Android Malware Classification by Machine Learning Apprehension and Static Feature Characterization

By

Md. Rashedul Hasan
(161-35-1556)

A thesis submitted in partial fulfillment of the requirement for the degree of

Bachelor of Science in Software Engineering

Department of Software Engineering
DAFFODIL INTERNATIONAL UNIVERSITY

Fall – 2019

APPROVAL

This **Thesis** titled "**Android Malware Classification by Machine Learning Apprehension and Static Feature Characterization**", submitted by **Md. Rashedul Hasan**, **161-35-1556** to the Department of Software Engineering, Daffodil International University has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of B.Sc in Software Engineering and approved as to its style and contents.

BOARD OF EXAMINERS

Dr. Touhid Bhuiyan
Professor and Head — **Chairman**
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University

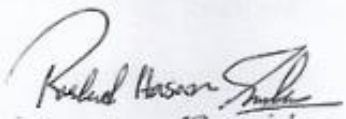
Dr. Md. Asraf Ali
Associate Professor — **Internal Examiner 1**
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University

Asif Khan Shakir
Lecturer — **Internal Examiner 2**
Department of Software Engineering
Faculty of Science and Information Technology
Daffodil International University

Prof Dr. Mohammad Abul Kashem
Professor — **External Examiner**
Department of Computer Science and Engineering
Faculty of Electrical and Electronic Engineering
Dhaka University of Engineering & Technology, Gazipur

DECLARATION

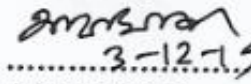
I hereby declare that I have taken this thesis under the supervision of **Afsana Begum, Senior Lecturer, Department of Software Engineering, Daffodil International University.** I also declare that neither this thesis nor any part of this has been submitted elsewhere for award of any degree.



.....03.12.2019.....

Md. Rashedul Hasan
ID: 161-35-1556
Batch : 19
Department of Software Engineering
Faculty of Science & Information Technology
Daffodil International University

Certified by:



.....3-12-19.....

Afsana Begum
Senior Lecturer
Department of Software Engineering
Faculty of Science & Information Technology
Daffodil International University

ACKNOWLEDGEMENT

First I express my heartiest thanks and gratefulness to almighty God for His divine blessing makes me possible to complete the final year thesis successfully.

I am really grateful and wish my indebtedness to **Afsana Begum** , **Senior Lecturer**, Department of Software Engineering, Daffodil International University, Dhaka. Deep Knowledge & keen interest of my supervisor in the field of "*Malware Analysis*" to carry out this thesis. Her endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior draft and correcting them at all stage have made it possible to complete this project.

I would like to express my heartiest gratitude to Dean, Associate Dean, and Head**,** Department of Software Engineering, for their kind help to finish this thesis and also to other faculty member and the staff of Software Engineering department of Daffodil International University.

Finally, I must acknowledge with due respect the constant support and patients of my parents. Specially I would like to remember my loving father as he departed for afterlife on October 13, 2019 - who meant the world to me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \quad (3.1)$$

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.2)$$

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.3)$$

$$\text{F1 Score} = \frac{(1 + \beta^2) \times \text{Recall} \times \text{Precission}}{\beta^2 \times Recall \times Precission} \quad (3.4)$$

LIST OF NOMENCLATURES

β	A comparative value for precision

LIST OF ABBREVIATIONS

ML	Machine Learning
VT	Virus Total
VT Graph	Virus Total Graph
BoW	Bag of Words

ABSTRACT

The increased usage and popularity of Android devices enables developers of malware to produce new ways to develop malware in various packaged forms in different applications. These malware causes fundamental information leakage and financial harm. Unethical programmers and exploit writers repackages malicious code and launches again in the market in the form of a new application. The repackaged software is regrettably most often remains undetected. In this research, emphasis was given to the problem of repackaging using the Bag-of-Word algorithm for implementing the source code and evaluating the results using the machine learning. The results of the evaluation resembles 0.55 percent better than the existing source code-based implantation in this field with modifications in the Bag-of-Word technique and additional preprocessing of dataset. In this research a vocabulary was generated to identify malicious source code structure. More 12 malicious patterns were added to the existing 69 mischievous patterns. The concept was practically incorporated via a web application. The proposed methodology offers a comparatively newer approach to analyze malware source code to address malware repackaging.

Keywords: Malware Analysis, Android Malware, Source code, text processing, repackaging, Bag-of-words.

CHAPTER 1
INTRODUCTION

The malware business is a lucrative one as malware writers explore different surfaces for intrusion and developing new exploits. , for the reason it has become a remunerative business. this can be one in all several reasons that produces finding out malware therefore fascinating. It's a remarkable mixture of technology, psychology, and commerce. psychological science is what makes malware effective, and commerce is what ensures a lot of hackers still develop new and attention-grabbing malware. Malware analysis is important to know the variance and the technology of malware development to initiate necessary countermeasure. Researchers have initiated different types of approaches in Malware analysis techniques such as harnessing ontologies and machine-learning capabilities for malware analysis (Navarro, L. C., Navarro, A. K., Grégio, A., Rocha, A., & Dahab, R., 2018) or visual analytics approaches of malware analysis by the means of reverse engineering (Cappers, B. C., Meessen, P. N., Etalle, S., & Wijk, J. J. V. , 2018). Even after many evident approaches malware developers are initiating new ways to launch malwares which generates the urge to conduct more influential research in this area.

1.1 Background

As our world is being connected at a swift pace number of cellular devices are on the rise. By an approximation by the year 2020 (Boxall A., 2015). The abundance of private information preserved on or accessible through these devices has made them an attractive target for cyber criminals (Dehghantanha, A., & Franke, K., 2014). From different types of studies researches revealed users are not accustomed to install anti-virus or anti anti-

malwares on their mobile devices as the efficiency of such applications are debatable. (Walls, J., & Choo, K.-K. R. ,2015) Malware developers or Exploit writers concentrates their focus on platforms or operating systems possessing rather a larger market share. (Kitagawa M , Gupta A, Cozza R, Durand I, Glenn D, Maita K, 2015) From the business perspective mobile devices are to be regarded as “feeble links” of consortium security. Who doesn’t realize Androids permission based security infrastructure offers defense in a lesser extent when it comes to application permissions. (Chia C , Choo K-KR Fehrenbacher D.,2017) In multiple instances malicious applications were approved by Google Play store , which made Android a prudent platform for malware developers. (Viennot N,Garcia E , Nieh J, 2014) Again repackaging of Malware source code intensifies the probability and the likelihood for a Malware to exist in different variants. The urged reasons creates a plea for more efficient Malware analysis capabilities and methods for to be taken into consideration.

1.2 Motivation

If existing analysis approaches are comprehensively ordered , they can be classified two major categories. Dynamic malware analysis and Static malware analysis. Of course there is also Hybrid malware analysis which takes consideration of both the methods. In static analysis , one audits and assesses the source code and pairs so as to discover suspicious examples. Dynamic analysis includes the execution of the application program while tracking and observing its expected and unexpected behavior. Malware detection engines respectively use the technology to identify threats according to different criteria’s. conclusive studies represents that malware detection engines can only detect a malicious behavior when an application is labeled and in a packaged form, they cannot be detected

when they are in raw form. For example: if a malicious apk file is detected by an antivirus engine the developer could decompile his .apk file to get the core source file and from those source files he may collect some portion or functionalities and can implement them in a specialized form in another .apk file to evade detection. As the form of that particular source code has been restated in another file in another form it will not be detected as malicious even if it was before. The reason is malicious functionality like collecting sms, or spying on call records has been implemented in the new .apk file but as a specialized form. Which makes a perception of insecurity and a probability of that malware being repackaged in another form or as an another application. It is to be anticipated such phenomena is an inception which is a major setback for an efficient detection. So, the research concentration has been focused in this particular area to find out a constructive solution.

1.3 Problem Statement

As malware analysis techniques are becoming familiar - Static, Dynamic and Hybrid analysis techniques are being utilized side by side. It is true in packaged form Android malwares are being detected by antivirus engines. However even with combined approach malwares remain undetected in raw form, which enables malware developers to repackage them.

1.4 Research Question

From an inclusive study and detailed assessment this research would try to find a solution about

a. How to address repackaging of malware and perform efficient detection?

1.5 Research Objective

The objective which had been focused on to achieve is:

a. To propose a new approach of static analysis for better detection and accuracy referencing a similar work conducted in the field. This approach will propose an efficient methodology combining malware analysis techniques to address repackaging , as it will also address implementation of text processing (Bag of words algorithm) with static analysis and performing machine learning to understand the effectiveness and accuracy of the proposed method.

To conduct the task raw coding in Python was used, virus total, JD-gui, Dex to jar tools and for web application PHP, HTML, CSS, Mysql Database were used.

1.6 Research Scope

Following the course of action it is expected that repackaging can be addressed properly as android malwares are being repackaged with raw source codes which are being utilized to create new variants. Obtaining better performance utilizing Machine Learning from the proposed methodology which features static source code based malware analysis with text processing. This work will address repackaging which is to be applicable for Android devices at large.

1.7 Organization of chapters

Chapter two discusses about literature review and related works in the field. Chapter three discusses the research methodology. Chapter four discusses analysis and result. Chapter five summarizes the whole concept , future work and concluding remarks.

CHAPTER 2
LITERATURE REVIEW

Android malwares are presenting a serious challenge due to their rapid growth to researchers. The researchers constantly suggest countermeasures and build instruments to combat these attacks (Sharma, Mohit & Chawla, Meenu & Gajrani, Jyoti., 2016).

