues not too dissimilar to those already affecting traditional IT network counterparts are beginning to surface. The most significant issue is the emergence of traditional malware such as viruses, worms, Trojan horses, and rootkits. Malicious software in this context behaves similarly to the same threats on traditional IT networks. Mobile malware is the highest threat to the security of IoT data, users personal information, identity, and corporate/financial information. We considered static, dynamic, and hybrid detection analysis. In this performance analysis, we compared static, dynamic, and hybrid analyses on the basis of data set, feature extraction techniques, feature selection techniques, detection methods, and the accuracy achieved by these methods. Therefore, we identify suspicious API calls, system calls, and the permissions that are extracted and selected as features to detect mobile malware. [7]

1.3 SCOPE

Detection of malware is a crucial task, as malware developers hide their malicious activities and introduce new methods to avoid detection. Anti-malware software must cope with new technologies such as code obfuscation, mimicry attacks etc. For Windows and other operating systems, use of resources is not a critical concern, whereas for the operating system of mobile devices resource usage is always a concern. Limited use of resources means that the process of malware detection is not a straightforward problem for the mobile platform. Detecting the malicious activities of mobile applications using limited resources within a

limited time period is a challenge to researchers.

A wide range of Android operating systems are used. Old versions of Android may suffer from different security issues. Additionally, many smartphone and tablet vendors run third party apps markets which may act as a source of malicious applications. People from China and Asia choose to download apps in their local languages which are available at third party apps store and are a potential source of malware. Crowdsourcing is used for the prevention of malware in Android. [4] However, fake reviews from users can be a security threat. Joining a site such as Admob is comparatively easier than joining iAd as no identity proof is required. This encourages the ad-based malware developers. These security threats can cause issues like personal data leakage, social, business, financial loss etc.

1.4 CONTRIBUTION

Researchers have used static, dynamic, and hybrid processes to detect mobile malware and malicious activities. Researchers primary concerns involve accuracy levels, and most the research papers describe the performance of their detection process using accuracy metrics. [1] For the operating system of mobile devices, performance overhead should be considered, as higher accuracy may cause higher overhead. Accuracy and performance overhead need to be well balanced to make the detection process efficient.

The static feature is formed by analyzing the structure and format of the sample and then extracting the hash value, string information, function information, header file information, and resource description information. The technology obtains most of the malware information from the malware itself, thus the analysis results are relatively comprehensive. However, static features cannot correctly discriminate malware

when the static information is packed or obfuscated or compressed [5], making it difficult for static features to express the true purpose of malware, thus affecting the accuracy of detection.

Dynamic features are the behavior of the sample execution and the features of the debug record, such as file operations, the creation and deletion of processes, and other dynamic behaviors. Since the malicious behaviors of malware at dynamic runtime can't be concealed, the extracted dynamic features provide a more realistic description than the static features. However, the extraction of dynamic features needs to be run in a virtual environment [2], which will be reset and restored to the previous state after each malicious sample has been analyzed to ensure that the virtual environment is a real user environment. As a result, features extraction efficiency is much lower than for static features.

We analyze the ongoing research efforts covering the three basic categories: static, dynamic, and hybrid analysis. These analyses represent the data set, features, feature selection method, detection method, and the accuracy. We also have mentioned the literature gap and the limitations of current research efforts. Thereby we have identified the suspicious feature lists which are commonly used by malware developers. The main contributions of this research effort are divided into the following main areas:

- The individual entries are indicated with a black dot, a so-called bullet.
- The text in the entries may be of any length.
- Defining the mobile malware detection process for IoT networks.
- Determining the security limitations for mobile platforms in industrial IoT networks.
- A comparative analysis of static, dynamic, and hybrid detection processes and their limitations and scopes.

- Identifying the suspicious permission, API call, and system call lists to enable IoT application developers in the safe use of APIs.

We propose to combine permission and API (Application Program Interface) calls and use machine learning methods to detect malicious Android Apps. In our design, the permission is extracted from each App's profile information and the APIs are extracted from the packed App file by using packages and classes to represent API calls. By using permissions and API calls as features to characterize each Apps, we can learn a classifier to identify whether an App is potentially malicious or not.

1.5 SWOT ANALYSIS

1.5.1 Strengths

The Support Vector machine and K-Nearest Neighbours are used to train the system which provides higher accuracy when compared to other machine learning and deep learning models. To the KNN and SVM, we have used Principal component analysis is used to give better results. This helps in reducing the dimensions and increase the accuracy

1.5.2 Weaknesses

The model is quite heavy and requires a lot of processing power for training/testing. Available datasets for network traffic are not very large or descriptive, which affects accuracy. In the KNN, we have not implemented procedures for firewall protection, load description and handling, DDoS attacks etc. The rules for forwarding and dropping packets are described by us, the developers, and are therefore subject to human error.

1.5.3 Opportunities

Further work can be done in anomaly-based Network Intrusion Detection Systems (NIDS), firewall protection, load description and handling, and their respective Northbound and Southbound APIs. More research into this domain can help improve accuracy and handle various types of attacks.

1.5.4 Threats

The project can only work with network packets. Adversarial malware on attacks can bypass the Discriminator classification schema, and has yet to be accounted for. Zero-day DDoS attacks can potentially cause network failure, as can human errors in flow table handling.

1.5.5 Organization of Thesis

This chapter consists of the introduction which includes the problem domain, problem description, scope, contribution, SWOT analysis. The next chapter includes the literature survey which was performed on the project relevant topics. It includes a concept wise survey on the various ideologies that were put forth in the project.

Chapter 3 consists of the requirement analysis which includes functional and non-functional requirements where non-functional requirements brief us upon the user interface, hardware and software. This chapter also gives a touch up on the dataset description, usecase and sequence diagrams.

Chapter 4 consists of the system architecture which includes the block diagram and its detailed description. This chapter also includes UI design and module wise description of the block diagram.

Chapter 5 consists of the system development which includes the prototype across the modules and the various malware detection algo-

rithms which were used for the project.

Chapter 6 consists of the results of the various modules that were discussed in the previous chapters. This includes screenshots of the various stages in the project. In the end of the chapter the evaluation metrics are also discussed. Chapter 7 consists of the conclusion of the project and the future works.

CHAPTER 2

RELATED WORK

2.1 IDENTIFYING SIGNIFICANT PERMISSIONS

J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an and H. Ye used SVM and a small dataset to test the proposed MLDP model. The SVM determines a hyperplane that separates both classes with a maximal margin based on the training dataset that includes benign and malicious applications. In this case, one class is associated with malware, and the other class is associated with benign apps. Then, we assume the testing data as unknown apps, which are classified by mapping the data to the vector space to decide whether it is on the malicious or benign side of the hyperplane. Then, we can compare all analysis results with their original records to evaluate the malware detection correctness of the proposed model by using the SVM. The problems that were encountered in this methodology was that the algorithm uses only static features, So the efficiency is based on these features alone. It does predict the malware with much accuracy when dynamic features are considered along with static features. The ideas that were adopted from this paper include static features used for malware detection and efficient feature selection as no app requests all the permissions, and the ones that an app requests are listed in the Android application package (APK) as part of manifest.xml. Three levels of data pruning methods were proposed to filter out permissions that contribute little to the malware detection effectiveness. Thus, they can be safely removed without negatively affecting malware detection accu-

racy. The results indicate that when a support vector machine is used as the classifier, we can achieve over 90% of precision, recall, accuracy, and F-measure, which are about the same as those produced by the baseline approach while incurring the analysis times that are 432 times less than those of using all permissions.

2.2 MALWARE THREATS AND DETECTING METHODS

S. Sharmeen, S. Huda, J. H. Abawajy, W. N. Ismail and M. M. Hassan analyze the efforts regarding malware threats aimed at the devices deployed in industrial mobile-IoT networks and related detection techniques. We considered static, dynamic, and hybrid detection analysis. In this performance analysis, we compared static, dynamic, and hybrid analyses on the basis of data set, feature extraction techniques, feature selection techniques, detection methods, and the accuracy achieved by these methods. Therefore, we identify suspicious API calls, system calls, and the permissions that are extracted and selected as features to detect mobile malware. This will assist application developers in the safe use of APIs when developing applications for industrial IoT networks. The detection method using K-nearest Neighbor (KNN) achieves best performance in accuracy. Random Forest (RF) and Support Vector Machine (SVM) are mostly used as detection methods and also exhibit high accuracy. The problems that were encountered in this methodology was that it fails to detect unknown/new variant of malware. There is also a need for selection of proper feature set to incorporate unknown behaviours. It also fails to predict the behaviour of malicious apps from their past actions. The ideas that were adopted from this paper include comparing and contrasting static, dynamic, and hybrid analyses on the basis of data set, feature extraction techniques, feature selection techniques, detection methods, and the accuracy achieved by these meth-

ods. Therefore, we identify suspicious API calls, system calls, and the permissions that are extracted and selected as features to detect mobile malware. This will assist application developers in the safe use of APIs when developing applications for industrial IoT networks.

2.3 DETECTING MALWARE BASED ON ANDROIDMANIFEST FILE

X. Li, J. Liu, Y. Huo, R. Zhang and Y. Yao found that the statistical information of Android components (mainly activity) from the Manifest file cannot be ignored, based on the traditional method of Android permission detection. In this paper, a new feature vector is extracted from the AndroidManifest file, which combines the permission information and the component information of the Android application. We combine the naive Bias classification algorithm, and propose a malicious application detection method based on AndroidManifest file information. The experimental results show that the new method performance better than that of the traditional permission detection. The problems that were encountered in this methodology was that the algorithm uses only static features, So the efficiency is based on these features alone. It does predict the malware with much accuracy when dynamic features are considered along with static features. The ideas that were adopted from this paper include the malware detection method based on AndroidManifest file. Sample statistical analysis results showed that the component information in the AndroidManifest file shows different statistical distribution patterns in different kinds of applications. The experimental results show that the new method performance better than that of the traditional permission detection.

2.4 EXTRACTING AND SELECTING FEATURES

K. Zhao, D. Zhang, X. Su and W. Li proposed to use Feature Extraction and Selection Tool (Fest), a feature-based machine learning approach for malware detection. We first implement a feature extraction tool, AppExtractor, which is designed to extract features, such as permissions or APIs, according to the predefined rules. Then we propose a feature selection algorithm, FrequenSel. Unlike existing selection algorithms which pick features by calculating their importance, FrequenSel selects features by finding the difference their frequencies between malware and benign apps, because features which are frequently used in malware and rarely used in benign apps are more important to distinguish malware from benign apps. In experiments, we evaluate our approach with 7972 apps, and the results show that Fest gets nearly 98% accuracy and recall, with only 2% false alarms. Moreover, Fest only takes 6.5s to analyze an app on a common PC, which is very time-efficient for malware detection in Android markets. The problems that were encountered in this methodology was that the works are inefficient due to lack of feature selection, which also results in the imbalance between accuracy and recall, and time overhead in building classifiers. The ideas that were adopted from this paper include extracting all the features which indicate the functions and behaviors in apps by AppExtractor, and an algorithm FrequenSel to select typical features which help distinguish malware from benign apps. FrequenSel selects features by finding the difference their frequencies between malware and benign apps, because features which are frequently used in malware and rarely used in benign apps are more important to distinguish malware from benign apps.

2.5 DETECTING MALWARE USING PERMISSIONS

Aung, Zarni and Zaw, Win proposed a framework that intends to develop a machine learning-based malware detection system on Android to detect malware applications and to enhance security and privacy of smartphone users. This system monitors various permission based features and events obtained from the android applications, and analyses these features by using machine learning classifiers to classify whether the application is benignware or malware. The performances of machine learning techniques were evaluated using the true positive rate, false positive rate and overall accuracy. The problems that were encountered in this methodology was that the algorithm uses only static features, So the efficiency is based on these features alone. It does predict the malware with much accuracy when dynamic features are considered along with static features. The ideas that were adopted from this paper include static features used for malware detection and efficient feature selection as no app requests all the permissions, and the ones that an app requests are listed in the Android application package (APK) as part of manifest.xml

2.6 DETECTING MALWARE USING PERMISSIONS AND API CALLS

N. Peiravian and X. Zhu proposed to use the permission extracted from each App's profile information and the APIs are extracted from the packed App file by using packages and classes to represent API calls. By using permissions and API calls as features to characterize each Apps, we can learn a classifier to identify whether an App is potentially malicious or not. An inherent advantage of our method is that it does not need to involve any dynamical tracing of the system calls but only uses simple static analysis to find system functions involved in each App. In addition, because permission settings and APIs are always

available for each App, our method can be generalized to all mobile applications. Experiments on real-world Apps with more than 1200 malware and 1200 benign samples validate the algorithm performance. The problems that were encountered in this methodology was that it fails to detect unknown/new variant of malware. There is also a need for selection of proper feature set to incorporate unknown behaviours. It also fails to predict the behaviour of malicious apps from their past actions. The ideas that were adopted from this paper include the permission extracted from each Apps profile information and the APIs are extracted from the packed App file by using packages and classes to represent API calls. By using permissions and API calls as features to characterize each Apps. The proposed framework extracts permissions from Android applications and further combines the API calls to characterize each application as a high dimension feature vector.