2.1 Related Works

The detection of deviations on battery usage was the foundation for early approaches to detecting mobile malware (Buennemeyer TK , Nelson TM , Clagett LM , Dunning JP , Marchany RC , Tront JG., 2008). Server operating events, such as API calls, Input / Output demands, and resource locks, have also been used in flexible malware detection approaches. For example, TaintDroid is an uncontrollable variables-based malware detection system for the data use behavior of the app (Enck W , Gilbert P , Han S , Tendulkar V , Chun B-G , Cox LP , et al., 2014). The researchers developed an anomaly monitoring system for the identification of malicious apps for Android Dalvik op-coding frequencies. Machine learning was used to classify malware according to their behavior. The researchers relied for example on runtime activity and classified Android malware in malware families using inter process communications with SVM (Dash SK , Suarez-Tangil G , Khan S , Tam K , Ahmadi M , Kinder J , et al., 2016). In Android malware detection, a random forest strategy is used with a set of 42 vectors, including battery, CPU and memory usage, and network interaction (Alam MS , Vuong ST., 2013).

System calls and regular expressions have been used by authors to detect data leakage, deploy and use destructive apps (Isohara T , Takemori K , Kubota A ., 2011). To order to avoid mobile device deterioration, mechanisms for both static and dynamic malware analysis focused on distributed computing and collaborative analysis are also advocated. For example, M0Droid is a malware-resistant Android solution which explores and produces signatures for system calls of Android apps on the network, which are pointed to threat detection devices (Damshenas M , Dehghantanha A , Choo K-KR , Mahmud R., 2015). static malware identification are believed to reduce investigation speed and interoperability because the techniques are taken consideration as manual practice. Numerous methods have also been introduced for streamlining the static analysis process. Researchers recommended the need for comprehensive methods to test the software's activities to convert malware-source code into CCS statements (Mercaldo F , Nardone V , Santone A , Visaggio CA., 2016). The authors implied a fingerprinting method for applications that capture the binary and structural functionalities of the app (Karbab EB , Debbabi M , Mouheb D., 2016). Techniques for automatic malware analysis can be used in machine learning. Even pattern recognition techniques are used to detect malware (Nataraj L , Karthikeyan S , Jacob G , Manjunath B., 2011). while other works use generic algorithms for machine learning, such as perception, SVM, locality-sensitive hatching and decision trees (Nath HV , Mehtre BM., 2014). Before the various machine learning algorithms were used to classify malicious programs, the authors extracted network Access Functions, process execution, string handling, file manipulation and information reading (Afonso VM , de Amorim MF , Grégio ARA , Junquera GB , de Geus PL ., 2015). 100 features had been extracted from API calls, permissions, actions, and related strings of

numerous Android apps and space analysis through Eigen for the identification of malicious programs were applied (Yerima SY , Sezer S , Muttik I., 2015). Sahs and Khan have used Androguard to get Android apps ' permission and flow charts and have established an SVM-based model for characterizing Android malware (Sahs J , Khan L ., 2012).

In The work by researchers - Nikola Milosevic a , Ali Dehghantanha b , Kim Kwang Raymond Choo (Milosevic, N., Dehghantanha, A., & Choo, K.-K. R., 2017) they implemented an Android malware detection machine-learning model based on device allowances. This was a light, computer-intensive solution that could be used on various mobile phones. They also introduced a new approach to code assessment using machine learning. The work basically focused Android malware analysis approaches based on permission and source code. In their source code assessment approach they obtained accuracy results with multiple algorithms however their source code based approach did not address repackaging

2.2 Research Summary

Malware repackaging is a conspicuous issue where the raw source code of a mal-ware is reused as in another form. Android malwares in packaged form can be detected easily with antivirus engines but as in raw form they remain undetected. The interest of this research is to propose a model using third party software's and machine learning to address malicious pattern of the code by the means of source inspection , text processing and machine learning. Thus after conclusive study a certain process to address repackaging of the malware by proper automation of static analysis would be proposed by this research.

CHAPTER 3
RESEARCH METHODOLOGY

Malware repackaging is a conspicuous issue where the raw source code of a malware is reused as in another form. Android malwares in packaged form can be detected easily with The work by researchers Nikola Milosevic a , Ali Dehghantanha b , Kim-Kwang Raymond Choo (Milosevic, N., Dehghantanha, A., & Choo, K.-K. R., 2017) was considered as a base paper in this research because it focused on Android malware analysis with raw source code however the issue of repackaging was not addressed properly. It is a vital issue and addressing repackaging in a proper way, better efficiency and accuracy can be obtained on the source code based approach.

3.1 Method Representation

The base paper for this research focused on addressing analysis process through raw source code where apk file is transformed into zip file and zip file is extracted into .dex file (dalvik executable file), where dex is converted to jar file , jar to class files to obtain java codes and merging the java codes to process with bag of words algorithm and construct a dataset for machine learning to obtain results. This paper follows the basic procedure as them with some modification.

In the figure 3.1 the proposed model is shown. Relating to the base paper the dataset is collected from M0Droid dataset. This dataset contain package names and systems calls. Relating to the base paper, as they extracted 368 source codes, same has been done here. To conduct this procedure .apk files were collected accordingly as stated in the M0droid dataset. Then .apk files were converted to .zip format and classes.dex file was obtained

from the package. This proposed method suggests to analyze the .dex file through VirusTotal and identify the core file with VT graph to understand the core location of the infection. VirusTotal has a unique feature called VT graph which shows the exact location of infected file which is causing the maliciousness (VirusTotal. n.d. Retrieved from http://www.virustotal.com/). To obtain the jar file dex-to-jar was utilized as jar files were extracted using another windows based tool jd-gui. Java codes only from infected files has been merged for a dataset and machine learning but for to obtain better results from the base paper a customization of Bag of Words have been implanted and malicious patterns have been categorized by the means of a filtering process through a customized version of the algorithm. From the results obtained from the custom Bag of Words algorithm another filtered dataset was constructed to conduct machine learning on the dataset. The below figure 3.1 represents the methodology diagram.

After obtaining the exact file location, .java files from only that exact location were merged for the first dataset. In the base paper researches utilized all the source code to create dataset and applied Bag-of-Words algorithm, however in this paper some preprocessing were made to construct the final dataset. As some keywords and library functions never could be a part of malicious activities, thus it was focused to remove those from dataset. Moreover in this research when the Bag-of-Words algorithm was applied before that fresh keywords and malicious keywords were separated.

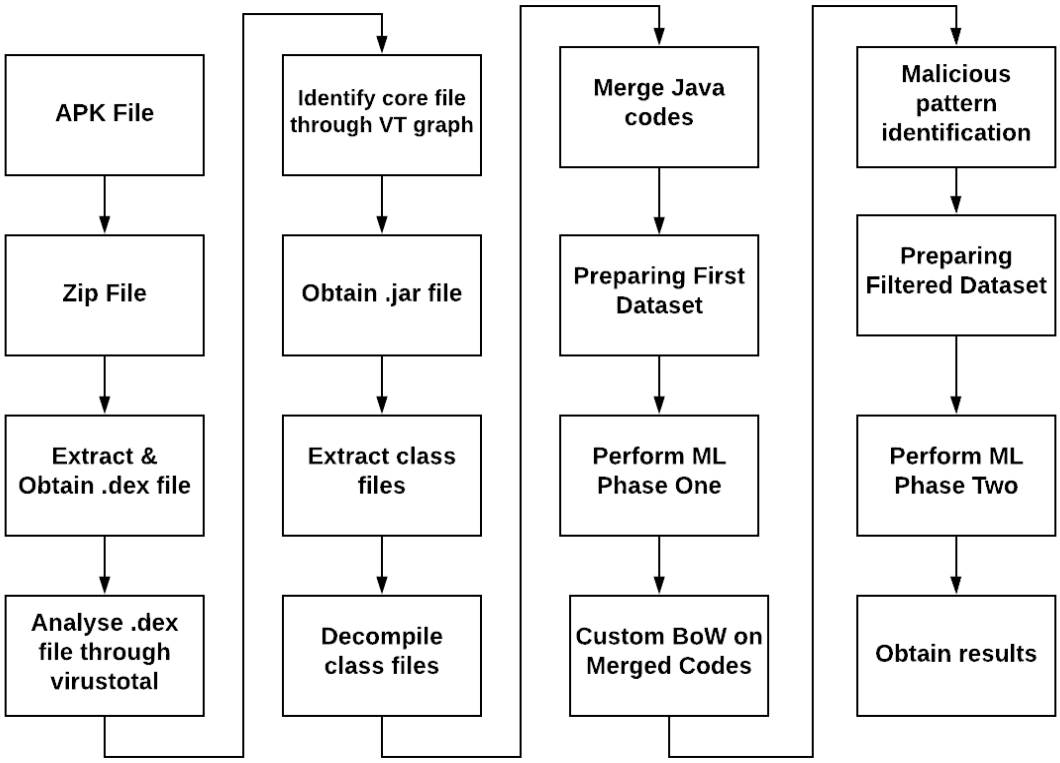


Figure 3.1: Methodology Diagram

3.2 Research Subject and Instrumentation

Let us discuss the method procedure on a sequential flow mentioning the detailed inspection of the whole research.

3.2.1 Obtaining the .dex file

The .apk file needs to be converted to .zip format, now by extracting the zip format the classes.dex file can be obtained. In certain cases there can be multiple classes.dex files depending on the application. The below figure shows the process:

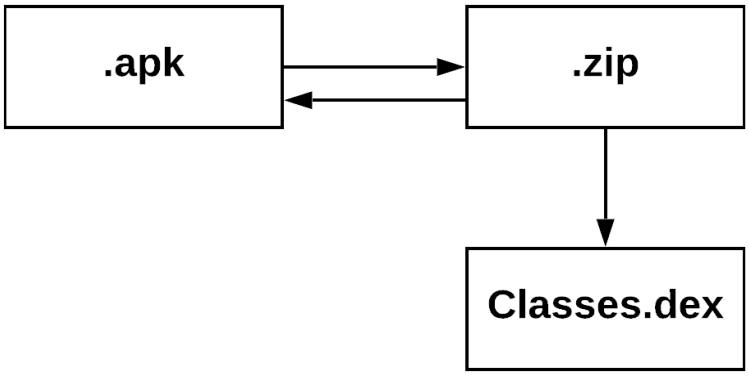


Figure 3.2: Obtaining .dex file

3.2.2 Analyze .dex file through VirusTotal

VirusTotal is an online service that analyzes files and URLs enabling the detection of viruses, worms, trojans and other kinds of malicious content using antivirus engines and website scanners. It also can be used to detect false positives. (VirusTotal. (n.d.). Retrieved from http://www.virustotal.com/.) When a .dex file is provided to VirusTotal it represents the status of the file immediately.

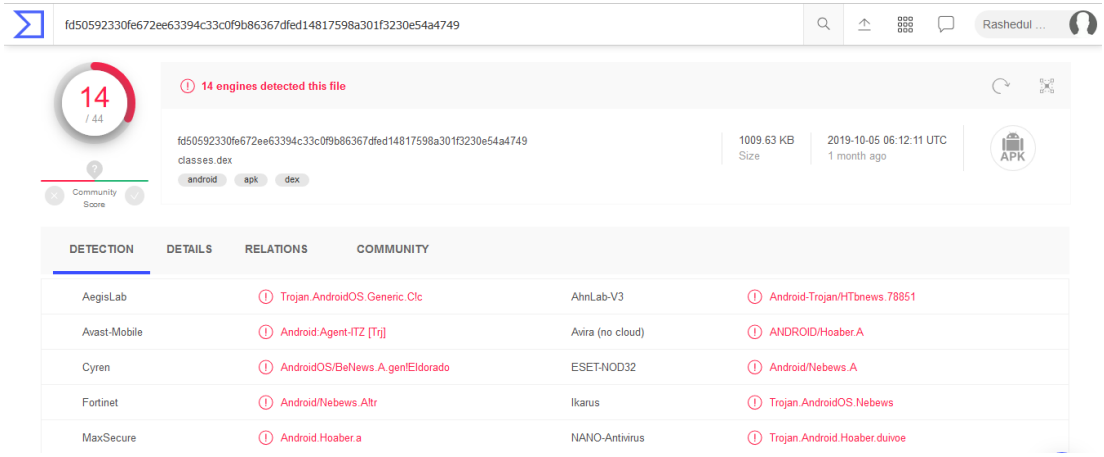


Figure 3.3: Status of .dex file

It has been a focal area for this research to address repackaging, because when in .dex format malicious behaviors are being detected by antivirus engines in VirusTotal, whereas when the .dex is decompiled as source code .java files they remain undetected, as the below figure which represents source files of the above .dex file.

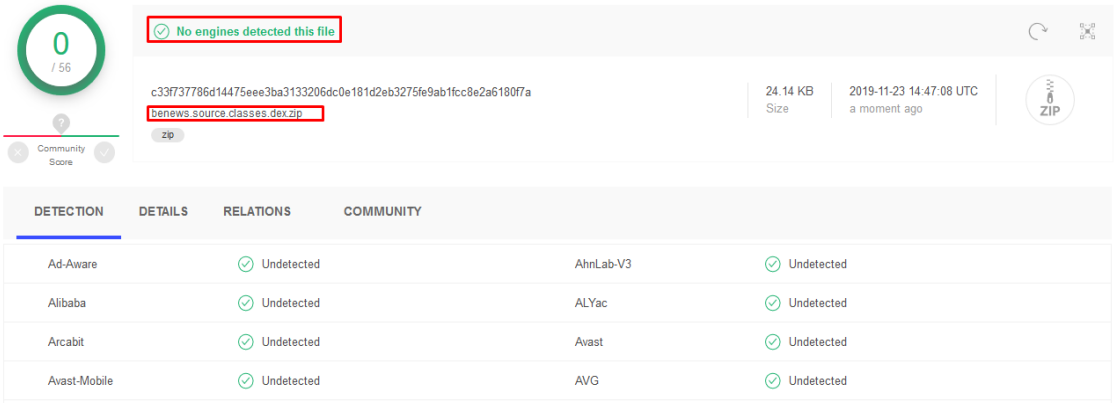


Figure 3.4: Raw source files remain undetected

Which is the area where we address the issue of malware repackaging via application.

3.2.3 Identify core file through VT graph

VirusTotal has a feature where it generates a graph summary of a scanned file through it's engine. The graph summary can be expanded and we can explore the set of files which are actually causing the maliciousness as of the below figure shows the files causing maliciousness are located on the org.benews location.

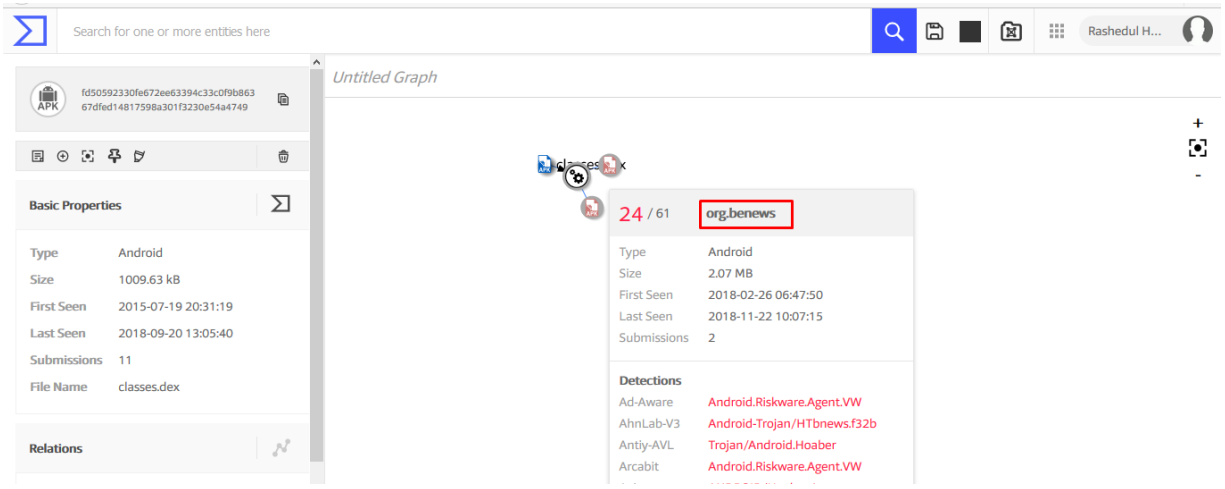


Figure 3.5: Identifying location of malicious file

3.2.4 Obtaining .jar file

The .dex file will now be converted to .jar file using Dex2jar tool. Which was also used by the authors of Machine learning aided Android malware classification (Milosevic, Nikola & Dehghantanha, Ali & Choo, Kim-Kwang Raymond., 2017.)

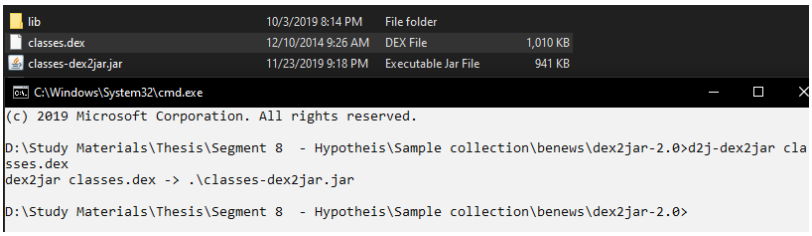


Figure 3.6: Obtaining .jar file from .dex file

To extract jar files and obtain class files a simple tool jd-gui has been utilized, from where java files can be saved and viewed in raw format. Expanding it will display the class files. In the below figure we are able to see the files in org.benews location which were identified by the VT Graph displaying malicious intent.

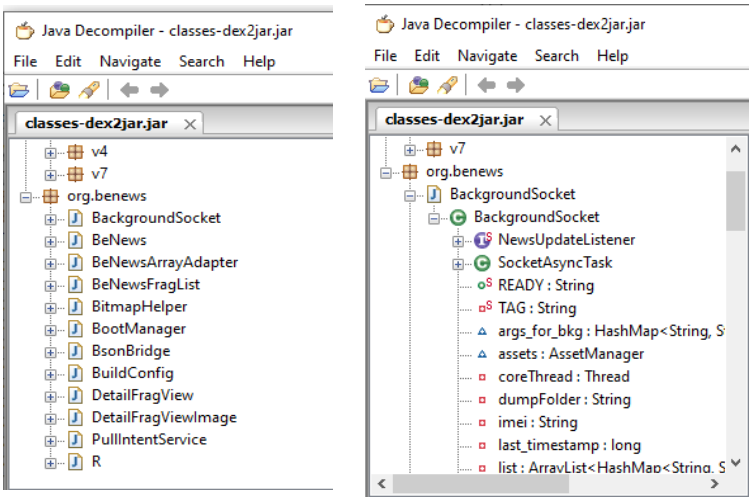


Figure 3.7: Processing .jar and .class files

3.2.5 Merging Java files

After obtaining all the java files all files will be merged as a single java file.

3.2.6 First Dataset

The first dataset would be prepared with raw code and labeled as per result obtained from VirusTotal. The attributes and the labeling procedure will be discussed in detailed in the next sub section.