CHAPTER 3

REQUIREMENTS ANALYSIS

3.1 FUNCTIONAL REQUIREMENTS

The system outputs a detection of Malware for a given set of APK dataset. The output of the detection should adhere to the following requirements:

- The output should be able to detect the malware based on the input dataset
- The system must be optimized for time and space complexities
The system must
- The system should be able to detect unknown/new variant of malware.
- The system should be able to predict the behaviour of malicious apps from their past actions.
- The system should be able to select proper feature set so as to incorporate unknown behaviours
- The system must be able to detect malware from any domain

3.2 NON-FUNCTIONAL REQUIREMENTS

3.2.1 User Interface

There must be a simple and easy to use system where the user should be able to detect malware using the APK dataset. Machine learning is used to build the model using the selected features as input. Comparative analysis is done and a classification report is generated.

3.2.2 Hardware

No special hardware interface is required for the successful implementation of the system

3.2.3 Software

- Tools: Anaconda3, Android Studio
- Programming Language: Python
- Operating System: Windows/Linux/Macintosh
- Dataset: APK dataset

3.3 DATASET DESCRIPTION

The first component of the malware detection system is the data set. The data set represents a representative collection of malware and benign ware. A proper data set is essential for analysis of the behavior of the malware. We need examples of both the malware and the benign apps for proper detection. The most common sources of Android Mobile Malware are the Genome project, Contagio, DREBIN data set, Virus Share, etc. The most widely used sources for known benign apps are Google Play, App China, Amazon, Android Market, etc. Malwares can hide their malicious activities and be available in Android market places. Android apps are available in both official and third-party apps market places. They are also good sources of Android applications which can be used as the data set in the detection engine.

We have a list of APK files that appeared on the Internet from January to April 2017. From this dataset, we select over 100,000 clean samples and 100,000 malicious samples randomly with uniform distribution. We take the classification of Avast Software anti-virus as our ground truth to decide whether given sample is malicious or not.

Taking decision of an anti-virus as the gold standard has, of course,

significant disadvantages. Most notably we can have some samples in our sample set wrongly classified. This could be quite a problem for machine learning. However, a decision of commercial anti-virus program seems accurate enough and overall false classification rate is reasonably low.

We have decided to choose decision of single anti-virus (and not for example majority vote over all available anti-viruses on VirusTotal) for the consistency. Different anti-viruses can have different policies what we should classify as malware (e.g. whether the application that gathers personal data and sends them to a server is malicious or still considered clean). Considering a combination of anti-viruses could very well confuse our machine learning algorithm.

In general, we believe that the advantages of this approach can easily outweigh the disadvantages. The most significant benefit of this method is that we can take large dataset for training our machine learning algorithm. Classifying every single sample by hand could take a lot of time, taking an automated decision allows us to work with hundreds of thousands of samples. Furthermore, we do not filter out input samples, and the distribution of our dataset resembles the real distribution of applications on the Internet (separately on clean set and malware set). For example, we may tend to choose samples which all anti-viruses classified into the same category to avoid incorrectly classified input. However, by this method, we would restrict ourselves to samples that are easy to classify and hence the measured result would not reflect the result we would get on a real-world dataset.

We have gathered 118,107 clean samples and 116,608 malware samples. To get an even better simulation of use of our classifier we have decided to take samples collected in first two months as the training set and the rest of them as the test set. The split after two months was

chosen in order to keep both training and test set sufficiently large. After this division, we end up with the training set of size 145,030 samples with ratio 91,968 to 53,062 (clean to malware) and test set consisting of 89,685 samples 26,139 out of them are clean and remaining 63,546 are malware.

We manually ensured we get (roughly) 1:1 ratio of clean and malware samples which of course does not reflect the real distribution. We did this for a couple of reasons. One of them is that we are not quite sure what the actual distribution is (though we assume that clean samples heavily outweigh malicious ones). Also if we had samples with the real distribution only small fraction of them would be malicious, and hence our machine learning algorithm would have only a small set of malware samples to learn from. This is a serious problem because the classifier would probably end up classify everything as clean or heavily overfit due to the small number of malware samples in the training set.

3.3.1 Constraints and Assumptions

Constraints

The model will only work on Windows Portable Executables (PEs). There are around 34,995 and 19,696 malicious and benign files respectively. The number of available adversarial examples is limited; restricting the robustness of the network.

Assumptions

The input training files are correctly labeled, i.e. there can be no misclassified PEs. Each input file has a Windows API call. Processor and memory requirements are met beforehand.