3.2.7 Performing first phase of Machine Learning

After obtaining all the java files they will be merged as a single java file and the first phase of machine learning has been conducted. The following subsection will discuss the details procedure of implementation via machine learning.

3.2.8 Machine Learning process

As we have conducted our research through raw source code data, we conducted the procedure of Machine Learning through Python. We specifically utilized the Scikit Learn machine learning library for Python (Pedregosa et al., JMLR , 2011). As we have also mentioned about our Manual procedure with Bag of Words Algorithm we implemented generalized CountVectorizer and TfidfTransformer for feature extraction from the text for the raw code processing from the dataset. As stated in the previous chapter in BoW we can count or measure the frequency of words as a comparison with our vocabulary, CountVectorizer and TfidfTransformer is just an automated form when we are working with a huge amount of data. We implemented train_test_split in the script for dividing the data into training set and testing set for better efficiency. We also implemented functions like confusion_matrix, classification_report and accuracy score to obtain better results from the test and train data. We processed the data through excel file where direct raw data was taken input for testing and training and the decision was given based on class. The below figure partially shows our source code implementation.


```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from pandas import ExcelWriter
from pandas import ExcelFile

from sklearn import preprocessing
Encode = preprocessing.LabelEncoder()
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import Pipeline
from sklearn import metrics
from sklearn.metrics import (brier_score_loss, precision_score, recall_score,
                             f1_score)
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import LeavePOut
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import StratifiedKFold
from subprocess import check_output
for clf in classifiers:


name = clf.__class__.__name__
text_clf = Pipeline([('vect', CountVectorizer()), ('tfidf', TfidfTransformer()),
              ('clf', clf),])
text_clf.fit(x_train, y_train)

predicted = text_clf.predict(x_test)
acc = metrics.accuracy_score(y_test,predicted)
print (name+' accuracy = '+str(acc*100)+'%')
print("\tPrecision: %1.3f" % precision_score(y_test, predicted))
print("\tRecall: %1.3f" % recall_score(y_test, predicted))
print("\tF1: %1.3f\n" % f1_score(y_test, predicted))

print(confusion_matrix(y_test, predicted))
print(classification_report(y_test, predicted))
print (name)
print(accuracy_score(y_test, predicted))


   acc_field = pd.DataFrame([[name, acc*100]], columns=result_cols)
   result_frame = result_frame.append(acc_field)


sns.set_color_codes("muted")
sns.barplot(x='Accuracy', y='Classifier', data=result_frame, color="r")

plt.xlabel('Accuracy %')
plt.title('Classifier Accuracy')
plt.show()
```

Figure 3.8: Script Implementation by Python and Scikit Learn

3.2.9 Machine Learning descriptive analysis

train_test_split was implemented in the script for dividing the data into training set and testing set for better efficiency. The metrics which have been used for the evaluation of the algorithms are accuracy, precision, recall and F1 Score, these are some very common metrics which are utilized in the text mining and machine learning communities. The below discusses more about accuracy, precision, recall , F1 score and confusion matrix.

a. Accuracy

Accuracy is one of the most spontaneous performance measuring parameter. It is basically a ratio of correctly labeled items with the total calculation of the measures. However only higher accuracy will not indicate the results have more efficiency . It also depends on the amount of positive and negative values in the total dataset. So for evaluating the total performance it is also necessary to obtain the results from other parameters like precision and recall.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \quad (3.1)$$

b. Precision

Precision can be considered as the ratio of predicted positive observations with comparison with total positive observations. This metric answers the question, of all the samples which are malware how many of them were labeled correctly. High precision indicates to low false positive rates.

$$\text{Precision} = \frac{TP}{TP+FP} \tag{3.2}$$

c. Recall

Recall can be considered as the ratio of predicted positive observations with comparison with all probable positive observations. Because False negative are actually labeled negative but they are actually positive. This metric answers the question, of all the samples which are truly malware were they actually labeled as malware.

$$\text{Recall} = \frac{TP}{TP+FN} \tag{3.3}$$

d. F1 Score

F1 score or F score which is the combination of precision and recall. Therefore it can regarded as the weighted average of precision and recall. It takes both positive values and the negative values into account. Well about F1 score – it can be more precisely defined than accuracy. Accuracy can be taken as a singular measure perhaps when the samples or instances are equal. However when they are not precision and recall are to be checked properly.

$$\text{F1 Score} = \frac{(1+\beta^2) \times \text{Recall} \times \text{Precission}}{\beta^2 \times Recall \times Precission} \quad (3.4)$$

β indicates a comparative value for precision. A value of β = 1 (which is usually utilized) stipulates a comparative equal value of recall and precision. A lower value signifies a larger emphasis on precision and a higher value signifies a larger emphasis on recall. (Hersh, W., 2005)

e. Confusion Matrix

A matrix of confusion is an overview of results of forecast on a problem of classification. The number of acceptable and unacceptable predictions are illustrated and subdivided by each class with count values. This is the main barrier to the matrix of confusion. The confusion matrix demonstrates that, when making assertions, the classification method becomes confused. It gives

insight about not only into a classifier's errors, but more importantly the types of inaccuracies being made. It can be divided into following categories:

- True Positive /TP = Items labeled positive and they are actually positive
- False Positive/FP = Items labeled positive but they are actually negative
- False Negative/FN = Items labeled negative but they are actually positive.
- True Negative/TN = Items labeled negative and they are truly negative

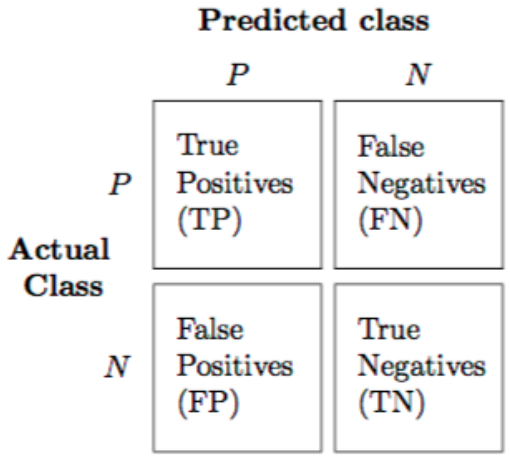


Figure 3.9: Confusion Matrix

3.2.10 Machine Learning Algorithm Implementation

SVM is an algorithm for supervised learning systems which can be used for problems of classification or regression. A label / Group is estimated by classification and a continuous value is anticipated by regression. SVM categorizes the hyperplane that distinguishes the classes which have been traced in n dimension space.

By transforming the data using mathematical functions called "kernels," SVM is drawing such a hyperplane. Kernel categories comprise linear, sigmoid, RBF, nonlinear, polynomial, etc. Basically — "RBF" is used to contend with non-linear problems, and is also a common kernel, if the data are therefore not previously known. For linear separable problems, the kernel — "linear" emphasized. This results have been obtained conducting "linear SVM" as the problem here is linear (only two variants Malware and NOT Malware).

3.2.11 Discussing Bag of Words Algorithm

The source code based research conducted in Machine learning aided Android malware classification (Milosevic, Nikola & Dehghantanha, Ali & Choo, Kim-Kwang Raymond. ,2017) used Bag of Words algorithm for text processing. The method they implemented with this technique took the text, or Java source code as a bag or set of words which ignoring the grammar or word order. Their approach considered the whole code including import statements, method calls, function arguments, and instructions.

So basically Bag of words model is a technique of Natural Language Processing or NLP to extract features from text or perhaps words from sentences. They way it is performed is by counting the frequency of words in a document. A document or referred as a corpora

can be defined as per the subject. It can be as large as the Wikipedia or short as two sentences. Basically it is a process of Tokenization, Counting and Normalization as the below figure represents:

Figure 3.10: BoW Representation

3.2.12 Implementation of Bag of Words

Basically below steps are to taken into account for the implementation of Bag of Words algorithm.

a. Collection of data from the source.

Providing information for further procedure, In case of sentences they can be comma separated.

b. Designing the vocabulary to address the data

Creating a list of all words to construct a vocabulary of words for developing a possible dictionary.

c. Generating Document vectors ,

The goal is to transform any free text document into a vector that can be used for the input or output of a learning model of the system. The simplest scoring method is to mark the presence of words as a Boolean value, 0 for absent, 1 for present.

d. Managing vocabulary based on requirements

Easy cleaning techniques for texts are available as a first step, for example:

- Ignoring case
- Ignoring punctuation
- Ignoring frequent words stop words, like “a,” “of,” etc.
- Fixing misspelled words.
- Reducing words

e. Scoring words by the means of count or frequency

The number of time a word will appear that needs to be scored against the formation of vocabulary.

There are few simple methods like:

- **Count:** Count each word in a document the number of times it appears.
- **Frequencies:** Calculating frequency of words as how frequency they appear in the document

f. BoW Example

The following models a text document using bag-of-words very simply . Here are two simple text documents:

(1) Kevin likes to watch cartoon. Sandra likes cartoon too.

(2) Kevin also likes to watch football games.

Based on these two text documents, a list is constructed as follows for each document:

"Kevin","likes","to","watch","cartoon","Sandra","likes","cartoon","to"

"Kevin", "also", "likes", "to", "watch", "football", "games"

Representing each bag-of-words as a particular object and attributing to the respective variable:

BoW1={"Kevin":1,"likes":2,"to":1,"watch":1,"cartoon":2,"Sandra":1,"too":1}

BoW2={"Kevin":1,"also":1,"likes":1,"to":1,"watch":1,"football":1,"games":}

If these sentences are being union-ed now:

(3) John likes to watch movies. Mary likes movies too. John also likes to watch football games.

It’s combined representation will be:

BoW3={"John":2,"likes":3,"to":2,"watch":2,"movies":2,"Mary":1,"too":1,"also":1,"football":1,"games":1};

So now it will depend on the context for further assessment or procedure such as machine learning how the contents should be taken. They can either be counted as per sentence or the frequency can be measured. We will discuss about the practical implementation and automated module representation of these in the next chapter.

3.2.13 Customizing Bag of Words

For the proposed model - after studying the generalized algorithm and implantation of it in the work Machine learning aided Android malware classification (Milosevic, Nikola & Dehghantanha, Ali & Choo, Kim-Kwang Raymond. ,2017) it was decided to customize the algorithm. Particularly customizations were made on three segments, let them be sequentially discussed:

a) The process of collecting data

Basically the obtained data was the raw source code, which has been fetched as a particular line to compare it with the vocabulary. Each line has to be minimal by quotation marks and separated by comma . So a python script was prepared by to take each line in the file into consideration and separate them accordingly. After it was completed the script would generate an output file for it. The below figure shows the script:


```
import sys

fname= sys.argv[1]

file= open(fname.replace('.txt','')+"_result.txt",'w')

with open(fname) as f:
    lines = f.readlines()
for line in lines:
        line=line.replace('\n','')
        wr='"'+line+'",'
        file.write(wr+'\n')
```

Figure 3.11: Custom script for processing text

Here a piece of code has been represented which will run through the script

```
import android.telephony.PhoneStateListener;
import android.telephony.TelephonyManager;
import android.util.Log;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class IncomingCall
extends BroadcastReceiver
{
public Context ctx;

 public void onReceive(Context paramContext, Intent paramIntent)
{
 this.ctx = paramContext;
try
{
((TelephonyManager)paramContext.getSystemService(phone)).listen(new MyPhoneStateListener(), 32);
 return;
}
catch (Exception paramContext)
{
Log.e(Phone Receive Error + paramContext);
}
}
public class MyPhoneStateListener
extends PhoneStateListener
{
public MyPhoneStateListener() {}
public String getcmd(Context paramContext, String paramString)
{
 return paramContext.getSharedPreferences(Cmd_conf, 0).getString(paramString, )
```

```
D:\Study Materials\Thesis\Segment 9 - Dataset\factory>python make.py sample.txt

D:\Study Materials\Thesis\Segment 9 - Dataset\factory>
```

Figure 3.12: Executing the script

After executing the script the below output would have been obtained as result.

```
"import android.telephony.PhoneStateListener;",
"import android.telephony.TelephonyManager;",
"import android.util.Log;",
"import java.util.Iterator;",
"import java.util.Map;",
"import java.util.Set;",
"public class IncomingCall",
"extends BroadcastReceiver",
"{",
"public Context ctx;",
"",
" public void onReceive(Context paramContext, Intent paramIntent)",
"{",
" this.ctx = paramContext;",
"try",
"{",
"((TelephonyManager)paramContext.getSystemService(phone)).listen(new MyPhoneStateListener(), 32);",
" return;",
"}",
"catch (Exception paramContext)",
"{",
"Log.e(Phone Receive Error + paramContext);",
"}",
"}",
"public class MyPhoneStateListener",
"extends PhoneStateListener",
"{",
"public MyPhoneStateListener() {}",
"public String getcmd(Context paramContext, String paramString)",
"{",
" return paramContext.getSharedPreferences(Cmd_conf, 0).getString(paramString, )",
"}",
```

Figure 3.13: Result after executing the code

b) Generating and Managing the Vocabulary

After assessing the basic Bag of Words a customized script was prepared according to the research necessity. It has three segments. Tokenizing sentences, extracting words from sentences and generating a format of Bag of Words. Python modules like numpy and re were utilized in the script. Words would be extracted from sentences where words would be ignored which will not appear in the vocabulary. The cleaned text would appear in the

corpora for tokenizing and sorting the words and finally bag of words will be generated based on the given input what is in this case are the source codes.

The below figure displays the prepared script

```
import numpy
import re

def tokenize(sentences):
    words = []
    for sentence in sentences:
        w = word_extraction(sentence)
        words.extend(w)

    words = sorted(list(set(words)))
    return words

def word_extraction(sentence):
    ignore = [  ]
    words = re.sub("[^\w]", " ",  sentence).split()
    cleaned_text = [w.lower() for w in words if w not in ignore]
    return cleaned_text

def generate_Bag_Of_Words(allsentences):
    vocab = tokenize(allsentences)
    print("Word List for Document \n{0} \n".format(vocab));

    for sentence in allsentences:
        words = word_extraction(sentence)
        bag_vector = numpy.zeros(len(vocab))
        for w in words:
            for i,word in enumerate(vocab):
                if word == w:
                    bag_vector[i] += 1

        print("{0} \n{1}\n".format(sentence,numpy.array(bag_vector)))


allsentences = [

]


generate_Bag_Of_Words(allsentences)
```

Figure 3.14: Custom BoW Script

c) The Process of Counting

Each line given will be compared with the vocabulary and a matrix will be generated, if the word is present it will appear as 1 in the matrix referring to it's presence in the vocabulary, if it is not present it will be represented as 0.

Example:

Let's consider Bag of Words is to be performed with the script to a piece of code which has been prepared as comma separated with by the previous script, Considering if any listener class implemented in the code such as android.telephony.PhoneStateListener in generic and specialized form then it will be considered as malicious as it has been refereed and found in multiple malicious instances (Palmer, D, 2017) and put it's result on the second dataset, here words and characters can be ignored in the code such as 'import', 'extends', '{', 'try', 'return', 'String', 'Log', 'exception','0', '32','ctx', 'e', 'this', 'util', 'void' etc. so it will be as below figure


```
import numpy
import re

def tokenize(sentences):
    words = []
    for sentence in sentences:
        w = word_extraction(sentence)
        words.extend(w)

    words = sorted(list(set(words)))
    return words

def word_extraction(sentence):
    ignore = [ 'import', 'extends', '{', 'try', 'return','String','Log' ]
    words = re.sub("[^\w]", " ",  sentence).split()
    cleaned_text = [w.lower() for w in words if w not in ignore]
    return cleaned_text

def generate_Bag_Of_Words(allsentences):
    vocab = tokenize(allsentences)
    print("Word List for Document \n{0} \n".format(vocab));

    for sentence in allsentences:
        words = word_extraction(sentence)
        bag_vector = numpy.zeros(len(vocab))
        for w in words:
            for i,word in enumerate(vocab):
                if word == w:
                    bag_vector[i] += 1

        print("{0} \n{1}\n".format(sentence,numpy.array(bag_vector)))

 allsentences = [
     "import android.telephony.PhoneStateListener;",
 "import android.telephony.TelephonyManager;",
 "import android.util.Log;",
 "import java.util.Iterator;",
 "import java.util.Map;",
 "import java.util.Set;",
 "public class IncomingCall",
 "extends BroadcastReceiver",
 "{",
 "public Context ctx;",
 " public void onReceive(Context paramContext, Intent paramIntent)",
 "{",
 " this.ctx = paramContext;",
 "try",
 "{",
 "((TelephonyManager)paramContext.getSystemService(phone)).listen(new MyPhoneStateListener(), 32);",
 " return;",
 "}",
 "catch (Exception paramContext)",
 "{",
 "Log.e(Phone Receive Error + paramContext);",
 "}",
 "}",
 "public class MyPhoneStateListener",
 "extends PhoneStateListener",
 "{",
 "public MyPhoneStateListener() {}",
 "public String getcmd(Context paramContext, String paramString)",
 "{",
 " return paramContext.getSharedPreferences(Cmd_conf, 0).getString(paramString, )",
 "}",

 ]


 generate_Bag_Of_Words(allsentences)
```

Figure 3.15: Implementing script

So the below will be generated as output where it can be seen, the vocabulary list and the matrix comparison by sentences and the words ignored will be excluded in the corpora. It can be seen that **android.telephony.PhoneStateListener** has been identified in multiple instances as a generic form and in specialized from.

```
Word List for Document
['android', 'broadcastreceiver', 'catch', 'class', 'cmd_conf', 'context', '
error', 'exception', 'getcmd', 'getsharedpreferences', 'getstring', 'getsys
temservice', 'incomingcall', 'intent', 'iterator', 'java', 'listen', 'map',
'myphonestatelistener', 'new', 'onreceive', 'paramcontext', 'paramintent',
'paramstring', 'phone', 'phonestatelistener', 'public', 'receive', 'set', '
telephony', 'telephonymanager']

import android.telephony.PhoneStateListener;
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 1. 0. 0. 0. 1. 0.]

import android.telephony.TelephonyManager;
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 1. 1.]

import android.util.Log;
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

import java.util.Iterator;
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

import java.util.Map;
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

import java.util.Set;
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 1. 0. 0.]
```

```
((TelephonyManager)paramContext.getSystemService(phone)).listen(new MyPhoneStateListener(), 32);
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 1. 0. 0.
 1. 0. 0. 0. 0. 0. 1.]

 return;
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

}
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

catch (Exception paramContext)
[0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

{
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0.]

Log.e(Phone Receive Error + paramContext);
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.
 1. 0. 0. 1. 0. 0. 0.]
```

Figure 3.16: Results Obtained / Output

So from this output it can be stated the source code as malicious for the designated application and label it as **"Malware",** regardless the decision obtained from VirusTotal.

3.2.14 Filtering the Dataset

Now the job is to filter the source code of each collected sample to prepare the second dataset. For this job fresh keywords are needed to be identified which are to be exempted from the vocabulary , and need to identify the malicious ones which will remain in the vocabulary set to detect most of the malicious system calls, functions and modules. So for this purpose two tables have been constructed which will display fresh and malicious words by a specific category.

a. Fresh Keywords Categorization:

Though there are some common techniques for the procedure (PeerzadaPeople, S. , 2019) such as :

- Get Rid of Extra Spaces
- Select and Treat All Blank Cells
- Convert Numbers Stored as Text into Numbers
- Remove Duplicates
- Highlight Errors
- Change Text to Lower/Upper/Proper Case
- Spell Check
- Delete all Formatting

Categorized as the below tabular format.