3.4 SYSTEM MODELS

3.4.1 Use case diagram

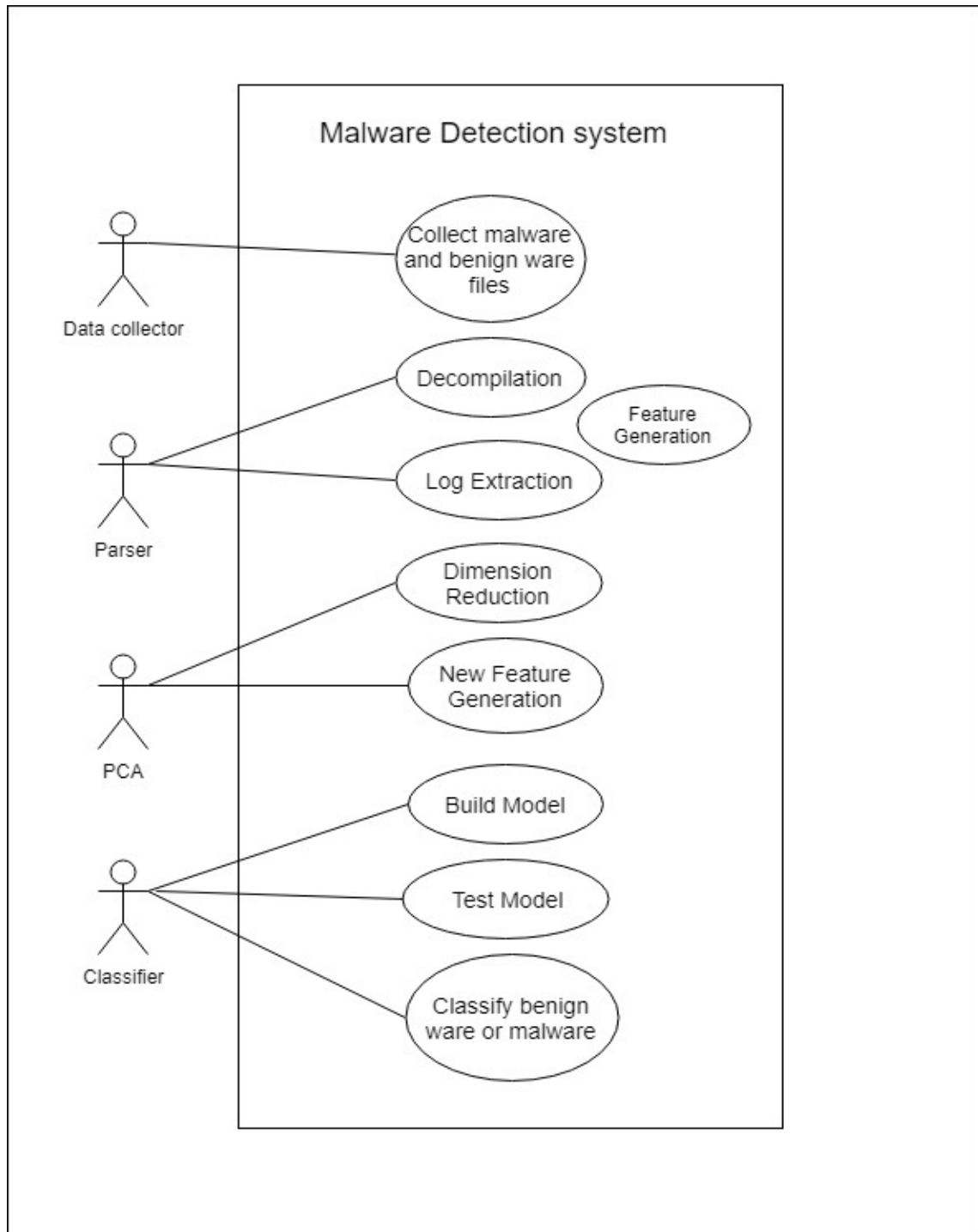


Figure 3.1 Use Case Diagram

3.4.2 Sequence diagram

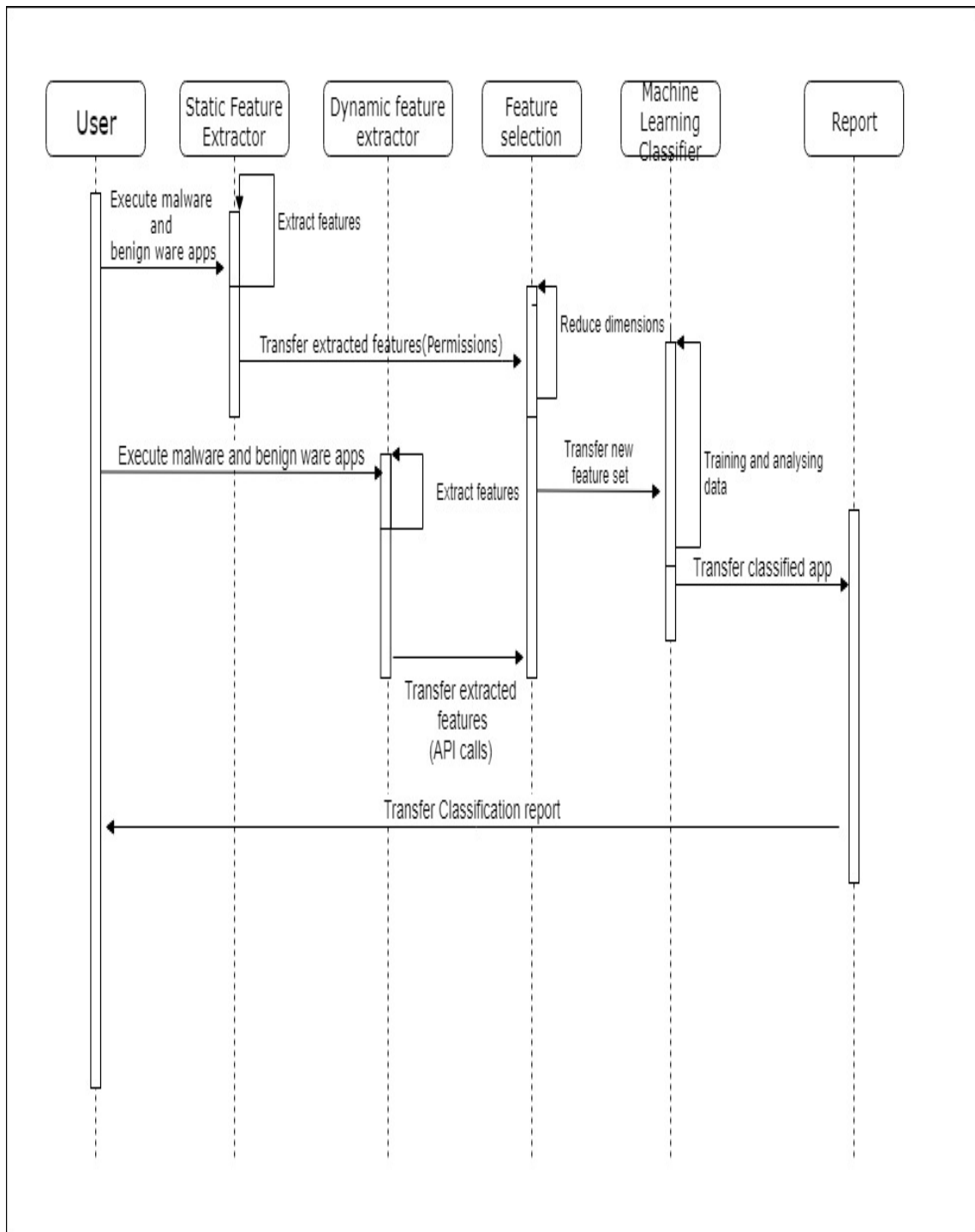


Figure 3.2 Sequence Diagram

CHAPTER 4

SYSTEM DESIGN

4.1 SYSTEM ARCHITECTURE

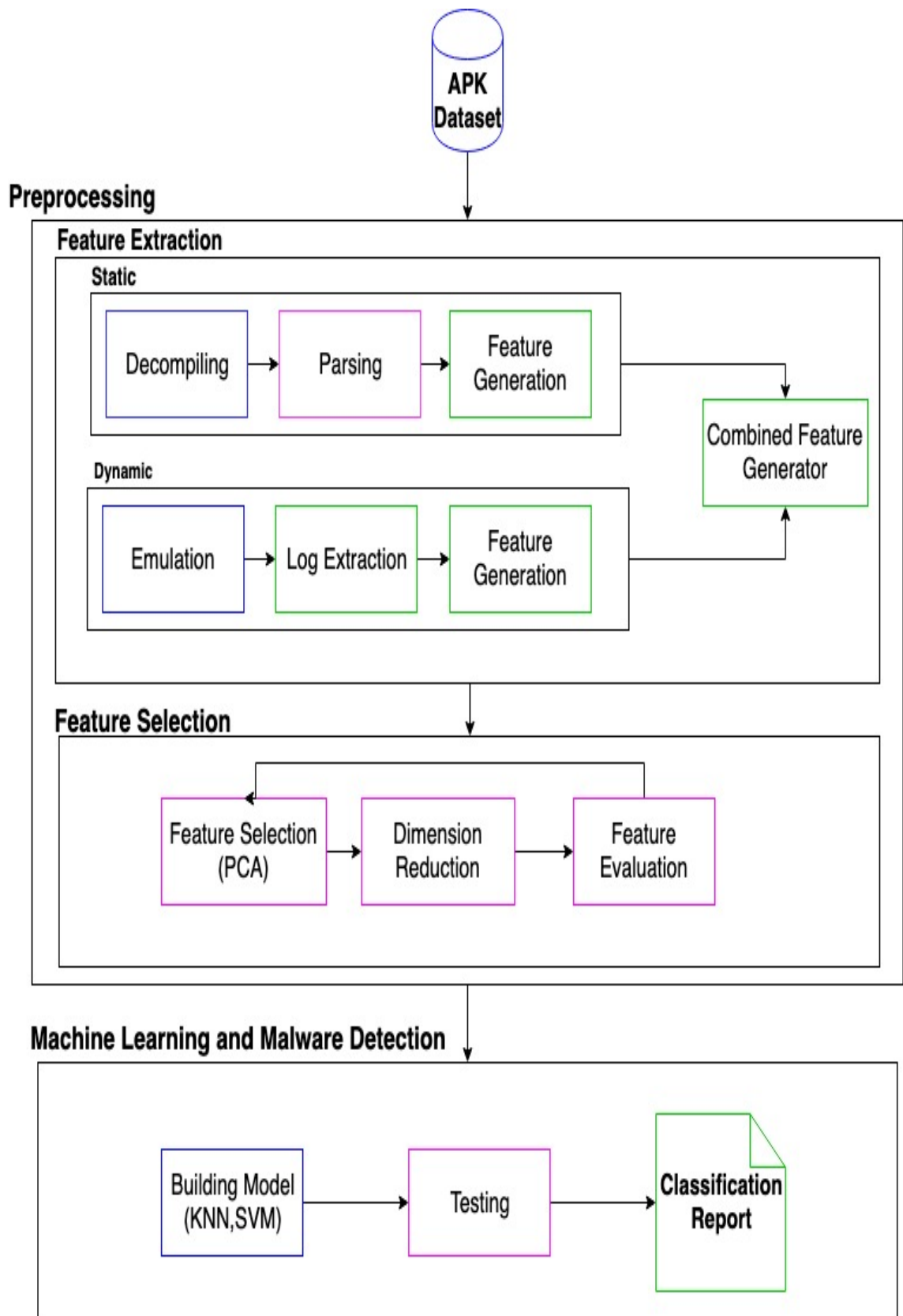
The first component of the malware detection system is the data set. The data set represents a representative collection of malware and benign ware. A proper data set is essential for analysis of the behavior of the malware. We need examples of both the malware and the benign apps for proper detection. The most common sources of Android Mobile Malware are the Genome project, Contagio, DREBIN data set, Virus Share, etc. The most widely used sources for known benign apps are Google Play, App China, Amazon, Android Market, etc. Malwares can hide their malicious activities and be available in Android market places.

The feature Extractor is the component which extracts the desired features from the malware and benign apps. The feature extractor can extract features from the manifest file, dex file, byte code, and log file. Based on the type of extracted feature set, we can categorize the feature extractor as a static feature extractor, dynamic feature extractor, or hybrid feature extractor. Some feature extractors extract features from a single category and some extract features from multiple categories. Static features can be extracted from the manifest file, dex file, and byte code. The most widely used tool is the APK tool. From the APK tool, we obtain the APK file, Manifest file, classes.dex, and Smali file. From these files, we can extract feature. The log file is generated by execut-

ing the application in a virtual machine. From the data set, we collect the malware and benign ware and execute these in the virtual machine, which generates the run time log file. From the log file, we can extract the dynamic features.

The feature selection includes sending the features that we have extracted from the dataset into a PCA model. In principal component analysis, this relationship is quantified by finding a list of the principal axes in the data, and using those axes to describe the dataset. Using Scikit-Learn's PCA estimator, The fit learns some quantities from the data, most importantly the "components" and "explained variance". To see what these numbers mean, let's visualize them as vectors over the input data, using the "components" to define the direction of the vector, and the "explained variance" to define the squared-length of the vector. Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance.

The detection method or the classifier is used to determine whether an app is malware or not. Based on the features, the classifier classifies apps as malware or benign ware. Most classifiers use machine learning. Classifiers based on machine learning use one or multiple classifiers. Layered classifiers may also be used in the detection process. In this case there are two or three layers and each layer contains a classifier to improve the accuracy of the detection system. In parallel classifiers, there are various individual classifiers. The outputs of these classifier are combined to obtain higher accuracy. Other classifiers such as AHP and penalty calculation do not use the machine learning approach. Common machine learning classifiers used in Android mobile malware detection are: SVM, KNN.

**Figure 4.1** System Architecture

4.2 UI DESIGN

The simple and easy way to use User Interface consists of the Windows Command Line where commands are used to execute tcp dump files for each module, starting from the generation of the feature set from the dataset to the detection of malware which is then processed into connection records. On executing each individual module, the intermediate results are obtained, where the output of each module is fed as input to the next module.

4.3 MODULE DESIGN

4.3.1 Acquiring APK dataset

The first module of the malware detection system is the data set. The data set represents a representative collection of malware and benign ware. A proper data set is essential for analysis of the behavior of the malware. Examples of both the malware and the benign apps for proper detection are taken under consideration. The most common sources of Android Mobile Malware are the Genome project, Contagio, DREBIN data set, Virus Share. The most widely used sources for known benign apps are Google Play, App China, Amazon, Android Market. Contagio360.zip and Malware1260.zip were some of the datasets used. The APK dataset is used as input to this system for the training and prediction process.

4.3.2 Preprocessing of data

Feature Extraction

To this module the APK dataset we have obtained from last module is given as input. The feature Extractor is the component which extracts the desired features from the malware and benign apps. The mechanism

that extracts the static features is called a Static Feature Extractor. Static features can be extracted from the manifest file, dex file, and byte code. The most widely used tool is the APK tool. From the APK tool, we obtain the APK file, Manifest file, classes.dex, and Smali file. We perform reverse engineering on the dataset followed by parsing to extract the static features. The mechanism that extracts the dynamic features is called the Dynamic Feature Extractor. The log file is generated by executing the application in a virtual machine. From the data set, we collect the malware and benign ware and execute these in the virtual machine, which generates the run time log file. From the log file, we can extract the dynamic features. We now combine the static and dynamic features to obtain the combined feature set which is further used for feature selection.

The feature Extractor is the component which extracts the desired features from the malware and benign apps. The feature extractor can extract features from the manifest file, dex file, byte code, and log file. Based on the type of extracted feature set, we can categorize the feature extractor as a static feature extractor, dynamic feature extractor, or hybrid feature extractor. Some feature extractors extract features from a single category and some extract features from multiple categories. The features that can be extracted by not executing the application are called static features. The mechanism that extracts the static features is called a Static Feature Extractor. Researchers may consider only a single category of static features and others consider a set of multiple static features. Common static features are: Permission, API call, String extracted, Native commands, XML elements, Meta data, Intents, Broadcast receivers, Hardware components, etc. Static features can be extracted from the manifest file, dex file, and byte code. The most widely used tool is the APK tool. From the APK tool, we obtain the APK file,

Manifest file, classes.dex, and Smali file. From these files, we can extract features.

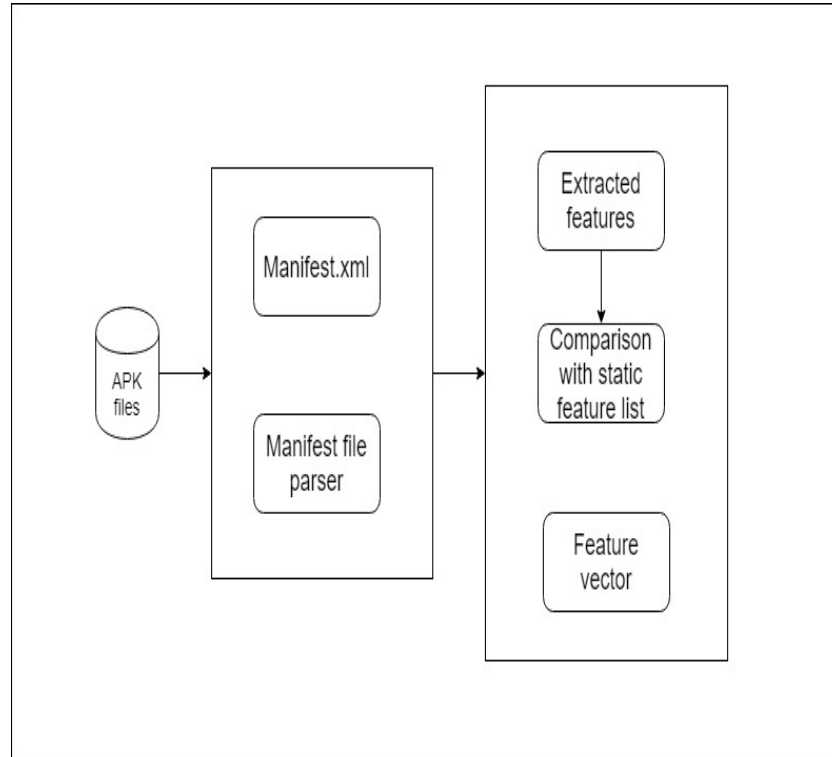


Figure 4.2 Static Feature Extraction

The features that can be extracted from an application by executing it are called dynamic features. The mechanism that extracts the dynamic features is called the Dynamic Feature Extractor. As the static features are not able to represent the full characteristic of an app, researchers use dynamic features, some researchers consider only a single category of dynamic features and others consider a set of multiple dynamic features. Common dynamic features are: System call, Network traffic, SMS, Process id, Process information, Memory usage, IP address, Log events, Power consumption, System component, User interaction, etc. The log file is generated by executing the application in a virtual machine. From the data set, we collect the malware and benign ware and execute these in the virtual machine, which generates the run time log file. From the

log file, we can extract the dynamic features. The features are extracted from static and dynamic and a combined feature vector is produced.

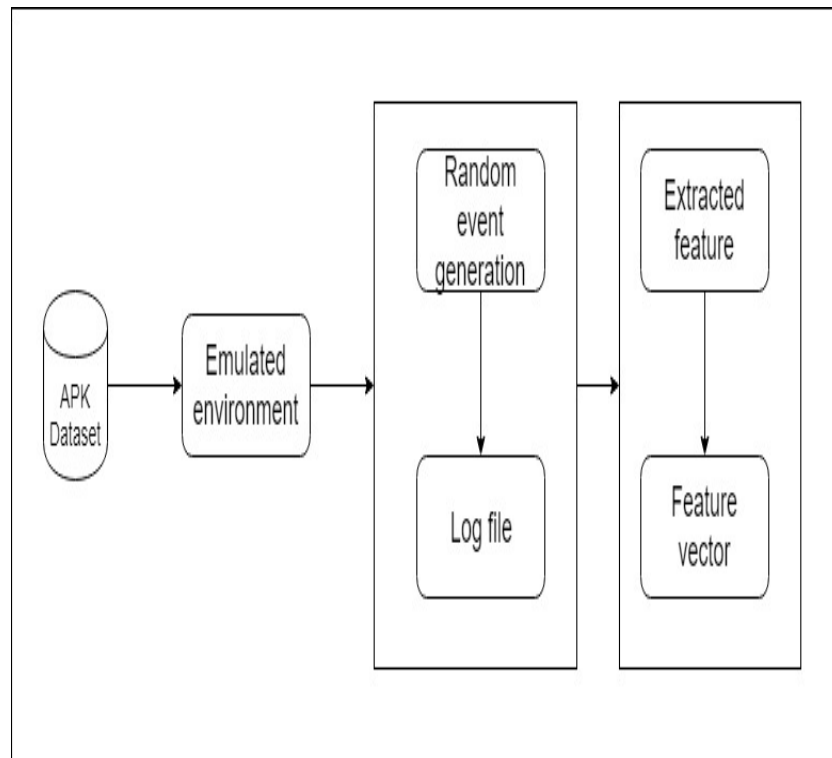


Figure 4.3 Dynamic Feature Extraction

Feature Selection

To this module the extracted features we have obtained from last module is given as input. There are many features, but we need to select those among them which will provide better accuracy in the classification process. We use the principal component analysis algorithm here to select the features that are needed for the detection of malware. The first step in this algorithm involves feature selection. In principal component analysis, this relationship is quantified by finding a list of the principal axes in the data, and using those axes to describe the dataset. Using Scikit-Learn's PCA estimator, The fit learns some quantities from the data, most importantly the "components" and "explained variance". To

see what these numbers mean, let's visualize them as vectors over the input data, using the "components" to define the direction of the vector, and the "explained variance" to define the squared-length of the vector. Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of the data that preserves the maximal data variance. We then perform feature evaluation to evaluate the significance of these ranked features. In the end relevant features that are required machine learning phase is obtained.

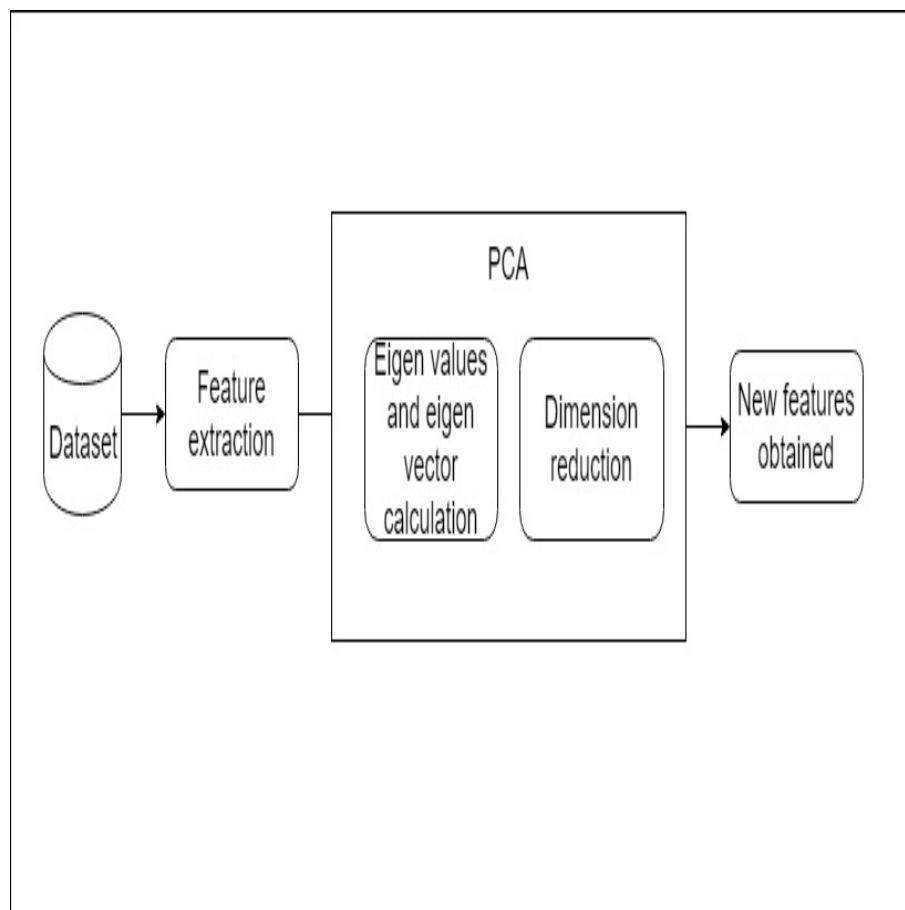


Figure 4.4 Feature selection

4.3.3 Machine learning and Malware detection

Building the model

To this module the selected features we have obtained from last module is given as input. The detection method or the classifier is used to determine whether an app is malware or not. Based on the features, the classifier classifies apps as malware or benign ware. Most classifiers use machine learning. Classifiers based on machine learning use one or multiple classifiers. Layered classifiers may also be used in the detection process. We now build the model using KNN, SVM etc. classification algorithms.

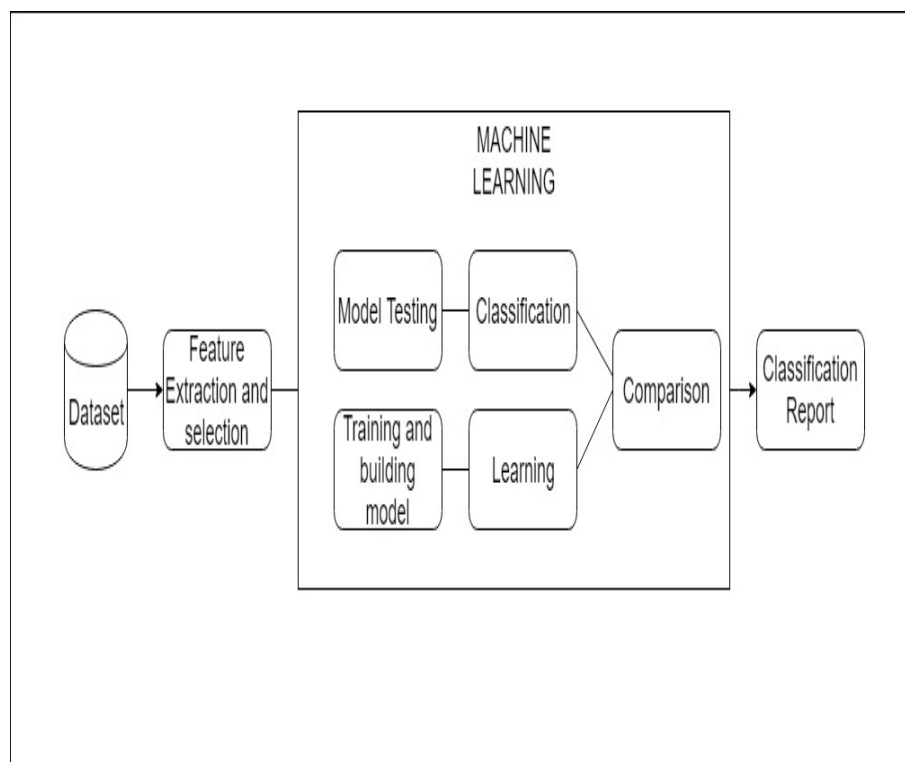


Figure 4.5 Machine learning

Testing

We first split the dataset into the train dataset and test dataset. We use the train dataset to build the model. Now after building the model we use the test dataset which will be upto 20% of the dataset to see if the model is trained properly and providing almost accurate results.

Classification report

We first calculate the performance metrics of our model. The accuracy produced by the model is therefore recorded in the report. After which we perform a comparative analysis has been carried out based on static, dynamic, and hybrid analysis for the Android mobile malware detection process. We evaluate them considering the number of used malware, feature extraction process, selected features, detection method, and accuracy. In the end of this we classified applications as output.

CHAPTER 5

SYSTEM DEVELOPMENT

5.1 PROTOTYPE ACROSS MODULES

The input and output to each module of the system is described in this section.

- **APK Dataset:** The first module of the malware detection system is the data set. The data set represents a representative collection of malware and benign ware. A proper data set is essential for analysis of the behavior of the malware. Examples of both the malware and the benign apps for proper detection are taken under consideration. Contagio360.zip and Malware1260.zip were some of the datasets used. The APK dataset is used as input to this system for the training and prediction process.
- **Feature Extraction:** To this module the APK dataset we have obtained from last module is given as input. The feature Extractor is the component which extracts the desired features from the malware and benign apps. The features that can be extracted from an application by executing it are called dynamic features. We now combine the static and dynamic features to obtain the combined feature set which is further used for feature selection.
- **Feature Selection:** To this module the extracted features we have obtained from last module is given as input. There are many features, but we need to select those among them which will provide better accuracy in the classification process. We use the princi-

pal component analysis algorithm here to select the features that are needed for the detection of malware. We then perform feature evaluation to evaluate the significance of these ranked features. In the end relevant features that are required machine learning phase is obtained.

- **Building the model:** To this module the selected features we have obtained from last module is given as input. The detection method or the classifier is used to determine whether an app is malware or not.
- **Testing:** We first split the dataset into the train dataset and test dataset. We use the train dataset to build the model. Now after building the model we use the test dataset which will be upto 20% of the dataset to see if the model is trained properly and providing almost accurate results.
- **Classification report:** We first calculate the performance metrics of our model. The accuracy produced by the model is therefore recorded in the report. After which we perform a comparative analysis has been carried out based on static, dynamic, and hybrid analysis for the Android mobile malware detection process. We evaluate them considering the number of used malware, feature extraction process, selected features, detection method, and accuracy. In the end of this we classified applications as output.

5.2 STATIC FEATURE EXTRACTION ALGORITHM

The static feature algorithm is given below. It outputs static features from the APK Dataset. The extracted features are then compared to the a list of all permissions and a vector is generated based on that.

1. permission_vector initialised to size (master_permission_list length, APK_count)

2. for each app in app_directory
3. decompiled_APK = decompile(current_APK)
4. current_APK_permissions = parse(decompiled_APK)
5. for each master_permission in master_permission_list
6. for each current_permission in current_APK_permissions
7. if(master_permission == current_permission)
8. permission_vector[app_index, master_permission_index]=1
9. else
10. permission_vector[app_index, master_permission_index]=0

5.3 DYNAMIC FEATURE EXTRACTION ALGORITHM

The static feature algorithm is given below. It outputs static features from the APK Dataset. The log file is generated by executing the application in a virtual machine. From the data set, we collect the malware and benign ware and execute these in the virtual machine, which generates the run time log file. From the log file, we can extract the dynamic features.

1. system_calls_vector initialised to size (mas-
ter_system_calls_list_length, APK_count)
2. for each app in app_directory
3. APK_package_name = extract_package_name(APK)
4. generate_events(APK_package_name)
5. log = extract_log(APK_package_name)
6. current_system_calls = extract_system_calls(log)
7. for each master_system_calls in master_system_calls
8. for each current_system_calls in cur-
rent_APK_system_calls
9. if(master_system_calls == current_system_calls)
10. system_calls_vector[app_index, master_system_calls_index]=1

11. else
12. system_calls_vector[app_index, master_system_calls_index]=0

5.4 MALWARE DETECTION ALGORITHM

The detection method or the classifier is used to determine whether an app is malware or not. Based on the features, the classifier classifies apps as malware or benign ware. Most classifiers use machine learning. Classifiers based on machine learning use one or multiple classifiers. Layered classifiers may also be used in the detection process. We now build the model using KNN and SVM classification algorithms.

1. Input = preprocessed_vector
2. X = featureExtraction.final_perm_vector
3. Y = featureExtraction.final_binary_class_vector
4. model = KneighborsClassifier(arg)
5. or model = svm.SVC(arg)
6. X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size)
7. model.fit(X, Y)
8. model.score(X, Y)
9. predicted= model.predict(X_test)
10. accuracy = accuracy_score(y_test, predicted)
11. report = classification_report(y_test, predicted)
12. Output = classification_report

CHAPTER 6

RESULTS AND DISCUSSION

6.1 DATASET

The first component of the malware detection system is the data set. The data set represents a representative collection of malware and benign ware. A proper data set is essential for analysis of the behavior of the malware. We need examples of both the malware and the benign apps for proper detection. The most common sources of Android Mobile Malware are the Genome project, Contagio, DREBIN data set, Virus Share, etc. The most widely used sources for known benign apps are Google Play, App China, Amazon, Android Market, etc. Malwares can hide their malicious activities and be available in Android market places. Android apps are available in both official and third-party apps market places. They are also good sources of Android applications which can be used as the data set in the detection engine.

6.2 RESULTS AND SCREENSHOTS

6.2.1 Static feature extraction

The features that can be extracted by not executing the application are called static features. The mechanism that extracts the static features is called a Static Feature Extractor. Researchers may consider only a single category of static features and others consider a set of multiple static features. Common static features are: Permission, API call, String extracted, Native commands, XML elements, Meta data, Intents,

Broadcast receivers, Hardware components, etc.

Static features can be extracted from the manifest file, dex file, and byte code. The most widely used tool is the APK tool. From the APK tool, we obtain the APK file, Manifest file, classes.dex, and Smali file. From these files, we can extract features.