Table 3.1: Fresh keyword categorization

Stop Words	**Variable**	**Number**	**Default Keywords**	**Unicode and characters**
“the”,	“localobject”,	'2130771992',	“public”,	'0x7f', '0xff', ,'0d',
“is”	“action_bar__margin”,	'2130771993',	“super”,	‘{’, ‘}’,
“in”	“search_view”,	'2130771994',	“catch”,	“@”,”#”,”%”,
“for”,	“ppy_menu”,	'2130771994',	“throws”,	“+”,”-”,
“where”,	“min-width”,	'2130771995',	“protected”,	“*”, “%”, “^”,
“when”,	“max-width”,	'2130771996',	“extends”,	$
“how”,	“ theme_appcompat”,	'2130771997',	“java”, “io”	~!
“to”,	“flagsize”	'2130771998',	“private”,	?, /
“txt”	“title_text_size”,	'2130771999',	“system”,	___
“png”	“widget”	2130772001',	“super”,	Blank Space
“.jpg”	“action_bar__margin”,	'2130772000', '	“float”,	Line Gaps
“A-Z”	“webview”	'2130772002',	“integer”,	More special characters
“a-z”	“websettings”	'2130772003',	“Boolean”	
All vocabularies	System variables	'2130772004',	“import”	
File Extensions	User Defined Variables	All other system call numbers	“using” etc	

b. Malicious keywords categorization:

For the reference of malicious keywords, malicious keywords were kept in the generated vocabulary for every source code, so malicious system calls, functions , methods and packages can be identified. For categorizing reference of java system calls packages and methods from the work of different researchers were obtained. Some references of API calls were obtained from Static detection of Android malware by using permissions and API calls (Chan, P. P. K., & Song, W.-K. ,2014).

Type	API calls
Privacy related API calls	getSimSerialNumber() getSubscriberId() getLine1Number() getCellLocation() getNetworkOperator()
Network related API calls	httpclient.execute() getOutputStream() getInputStream() getNetworkInfo() HttpURLConnection.connect() openStream() setDataAndType() setRequestMethod()
SMS related API calls	sendTextMessage() sendDataMessage()
Wifi related API calls	getWifiState() setWifiEnabled() getWifiState()
Components related API calls	startService()

Figure 3.17: Collection of API call sets

A reference of functions were identified for a particular malware variant from the work - Obfuscated Malware Detection Using API Call Dependency (Patanaik, C. K., Barbhuiya, F. A., & Nandi, S. ,2012)

1	GlobalAlloc	GetModuleFileName, CreateFile,UnmapViewOfFile GlobalFree
2	GetModuleFileName	
3	CreateFile	GetFileSize, CreateFileMapping, UnmapViewOfFile, CloseHandle
4	GetFileSize	UnmapViewOfFile
5	CreateFileMapping	MapViewOFile
6	MapViewOfFile	GlobalAlloc, UnmapViewOfFile
7	GlobalAlloc	lstrlen
8	lstrlen	
9	UnmapViewOfFile	
10	CloseHandle	
11	CloseHandle	
12	GlobalFree	

Figure 3.18: Functions from a malware variant

Again few classes were identified which indicates malicious capabilities from the context of Examining Features for Android Malware Detection (Leeds, M., Keffeler, M., & Atkison, T. , 2017)

CLASSES INDICATIVE OF MALICIOUSNESS

Rank	Class	Weight
1	java.net.SocketException	0.309
2	java.lang.StringBuffer	0.303
3	java.lang.Character	0.294
4	javax.crypto.Cipher	0.282
5	java.io.ByteArrayInputStream	0.193
6	java.lang.UnsatisfiedLinkError	0.165
7	java.net.Proxy	0.161
8	java.util.Hashtable	0.161
9	java.util.zip.InflaterInputStream	0.157
10	java.util.GregorianCalendar	0.153

Figure 3.19: Malicious classes by the means of weight

Another research GroddDroid: a Gorilla for Triggering Malicious Behaviors (Abraham, A., Andriatsimandefitra, R., Brunelat, A., Lalande, J.-F., & Tong, V. V. T., 2015) categorized malicious classes by categories.

Table 1: Proposed risk scores by class categories

Category	Associated classes	Score
SMS	android.telephony.SmsManager	50
Telephony	android.telephony.TelephonyManager	20
Binary	java.lang.Runtime java.lang.Process java.lang.System	10
Dynamic	dalvik.system.BaseDexClassLoader dalvik.system.PathClassLoader dalvik.system.DexClassLoader dalvik.system.DexFile	10
Reflection	java.lang.reflect.*	3
Crypto	javax.crypto.* java.security.spec.*	3
Network	java.net.Socket java.net.ServerSocket java.net.HttpURLConnection java.net.JarURLConnection	3

Figure 3.20: Malicious keywords by categories

For obtaining the rest of possible java API calls, functions, methods and classes source code which were identified malicious by VirusTotal was inspected and manually separated . A paid survey was conducted to identify few more malicious java API calls, functions, methods and classes. It was conducted through a freelancing platform (www.freelancer.com) few questions were placed as of a survey format. Most of the participants were male having an age group over thirty. From the results of the survey few attributes were identified which can be regarded malicious in a possible sense. They have been represented in a tabular format categorized indecisively.

API Calls:

Table 3.2: API Call set Table 1

Privacy	**Network**	**SMS**
getSimSerialNumber(),	httpclient.executeO	sendTextMessageO
getSubscriberId(),	getOutputStreamO	sendDataMessageO
getCellLocation(),	getInputStreamO	
getNetworkOperator()	getNetworkInfoO	
	HttpURLConnection.connectO	
	openStreamO	
	setDataAndTypeO	
	setRequestMethodO	

Table 3.3: API Call set Table 2

WIFI	**Component Based**
getWifiState()	startService()
setWifiEnabled()	

Functions:

Table 3.4: Function speculated for maliciousness

By Allocation	**By Aspect**	**By Extraction**	**By Module**
GlobalAlloc	GetModuleFileName	UnmapViewOfFile	CreateFile
GlobalFree	GetFileSize	CreateFileMapping	CloseHandle

Classes:

Table 3.5: Android java classes Table 1

SMS	**Telephony**	**Binary**
android.telephony.SmsManager	android.telephony.TelephonyManager	java.lang.runtime
android.telephony.SmsMessage	android.os.IBinder	java.lang.process
	android.os.IBinder. FIRST_CALL_TRANSACTION	java.lang.system
	android.location.LocationListener	

Table 3.6: Android java classes Table 2

Dynamic	**Reflection**
dalvik.system.BaseDexClassLoader	java.lang.reflect.*
dalvik.system.PathClassLoader	java.lang.StringBuffer
dalvik.system.DexClassLoader	java.lang.Character
dalvik.system.DexFile	java.io.ByteArrayInputStream
dalvik.system.BaseDexClassLoader	java.lang.UnsatisfiedLinkError
	java.io.FilterOutputStream
	java.lang.Runtime

Table 3.7: Android java classes Table 3

Crypto / Security	**Connection**
javax.crypto.*	java.net.Socket
java.security.spec.*	java.net.ServerSocket
java.net.Proxy java.util.Hashtable	java.net.HttpURLConnection
java.security.Signature	java.net.JarURLConnection
java.security.SecureRandom	java.net.Socket
	java.net.SocketException
	java.net.MalformedURLException
	java.net.Proxy
	java.net.SocketTimeoutException

Table 3.8: Android java classes Table 4

Network	**Other**
android.net.NetworkInfo	android.content.pm.ApplicationInfo
android.net.NetworkInfo.State	android.content.pm.PackageManager
android.net.Proxy	android.provider.ContactsContract.Contacts;
	android.view.View.OnClickListener
	java.io.ByteArrayOutputStream
	java.io.InputStream

Based on the above API call requests, functions and classes patterns were identified and source code was filtered , in some cases API call requests, functions and classes were accompanied by different attributes which were also regarded malicious based on the context. Though this was a list which was generated however the malicious keywords categorization is not limited to this API call requests, functions and classes, It is basically based on the sample nature and sources.

3.2.15 Constructing New Dataset

After malicious pattern from the source code were identified source codes were labeled again with the decision obtained from the filtering procedure regardless it is same or not same with VirusTotal.

3.2.16 Performing Machine Learning Second Phase

After the second set is prepared again machine learning had been conducted to obtain the final results. It has been discussed elaborately in the next chapter.

3.3 Data Collection Procedure

For collecting applications the research M0Droid Dataset was obtained, M0Droid had been stated as a comprehensive behavior detection technique based on Android malware consisting of a lightweight client and server analyzer (Damshenas M , Dehghantanha A , Choo K-KR , Mahmud R, 2015). The data was generated using real Android APKs as input, and signatures as output. The dataset comprises these signatures. The tool to generate the signatures is designed by the authors, and is called the server analyzer. This dataset was also utilized in Machine learning aided Android malware classification (Milosevic, Nikola & Dehghantanha, Ali & Choo, Kim-Kwang Raymond, 2017), This dataset has 400 applications listed with package name where 200 are stated malicious and 200 are stated benign. package names were obtained from the dataset and .apk files were collected accordingly for the research.

However 368 .apk files were successfully decompiled to finally obtain their java source for the research and the 1st and 2nd dataset is comprised with the source code of 368 instances.