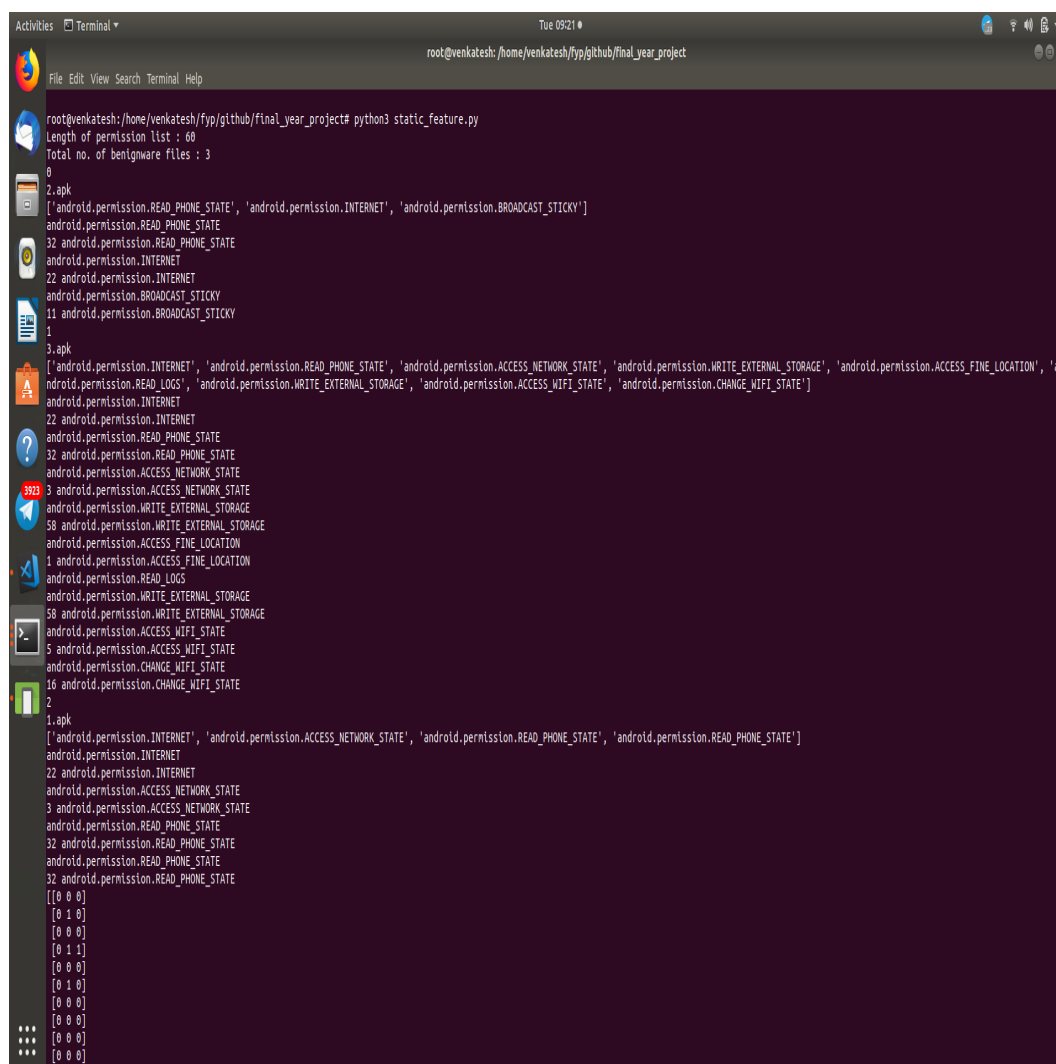


Figure 6.1 Static Feature Extraction

6.2.2 Dynamic feature extraction(Generating Events)

The log file is generated by executing the application in a virtual machine. From the data set, we collect the malware and benign ware and execute these in the virtual machine (Figure 4), which generates the

run time log file. From the log file, we can extract the dynamic features.

```

root@venkatesh:/home/venkatesh/fyp/github/final_year_project/dynamic# python3 log_extraction.py
Package Name : con.truecaller
Already installed
Initialising Command adb shell monkey -p con.truecaller -v 1
Process ID Extraction Command adb shell ps | grep 'con.truecaller'
Returned Process ID u0_a73 21704 1162 1292032 29484 ffffffff b7615d52 R con.truecaller

Underprocessing Processing Process ID :
['u0_a73', '', '', '21704', '1162', '', '1292032', '29484', 'fffffff', 'b7615d52', 'R', 'con.truecaller\n']
Process ID :21704
Random Events Command adb shell monkey -p con.truecaller -v 1500
  
```

Figure 6.2 Dynamic Feature Extraction(Generating events)

6.2.3 Dynamic feature extraction(Extracting Log)

This consists of the process ID for the APK which is used to run on the emulated environment for obtaining the dynamic features.

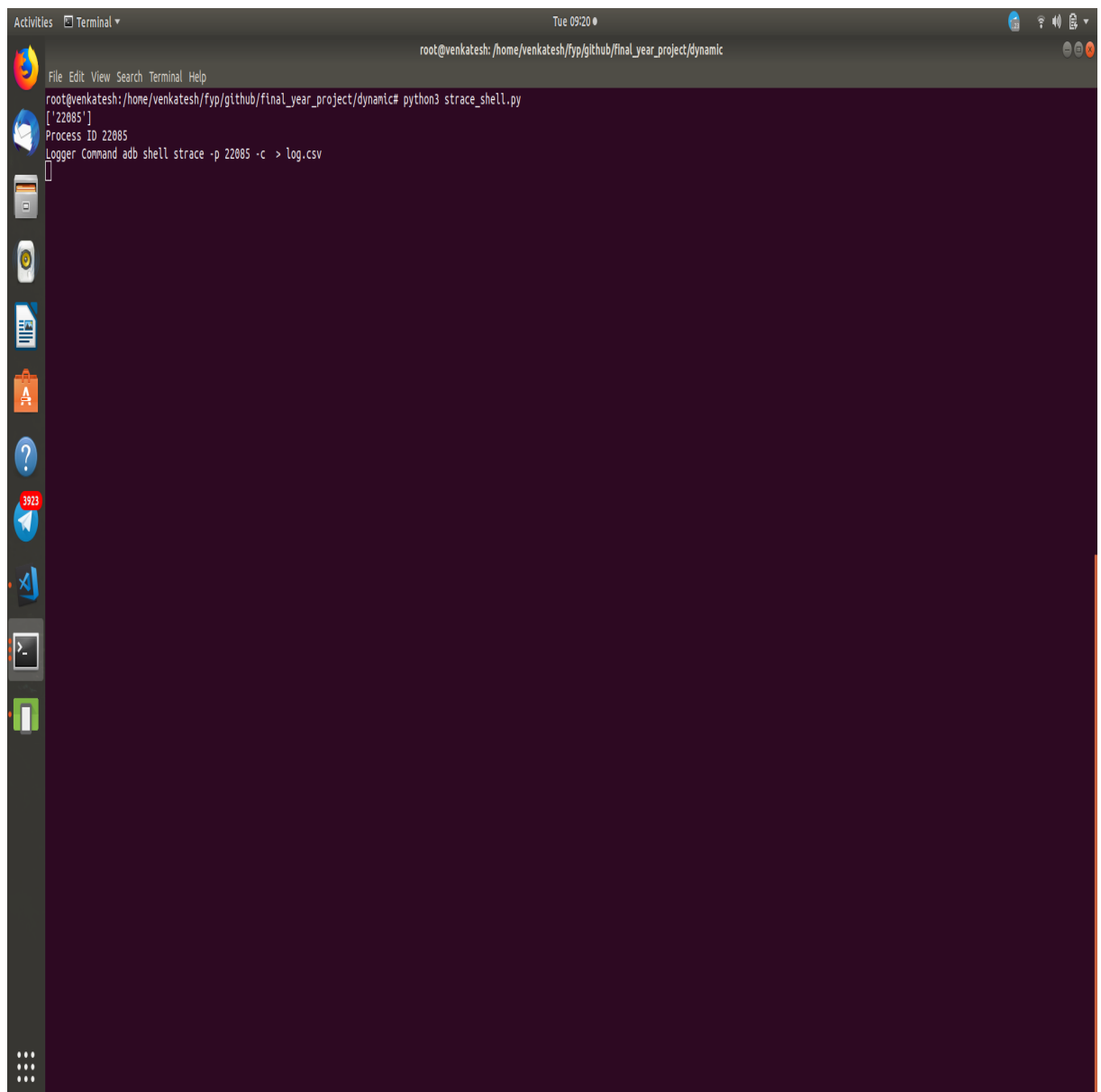


Figure 6.3 Dynamic Feature Extraction(Extraction Log)

6.2.4 Dynamic feature extraction(Log File)

This consists of summary of the application run for a particular set of random events. It includes all the system calls that were executed during this phase.


```

1 Process 22085 attached
2 % time seconds usecs/call calls errors syscall
3 -----
4 60.27 0.106024 68 1557 248 futex
5 8.99 0.015811 74 213 mmap2
6 7.69 0.013526 10 1385 ioctl
7 6.17 0.010849 3 3913 1410 recvfrom
8 6.05 0.010638 54 197 writev
9 5.07 0.008923 8 1059 epoll_pwait
10 2.28 0.004011 2 2204 sendto
11 2.00 0.003514 0 7280 clock_gettime
12 0.76 0.001330 25 54 epoll_ctl
13 0.39 0.000685 2 332 read
14 0.29 0.000565 1 354 pread64
15 0.06 0.000109 0 1003 write
16 0.00 0.000000 0 27 dup
17 0.00 0.000000 0 1 getppid
18 0.00 0.000000 0 100 gettimeofday
19 0.00 0.000000 0 188 munmap
20 0.00 0.000000 0 1 getpriority
21 0.00 0.000000 0 16 clone
22 0.00 0.000000 0 261 mprotect
23 0.00 0.000000 0 1 sched_yield
24 0.00 0.000000 0 9 prctl
25 0.00 0.000000 0 54 rt_sigprocmask
26 0.00 0.000000 0 1 fstat64
27 0.00 0.000000 0 798 getuid32
28 0.00 0.000000 0 2 getdents64
29 0.00 0.000000 0 28fcntl64
30 0.00 0.000000 0 1 statfs64
31 0.00 0.000000 0 11 openat
32 0.00 0.000000 0 20 fstatat64
33 0.00 0.000000 0 6 fsync

```

```

> git for-each-ref --format %(refname) %(objectname) --sort -committerdate
> git remote --verbose
> git show :dynamic/trace_shell.py
> git show :dynamic/log_extraction.py
> git status -z -u
> git symbolic-ref --short HEAD
> git rev-parse master
> git rev-parse --symbolic-full-name master@{u}
> git rev-list --left-right master...refs/remotes/origin/master
> git for-each-ref --format %(refname) %(objectname) --sort -committerdate
> git remote --verbose
> git show :dynamic/vector_generation.py
> git show :dynamic/log_extraction.py
> git show :dynamic/vector_generation.py
> git show :dynamic/extracted_syscall.csv
> git show :dynamic/log.csv

```

Figure 6.4 Dynamic Feature Extraction(Log File)

6.2.5 Dynamic feature extraction(Extracted System Calls)

This consists of the system calls alone extracted from the summary of the output obtained when the APK is run in the emulated environment.