3.4 Preprocessing and Labeling

The datasets which were created extracting and collecting .apk packages consisted mainly of four attributes

- response_id = manual response ID no
- package = Package Name
- class = Malware / NOT_Malware
- raw_code = Raw malware source code

as the main concentration was to process raw java code , a python script was developed which would process the raw code and provide decision based on the class. There were only two classes

- Malware
- NOT_Malware

The datasets were trained and results were obtained according to this criteria. To process both the datasets which consisted of raw source code. A custom python script was compiled for machine learning which will be discussed in the next chapter.

3.5 Implementation Requirements

The data collection , processing and machine learning has been completed with a computer of below hardware and software configuration

Table 3.9: Implementation requirements – Windows

	Hardware Requirements
Operating System	Windows 10
Installed Ram	8 Gigabytes
Processor	Intel Core i3-6006U CPU @ 2.00 Ghz
HDD	1 Terabyte
	Software Requirements
Language	Python
Framework	Python 2.7
Modules	Scikit Learn Machine Learning library

Again to reconfirm the results obtained from machine learning algorithm , the machine learning procedure was conducted on a computer with the following configuration

Table 3.10: Implementation requirements - macOS

	Hardware Requirements
Operating System	macOS Catalina
Installed Ram	8 Gigabytes
Processor	Intel Core i7 processor
HDD	1 Terabyte
	Software Requirements
Language	Python
Framework	Python 2.7
Modules	Scikit Learn Machine Learning library

To implement the dictionary, a tool was designed by using php, CSS, HTML and mySql database. This application is supposed to take the source code as user input, search for malicious pattern and check it with the database. If any malicious pattern is identified then it would be categorized as malicious and shown to the user in the application front end. Below figures 4, 5, 6 show the application working procedure.

CHAPTER 4

RESULTS AND DISCUSSION

After collection of data in a concentric approach, the job is to measure the performance and making speculation of proper accuracy. It is also needed to be understood how the results has been mathematically reasoned. How much of the data has been correctly classified and how much incorrectly.

4.1 Experimental Results

As stated before the dataset of M0Droid (Damshenas M , Dehghantanha A , Choo K-KR , Mahmud R) was utilized from where application packages were extracted and collected, source codes for 368 samples were successfully decompiled . Both of the datasets were constructed based on that. First dataset which was basically the results obtained from VirusTotal showed below results. The implementation criteria of machine learning followed classification methods,

Table 4.1 : ML results from the final dataset

Algorithm	**Accuracy**	**Precision**	**Recall**	**F1**
SVM (SVC) Linear	95.65	0.5	0.5	0.5

4.2 Results Comparison

Basically the target approach was automating the process of analyzing source code by the means of machine learning. Partially similar research was conducted on Machine learning aided Android malware classification (Milosevic, Nikola & Dehghantanha, Ali & Choo, Kim-Kwang Raymond, 2017). The certain research also utilized the M0droid dataset mentioned before. The research focused on permission based classification with permission based clustering as also source code based classification with source code based clustering. As the prepared datasets for the thesis were supervised , the main focus was to make a comparison based on classification oriented machine learning.

Machine learning aided Android malware classification (Milosevic, Nikola & Dehghantanha, Ali & Choo, Kim-Kwang Raymond, 2017) obtained their best results for source code based classification with SVM as the following table represents.

Table 4.2 : ML results from existing work

Algorithm	**Accuracy**	**Precision**	**Recall**	**F1**
SVM with SMO	95.1	0.952	0.951	0.951

Comparison with obtained results from this proposed methodology

Table 4.3 : Results comparison

Parameters	Existing model	Proposed Model	Comparison
Algorithm	**SVM**	**SVM**	Same
Accuracy	95.1	95.65	0.55% better
Precision	0.952	0.5	0.452 Increase by
Recall	0.951	0.5	0.451 Increase by
F1	0.951	0.5	0.451 Increase by

From the result comparison it can be clearly stated that the proposed model has a better accuracy than the existing model, which displays at least a 0.55% better accuracy, Recall is also 0.452 more and both precision and F1 score is 0.451 more than the best accuracy results obtained in the similarly existing model. Which proves the proposed method provides better results than existing model.

4.3 External malicious pattern observation and categorization

Which proves the proposed method provides better results than existing model. After obtaining malicious patterns from different references mentioned in the previous chapter from the malicious source code phase one dataset , which was created with the help of VirusTotal VTGraph the following Methods and call requests were categorized as malicious. The certain methods, classes and call requests and functions were obtained from the malicious code snippets , where their activity indicated occurrence of malicious intent with the code.

Table 4.4 : Collected entities from observation

MessagesContentSender	getPasswordAuthentication()
fetchContact()	fetchContacts()
GetappInfo()	Android.telephony.phoneStateListener
IMAdTrackerReceiver()	
onStartCommand()	
onTerminate()	
PendingIntent.getService()	
InstallActivity.this.startActivity()	
getPassword()	

Considering **MessagesContentSender**, this particular functions retrieves Message Content and sends it to third party , even this function was identified in malware variants in multiple instances (Gbrindisi. n.d. gbrindisi/malware). Again **fetchContact()** fetches contact information and sends it to third party connection which was identified in multiple malicious samples, it's basically a method which is utilized to fetch the information and send it to receiver (Abraham, A., Andriatsimandefitra, R., Brunelat, A., Lalande, J.-F., & Tong, V. V. T., 2015). Another one **GetappInfo()** which obtains application information and it was also identified in malware samples in multiple instances (https://www.hybrid-analysis.com/sample/0d8645e25046b66799350b572d4c83a0fc63584794f00849aa7165e8af29cb25?environmentId=200.) . It showed malicious intent in the obtained source codes. Again **IMAdTrackerReceiver()** is an external broadcast receiver to invoke other classes as it has also been identified in multiple malicious instances (https://www.hybrid-analysis.com/sample/0ec8ef71bf113535b169721d8b2b72b53530476d1f746077ca85303ce385529e?environmentId=200).

Again **onStartCommand()** has been identified as an initialization command and also it was identified in Trojan source code accused for spaying messages from WhatsApp (Android Trojan reads Whatsapp-Messages 2018, October 1). Again **onTerminate()** was identified in multiple instances terminating a course of action for subduing malicious intention in certain phases. Again **PendingIntent.getService()** has been found in multiple malicious sources. PendingIntend is a token is given to a foreign application that enables foreign application to use the permission from the application to perform a default code piece. It is used by NotifyManager, AlarmManager, Home Screen AppWidgetManager, or by other 3rd party applications. It has also been identified in malware instances (ch0psticks. (n.d.), An analysis of android). Again **InstallActivity.this.startActivity()** focuses on installing a particular activity installing it at a sheath fashion which has been found in multiple sources as from it's particular behaviors. (https://www.hybrid-analysis.com/sample/0c7304075f0fdded9db3915139c00b86b98eba698a53bea4a4b1b7d3c80a004c?environmentId=200). Again **getPassword()** has been invoked in many Malware instances where this has been utilized as a malicious intent to retrieve password where's it could have been taken as getText() to retrieve the string (Kreider, N. K. N. ,1962, May 1). Again **getPasswordAuthentication()** was identified on multiple malware sources where it invokes password authentication multiple times, it has been identified for its malicious behavior in multiple instances (Oracle Java Mail SMTP Header Injection. n.d.). fetchContacts() has been identified retrieving contacts from the infected device. Finally **android.telephony.PhoneStateListener** has been distinguishes in generic and specialized form which has been considered as malicious as it has been refereed and found in multiple malicious instances (Palmer, D, 2017).

4.4 Development of a practical application

After few measurements a practical GUI based application was developed using PHP. HTML, CSS and MySQL. This application is supposed to take the source code as user input, search for malicious pattern and check it with the database. If any malicious pattern is identified then it would be categorized as malicious and shown to the user in the application front end. Below figures show the application working procedure.