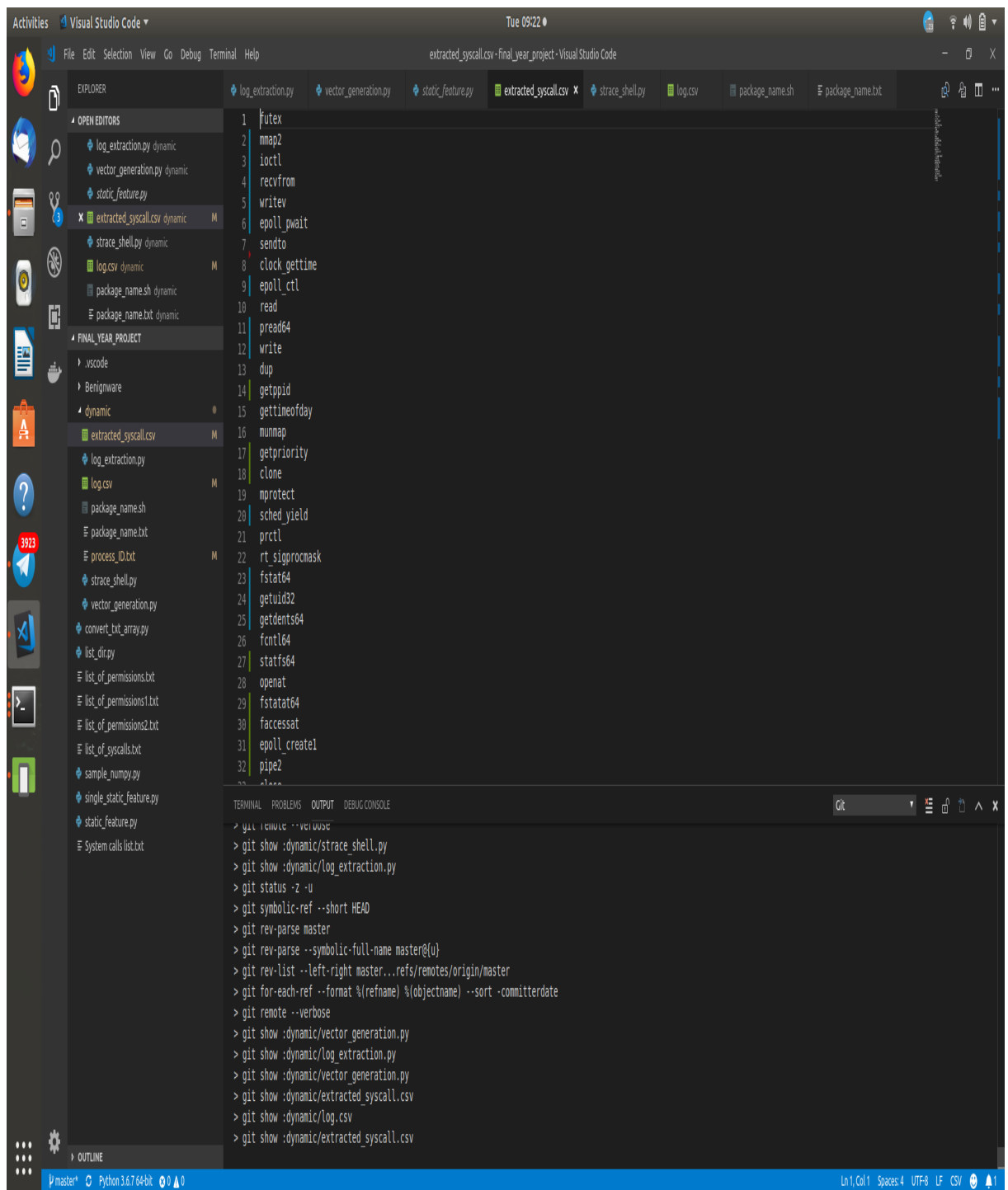


Figure 6.5 Dynamic Feature Extraction(Extracted System Calls)

6.2.6 Dynamic feature extraction(Generating Vector)

The extracted system calls are compared with an all system call list and vector is generated based on that.

```

root@venkatesh: /home/venkatesh/fyp/github/final_year_project/dynamic

File Edit View Search Terminal Help
Package name : con.truecaller
Already installed
Initialising Command adb shell monkey -p con.truecaller -v 1
Process ID Extraction Command adb shell ps | grep 'con.truecaller'
Returned Process ID u0_a73 21704 1162 1292832 29484 ffffffff b7615d52 R con.truecaller

Underprocessing Processing Process ID :
['u0_a73', '', '', '21704', '1162', '', '1292832', '29484', 'ffffffff', 'b7615d52', 'R', 'con.truecaller\n']
Process ID : 21704

Random Events Command adb shell monkey -p con.truecaller -v 1500
log_extraction.py:94: FutureWarning: The signature of 'Series.to_csv' was aligned to that of 'DataFrame.to_csv', and argument 'header' will change its default value from False to True: please pass an explicit value to suppress this warning.
log_csv.to_csv('extracted_syscall.csv', sep='\t', index=False)
List : ['futex', 'epoll_ctl', 'sendto', 'write', 'clock_gettime', 'getuid32', 'recvfrom', 'pread64', 'read', 'ioctl', 'epoll_wait', 'dup', 'gettimeofday', 'mmap', 'mprotect', 'writev', 'prctl', 'rt_sigprocmask', 'mmap2', 'fcntl64', 'openat', 'close']
Length of extracted syscall list : 22
Total List : ['_exit', '_exit_thread', '__fork', '__waitpid', '__waitid', '__sys_clone', 'execve', '__setuid:setuid32', '__setuid:setuid', 'getuid:getuid', 'getgid:getgid32', 'getgid:getgid', 'getuid:getuid32', 'getuid:getuid', 'getgid:getgid32', 'getgid:getgid', 'getuid():', 'readahead', 'getgroups:getgroups32', 'getgroups:getgroups', 'getpgid', 'getppid', 'getsid', 'setsid', 'setgid:setgid32', 'setgid:setgid', 'setuid:setuid32', '__setuid:setuid32', '__setuid:setuid', '__setresuid:setresuid32', '__setresuid:setresuid', 'setresgid:setresgid32', 'setresgid:setresgid', '__brk:brk', 'kill', 'tkill', 'tgkill', '__ptrace:ptrace', '__set_thread_area:set_thread_area', 'getpriority', 'setpriority', 'setrlimit', 'ugetrlimit', 'getrlimit', 'setrusage', 'setgroups:setgroups32', 'setgroups:setgroups', 'getpgrp', 'setpgid', 'vfork', 'setregid:setregid32', 'setregid:setregid', 'chroot', 'prctl', 'capget', 'capset', 'sigaltstack', 'acct', 'pwrite64', '__open:open', '__openat:openat', 'close', 'creat', 'off_t', '__llseek:llseek', 'getpid', 'mmap', '__mmap2:mmap2', 'munmap', 'mremap', 'nsync', 'madvise', 'mlock', 'munlock', 'mlockall', 'munlockall', 'mincore', 'ioctl', 'readv', 'writev', '__fcntl:fcntl', 'flock', 'fchmod', 'dup', 'pipe', 'pipe2', 'dup2', 'select:newsselect', 'ftruncate', 'ftruncate64', 'getdents:getdents64', 'fsync', 'fdatasync', 'fchown:fchown32', 'fchown:fchown', 'sync', 'fcntl64', 'fstatfs64', 'sendfile', 'fstatat:fstatat64', 'mkdirat', 'fchownat', 'fchmodat', 'renameat', 'fsetxattr', 'fgetxattr', 'flistxattr', 'fremovexattr', 'link', 'unlink', 'unlinkat', 'chdir', 'mknod', 'chmod', 'chown:chown32', 'chown:chown', 'lchown:lchown32', 'lchown:lchown', 'mount', 'umount', 'umount2', 'fstat:fstat64', 'stat:stat64', 'lstat:lstat64', 'mknod', 'readlink', 'rmdir', 'rename', '__getcwd:getcwd', 'access', 'faccessat', 'symlink', 'fchdir', 'truncate', 'setxattr', 'lsetxattr', 'getxattr', 'lgetxattr', 'listxattr', 'llistxattr', 'removexattr', 'lremovexattr', '__statfs64:statfs64', 'long_unshare', 'pause', 'gettimeofday', 'settimeofday', 'clock_gettime', 'clock_gettime', 'clock_getres', 'clock_nanosleep', 'getitimer', 'setitimer', '__timer_create:timer_create', '__timer_settime:timer_settime', '__timer_gettime:timer_gettime', '__timer_getoverrun', '__timer_delete:timer_delete', 'utimes', 'utimensat', 'sigaction', 'sigprocmask', '__sigsuspend:sigsuspend', '__sigsuspend:sigsuspend', '__rt_sigaction:rt_sigaction', '__rt_sigprocmask:rt_sigprocmask', '__rt_sigtimedwait:rt_sigtimedwait', 'sigpending', 'socket', 'socketpair', 'bind', 'connect', 'listen', 'accept', 'getsockname', 'getpeername', 'sendto', 'recvfrom', 'shutdown', 'setsockopt', 'getsockopt', 'sendmsg', 'recvmsg', 'socket:socketcall:1', 'bind:socketcall:2', 'connect:socketcall:3', 'listen:socketcall:4', 'accept:socketcall:5', 'getsockname:socketcall:6', 'getpeername:socketcall:7', 'socketpair:socketcall:8', 'sendto:socketcall:11', 'recvfrom:socketcall:12', 'shutdown:socketcall:13', 'setsockopt:socketcall:14', 'getsockopt:socketcall:15', 'sendmsg:socketcall:16', 'recvmsg:socketcall:17', 'sched_setscheduler', 'sched_getscheduler', 'sched_yield', 'sched_setparam', 'sched_getparam', 'sched_get_priority_max', 'sched_get_priority_min', 'sched_rr_get_interval', 'sched_setaffinity', 'sched_getaffinity:sched_getaffinity', '__getcpu:getcpu', 'ioprio_set', 'ioprio_get', 'uname', '__wait4:wait4', 'mode_t_unmask', 'reboot:reboot', '__syslog:syslog', 'init_module', 'delete_module', 'klogctl:syslog', 'sysinfo', 'personality', 'long_perf_event_open', 'futex', 'epoll_create', 'epoll_ctl', 'epoll_wait', 'inotify_init', 'inotify_add_watch', 'inotify_rm_watch', 'poll', 'eventfd:eventfd2', '__set_tls:ARM_set_tls', 'cacheflush:ARM_cacheflush', '__flush_cache:cacheflush', 'syscall', 'read', 'write', 'pread64']
Length of total syscall list : 244
clock_gettime
54 clock_gettime
close
58 close
dup
63 dup
epoll_ctl
66 epoll_ctl
epoll_wait
fcntl64
77 fcntl64
futex
90 futex
gettimeofday
120 gettimeofday
getuid32
127 ioctl
mmap2

```

Figure 6.6 Dynamic Feature Extraction(Generating Vector)

6.2.7 Dynamic feature extraction(Generated Vector)

The generated vector is shown as output

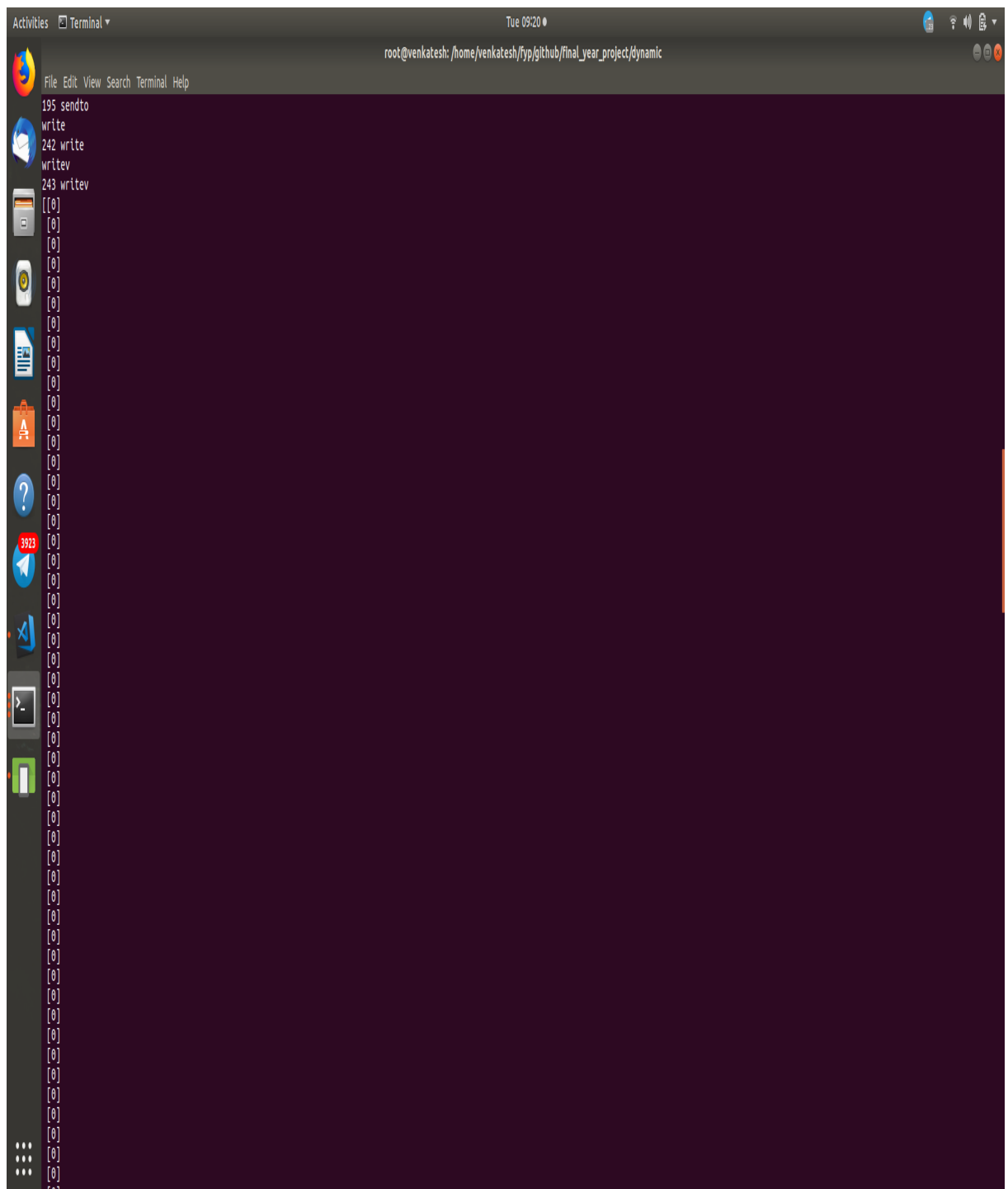
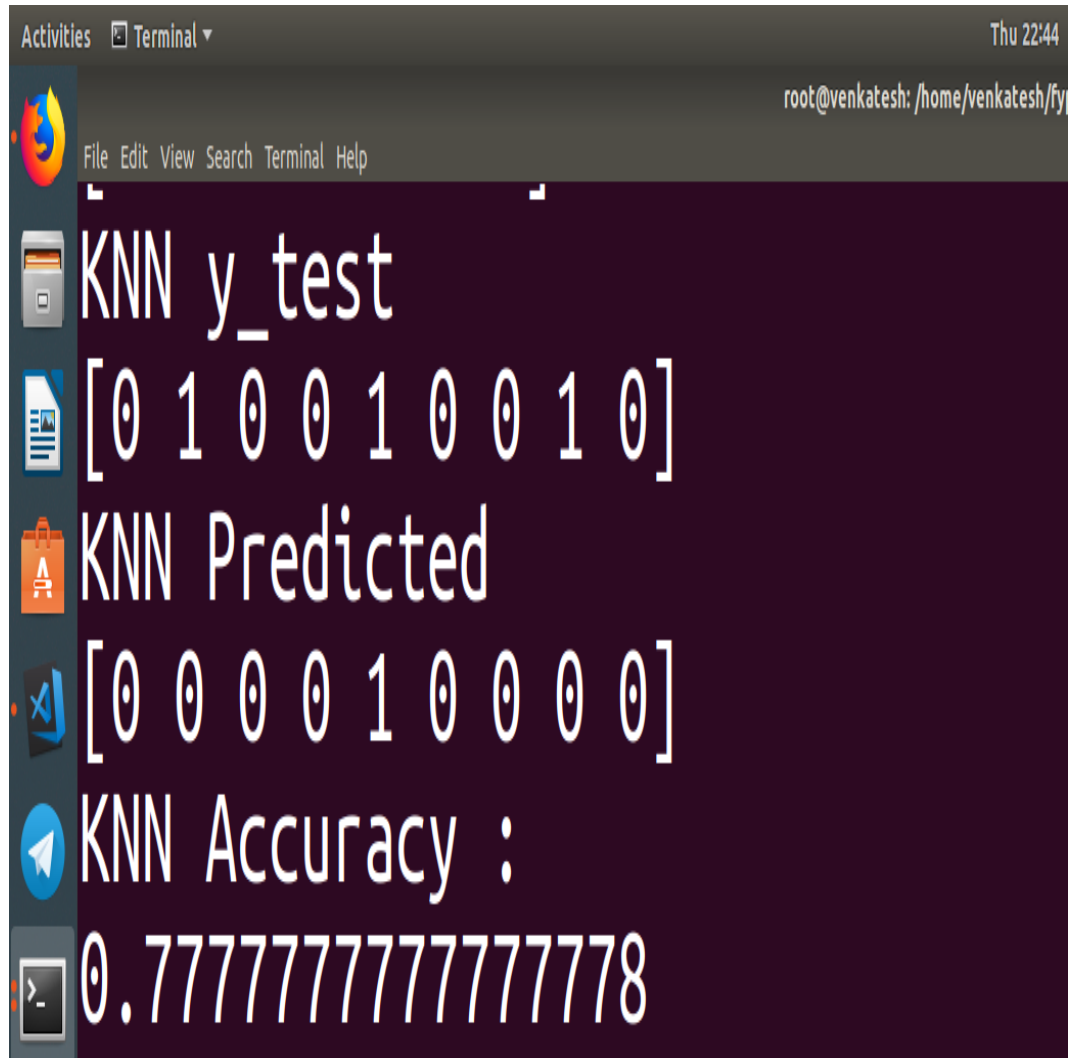


Figure 6.7 Dynamic Feature Extraction(Generating Vector)

6.2.8 KNN

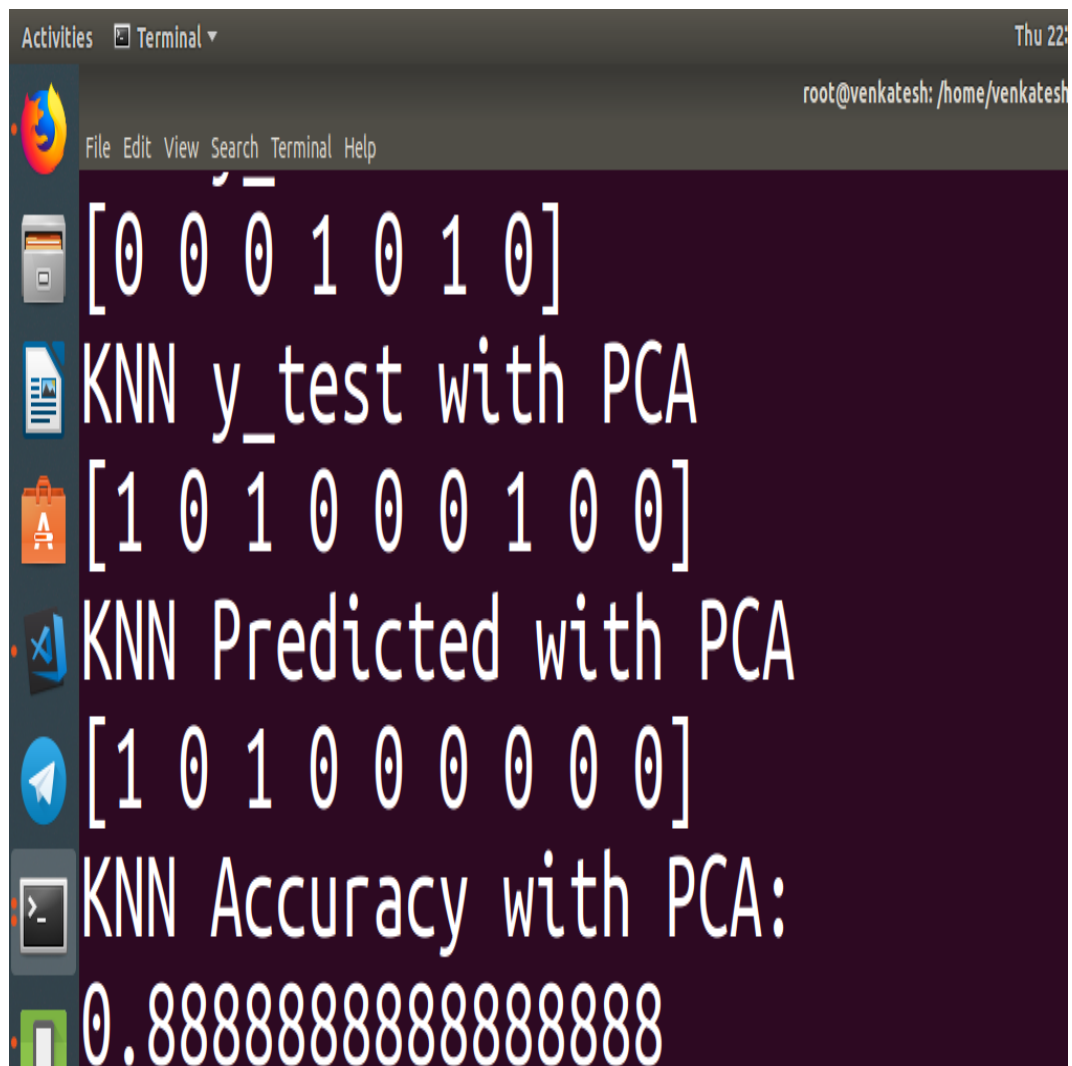
The selected features are then sent as input to the KNN algorithm. Training and testing is done based on both vectors X and Y. It is seen that

the accuracy is better when KNN is performed on the features selected using PCA.

A screenshot of a Linux terminal window. The title bar shows 'Activities' and 'Terminal'. The top right corner displays the time 'Thu 22:44' and the user path 'root@venkatesh: /home/venkatesh/fy'. The terminal has a dark purple background with white text. On the left side, there is a vertical dock with icons for Firefox, a file manager, a document, a shopping bag, a code editor, a Telegram icon, and a terminal icon. The terminal output is as follows:

```
KNN y_test  
[0 1 0 0 1 0 0 1 0]  
KNN Predicted  
[0 0 0 0 1 0 0 0 0]  
KNN Accuracy :  
0.7777777777777778
```

Figure 6.8 KNN Accuracy

A terminal window with a dark background and light-colored text. The window title is 'Activities Terminal'. The user is 'root@venkatesh' in the directory '/home/venkatesh'. The terminal shows a 7x8 matrix of 0s and 1s, followed by the text 'KNN y_test with PCA', another 7x8 matrix of 0s and 1s, the text 'KNN Predicted with PCA', a third 7x8 matrix of 0s and 1s, the text 'KNN Accuracy with PCA:', and finally the accuracy value '0.8888888888888888'.

```
Activities Terminal Thu 22:10  
root@venkatesh: /home/venkatesh  
File Edit View Search Terminal Help  
[0 0 0 1 0 1 0]  
KNN y_test with PCA  
[1 0 1 0 0 0 1 0 0]  
KNN Predicted with PCA  
[1 0 1 0 0 0 0 0 0]  
KNN Accuracy with PCA:  
0.8888888888888888
```

Figure 6.9 KNN Accuracy(With PCA)

6.2.9 SVM

The selected features are then sent as input to the SVM algorithm. Training and testing is done based on both vectors X and Y. The SVM train test and SVM Accuracy screenshots have been shown below

```
SVM X_train
[[ 1.26562683  0.22285325]
 [-0.41383224  0.38516509]
 [-0.28501273  0.1680418 ]
 [ 0.25037518 -0.57015804]
 [ 0.25037518 -0.57015804]
 [-0.85603262  0.1168416 ]
 [-0.33754971  0.23399505]]

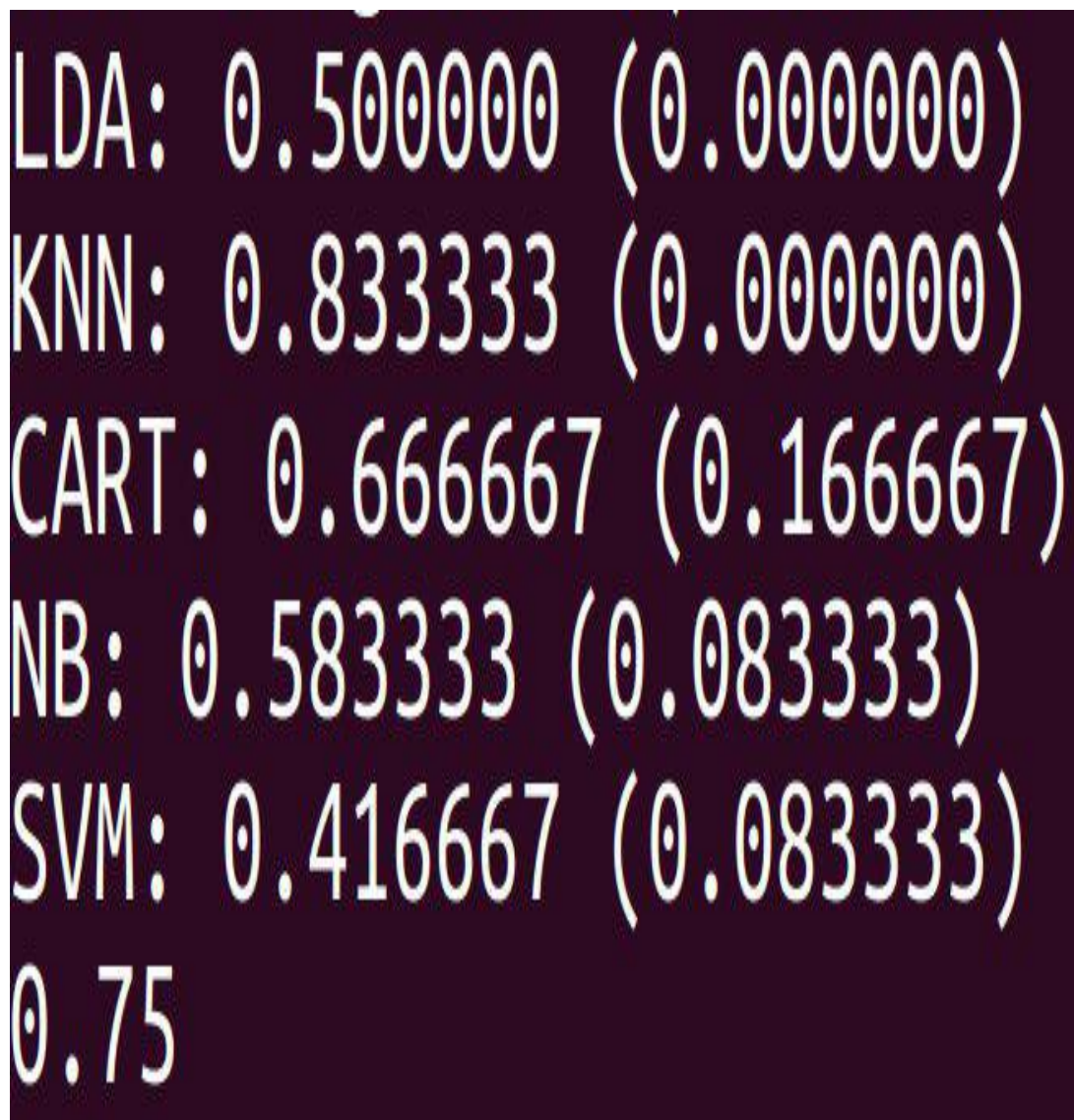
SVM X_test
[[ 0.73023892  0.9610531 ]
 [-0.09289056 -0.78412837]
 [-0.62827848 -0.04592853]
 [-0.62827848 -0.04592853]
 [ 0.25037518 -0.57015804]
 [-0.60728623  0.46064938]
 [ 0.25037518 -0.57015804]]
```

Figure 6.10 SVM train test

```
[ -0.60728623  0.46064938]
[  0.25037518 -0.57015804]
[ -0.41383224  0.38516509]
[  1.26562683  0.22285325]]
SVM y_train
[1 0 0 0 0 1 0]
SVM y_test
[0 0 1 1 0 1 0 0 0]
SVM Predicted
[0 0 0 0 0 0 0 0 0]
SVM Accuracy :
0.6666666666666666
```

Figure 6.11 SVM Accuracy

6.2.10 Final Accuracy



```
LDA: 0.500000 (0.000000)
KNN: 0.833333 (0.000000)
CART: 0.666667 (0.166667)
NB: 0.583333 (0.083333)
SVM: 0.416667 (0.083333)
0.75
```

Figure 6.12 Final Accuracy

6.3 EVALUATION PARAMETERS

The True Positive Rate (TPR) defines the percentage of benign apps identified accurately, where

$$TPR = TP / (TP + FN) \quad (6.1)$$

TP is the number of accurately identified benign apps and FN is the number of incorrectly identified benign apps.

The False Positive Rate (FPR) defines the percentage of incorrectly identified malware apps, where

$$FPR = FP / (TN + FP). \quad (6.2)$$

FP is the number of incorrectly identified malware and TN is the number of correctly identified malware.

Accuracy is a metric used to describe the overall performance. Accuracy is the percentage of correctly identified apps, where

$$ACC = (TP + TN) / (TP + TN + FP + FN). \quad (6.3)$$

Application	Accuracy	Precision	Recall	F1 Score
Benignware	81.94	90	90	90
Malware	81.94	88	88	88

Table 6.1 Evaluation Metrics

Table 6.1 represents the values of the evaluation metrics we have used to evaluate the performance of our model

Application	Expected	Predicted
1	Benignware	Benignware
2	Malware	Malware
3	Benignware	Benignware
4	Benignware	Benignware
5	Benignware	Benignware
6	Malware	Benignware
7	Benignware	Benignware
8	Malware	Malware
9	Malware	Malware
10	Benignware	Benignware
11	Benignware	Benignware
12	Malware	Malware
13	Malware	Malware
14	Malware	Malware
15	Malware	Malware
16	Benignware	Malware
17	Benignware	Benignware
18	Benignware	Benignware

Table 6.2 Testcase results

6.4 TESTCASES

The system takes set of applications as input and gives predicted output.

```

Activities Terminal Tue 07:48
root@venkatesh:/home/venkatesh/fyp/github/final_year_project

warnings.warn("Variables are collinear.")
KNN: 0.819444 (0.041667)
CART: 0.805556 (0.027778)
NB: 0.805556 (0.055556)
SVM: 0.708333 (0.013889)

0 = benignware
1 = malware
Predicted :[0. 1. 0. 0. 0. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0.]
Expected  :[0. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1. 0. 0.]
Accuracy
0.8888888888888888
[[9 1]
 [1 7]]

           precision    recall  f1-score   support

     0.0         0.90      0.90      0.90         10
     1.0         0.88      0.88      0.88          8

   micro avg       0.89      0.89      0.89         18
   macro avg       0.89      0.89      0.89         18
  weighted avg       0.89      0.89      0.89         18

root@venkatesh:/home/venkatesh/fyp/github/final_year_project#

```

Figure 6.13 Testcases

6.4.1 Comparison between algorithms

Accuracy of detecting whether an application is a benignware or malware is plotted in a graph against various machine learning models.

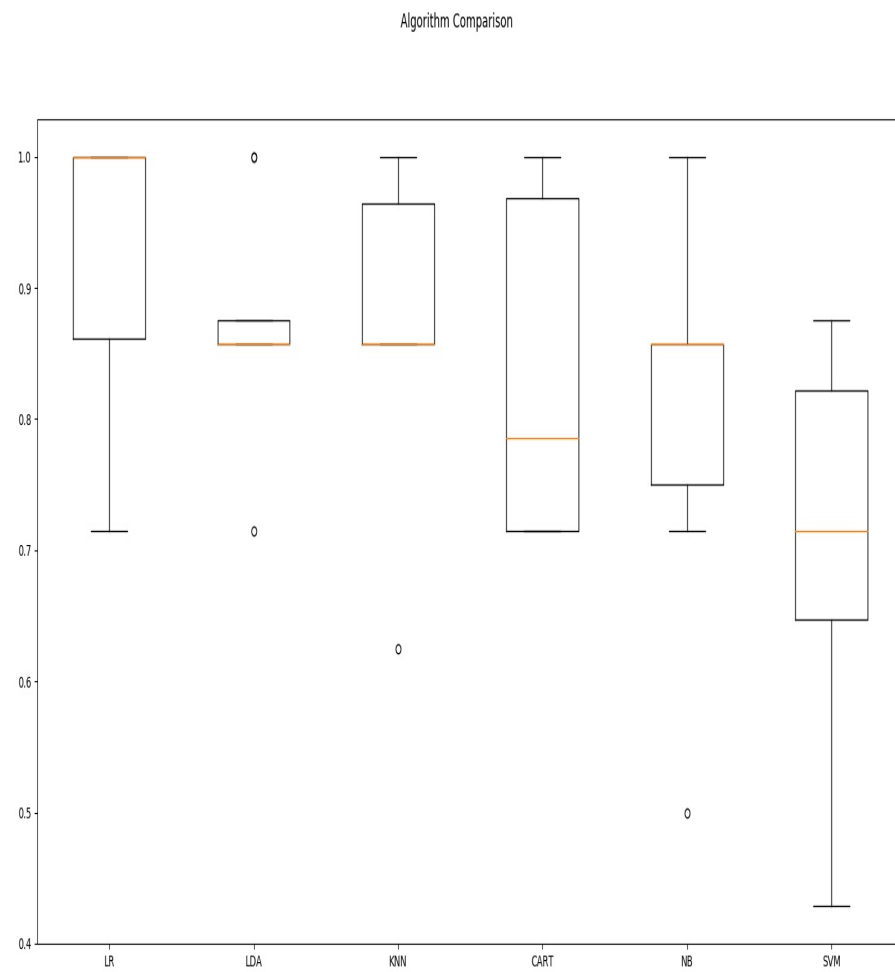


Figure 6.14 Accuracy of different algorithms

CHAPTER 7

CONCLUSION

7.1 CONTRIBUTIONS

Researchers have used static, dynamic, and hybrid processes to detect mobile malware and malicious activities. Researchers primary concerns involve accuracy levels, and most the research papers describe the performance of their detection process using accuracy metrics. For the operating system of mobile devices, performance overhead should be considered, as higher accuracy may cause higher overhead. Accuracy and performance overhead need to be well balanced to make the detection process efficient.

The static feature is formed by analyzing the structure and format of the sample and then extracting the hash value, string information, function information, header file information, and resource description information. The technology obtains most of the malware information from the malware itself, thus the analysis results are relatively comprehensive. However, static features cannot correctly discriminate malware when the static information is packed or obfuscated or compressed [7], making it difficult for static features to express the true purpose of malware, thus affecting the accuracy of detection.

Dynamic features are the behavior of the sample execution and the features of the debug record, such as file operations, the creation and deletion of processes, and other dynamic behaviors. Since the malicious behaviors of malware at dynamic runtime cant be concealed, the

extracted dynamic features provide a more realistic description than the static features. However, the extraction of dynamic features needs to be run in a virtual environment [8], which will be reset and restored to the previous state after each malicious sample has been analyzed to ensure that the virtual environment is a real user environment. As a result, features extraction efficiency is much lower than for static features.

We analyze the ongoing research efforts covering the three basic categories: static, dynamic, and hybrid analysis. These analyses represent the data set, features, feature selection method, detection method, and the accuracy. We also have mentioned the literature gap and the limitations of current research efforts. Thereby we have identified the suspicious feature lists which are commonly used by malware developers. The main contributions of this research effort are divided into the following main areas:

- The individual entries are indicated with a black dot, a so-called bullet.
- The text in the entries may be of any length.
- Defining the mobile malware detection process for IoT networks.
- Determining the security limitations for mobile platforms in industrial IoT networks.
- A comparative analysis of static, dynamic, and hybrid detection processes and their limitations and scopes.
- Identifying the suspicious permission, API call, and system call lists to enable IoT application developers in the safe use of APIs.

We propose to combine permission and API (Application Program Interface) calls and use machine learning methods to detect malicious Android Apps. In our design, the permission is extracted from each App's profile information and the APIs are extracted from the packed App file by using packages and classes to represent API calls. By using

permissions and API calls as features to characterize each Apps, we can learn a classifier to identify whether an App is potentially malicious or not.

7.2 SUMMARY

We have analyzed the static, dynamic, and hybrid analysis methods for mobile malware detection. The analysis includes recent literature on Zero-day detection. The detection process, feature extraction and selection process, and detection algorithms are discussed in this study. We have found that machine learning approaches are commonly used to classify malware and benign ware. The suspicious permission list, API call list, and the system call list are also identified to assist application developers. In future, we plan to design and implement a framework that will be able to detect mobile malware with a high accuracy to ensure Zero-day detection.

7.3 FUTURE WORK

There are several future research questions. First, we would like to study the loss landscape and determine how to improve the deep learning models accuracy for classification. Next, we would like to implement application programming interfaces for firewall protection, Distributed Denial of Services handling, load description and load handling etc. We can also attempt to handle adversarial malware attacks. We plan to conduct further research on feature selection for multivariate anomaly detection, and investigate principled methods for choosing the latent dimension and PC dimension with theoretical guarantees. We also hope to perform a detailed study on the stability of the detection model. In terms of applications, we can explore the use of SVM/KNN for other anomaly detection applications such as predictive maintenance and fault diagno-

sis for smart buildings and machineries.

REFERENCES

- [1] Win. Aung, Zarni Zaw, “Permission-based android malware detection”, volume 9, pp. 228–234. *International Journal of Scientific and Technology Research*, 2013.
- [2] I. Bulut and A. G. Yavuz, *Mobile malware detection using deep neural network*, volume 26, *IEEE Access*, pp. 205-215, 2017.
- [3] Q. Yan Z. Li W. Srisa-an J. Li, L. Sun and H. Ye, *Significant Permission Identification for Machine-Learning-Based Android Malware Detection*, volume 8, *IEEE Transactions on Industrial Informatics*, 2018.
- [4] X. Su K. Zhao, D. Zhang and W. Li, “Fest: A feature extraction and selection tool for android malware detection”, *IEEE Symposium on Computers and Communication (ISCC)*, vol. 89, num. 4, pp. 714–720, 2015.
- [5] N. Peiravian and X. Zhu, “Machine learning for android malware detection using permission and api calls”, volume 26, pp. 300–305. *IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013.
- [6] J. H. Abawajy W. N. Ismail S. Sharmeen, S. Huda and M. M. Hassan, *Malware Threats and Detection for Industrial Mobile-IoT Networks*, volume 51, *IEEE Access*, vol. 6, pp. 15941-15957, 1983.
- [7] Y. Huo R. Zhang X. Li, J. Liu and Y. Yao, “An android malware detection method based on androidmanifest file”, *CCIS*, vol. 22, pp. 239–243, 2016.