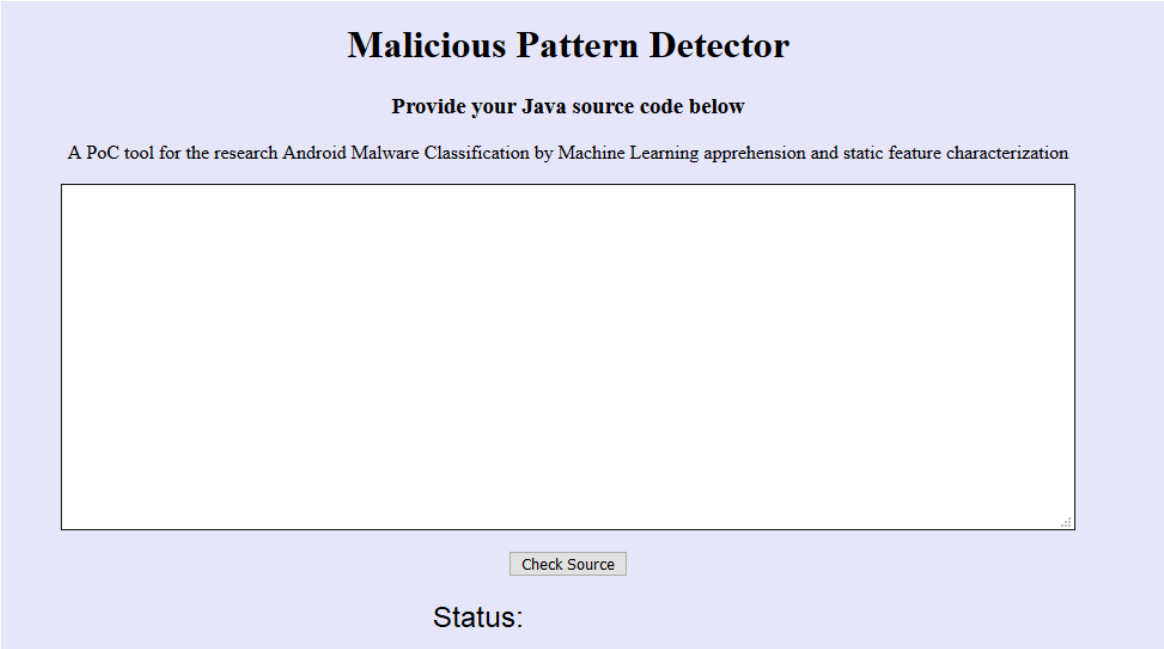


Figure 4.1: Application UI

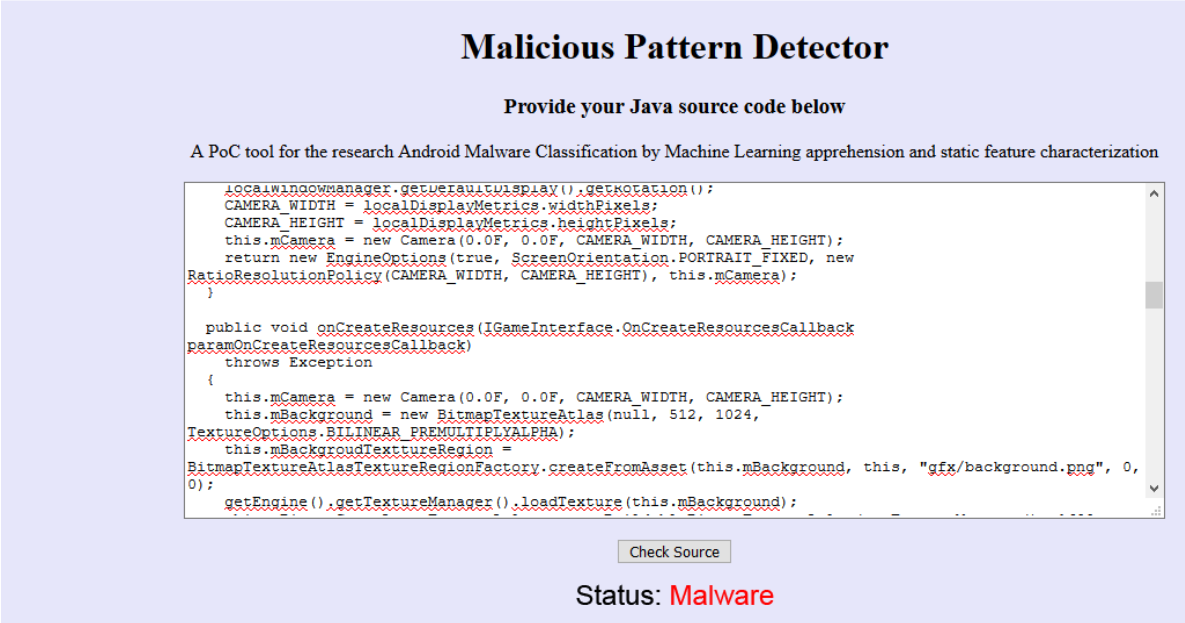


Figure 4.2: Malicious Pattern Identification

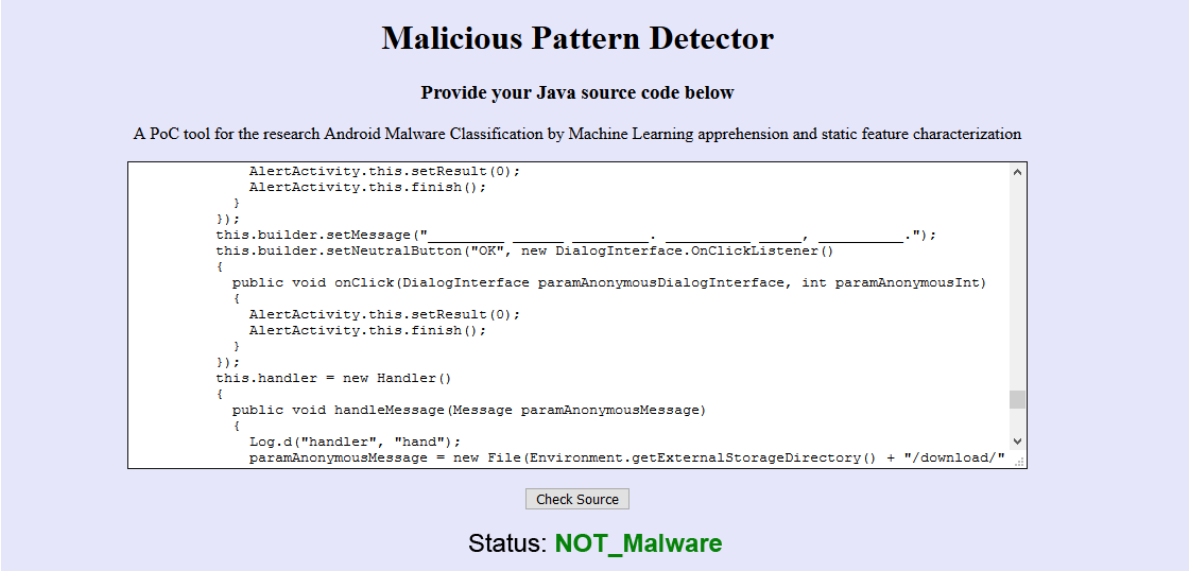


Figure 4.3: Non malicious Pattern Identification

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

The whole research basically focused on the overall mechanism of static analysis and addressing the prevention of source code repackaging of malwares. Existing works in the field provided scheme for the development of a model which may provide better results in case of efficiency and accuracy.

5.1 Findings and Contributions

The proposed model addresses repackaging in a proper way . Normal static analysis does not properly addresses the issue which has been the main goal of this research. Automating static analysis method by the means of classification algorithm (SVM) was another agenda in terms of accuracy and performance with comparison with few of the similar research in the field. This approach takes focus on API calls, functions, classes , methods and statements in generic and specialized forms to detect malicious pattern rather than few fixed defined features what has been observed in similar researches. Moreover the proposed model has obtained 0.55% better accuracy than a similar classification based adaptation on source code.

5.2 Recommendations for Future Works

Proposed method may create bit computational overhead on a larger implementation as it requires complete de-compilation of source code, however detailed analysis of the source code ensures presence of malicious pattern in a dynamic course of action. Future research direction includes:

- Combining multiple machine learning algorithm to increase efficiency.
- Increasing the criteria of identifying malicious pattern
- Developing a dictionary of malicious and fresh keywords for better accuracy.

REFERENCES

Cappers, B. C., Meessen, P. N., Etalle, S., & Wijk, J. J. V. (2018). Eventpad: Rapid Malware Analysis and Reverse Engineering using Visual Analytics. *2018 IEEE Symposium on Visualization for Cyber Security (VizSec)*. doi: 10.1109/vizsec.2018.8709230

Navarro, L. C., Navarro, A. K., Grégio, A., Rocha, A., & Dahab, R. (2018). Leveraging ontologies and machine-learning techniques for malware analysis into Android permissions ecosystems. *Computers & Security*, *78*, 429–453. doi: 10.1016/j.cose.2018.07.013

Boxall A. The number of smartphone users in the world is expected to reach a giant 6.1 billion by 2020; 2015 http://www.digitaltrends.com/mobile/smartphone- users- number- 6- 1- billion- by-2020/ .

Dehghantanha, A., & Franke, K. (2014). Privacy-respecting digital investigation.*2014 Twelfth Annual International Conference on Privacy, Security and Trust*. doi: 10.1109/pst.2014.6890932

Walls, J., & Choo, K.-K. R. (2015). A Review of Free Cloud-Based Anti-Malware Apps for Android.*2015 IEEE Trustcom/BigDataSE/ISPA*. doi:10.1109/trustcom.2015.482

Kitagawa M , Gupta A , Cozza R , Durand I , Glenn D , Maita K , et al. Market share: final pcs, ultramobiles and mobile phones, all countries, 2q15 update Technical Report; 2015 .

Chia, C., Choo, K.-K., & Fehrenbacher, D. (2017). How Cyber-Savvy are Older Mobile Device Users? *Mobile Security and Privacy*, 67–83. doi: 10.1016/b978-0-12-804629-6.00004-3

Viennot, N., Garcia, E., & Nieh, J. (2014). A measurement study of google play. *ACM SIGMETRICS Performance Evaluation Review*, *42*(1), 221–233. doi: 10.1145/2637364.2592003.

Sharma, M., Chawla, M., & Gajrani, J. (2016). A Survey of Android Malware Detection Strategy and Techniques. *Advances in Intelligent Systems and Computing Proceedings of International Conference on ICT for Sustainable Development*, 39–51. doi: 10.1007/978-981-10-0135-2_4

Buennemeyer, T. K., Nelson, T. M., Clagett, L. M., Dunning, J. P., Marchany, R. C., & Tront, J. G. (2008). Mobile Device Profiling and Intrusion Detection Using Smart Batteries. *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. doi: 10.1109/hicss.2008.319

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., … Sheth, A. N. (2014).TaintDroid. *ACM Transactions on Computer Systems*, *32*(2), 1–29. doi: 10.1145/2619091

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., & Cavallaro, L. (2016). DroidScribe: Classifying Android Malware Based on Runtime Behavior. *2016 IEEE Security and Privacy Workshops (SPW)*. doi: 10.1109/spw.2016.25

Alam, M. S., & Vuong, S. T. (2013). Random Forest Classification for Detecting Android Malware. *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. doi: 10.1109/greencom-ithings-cpscom.2013.122

Isohara, T., Takemori, K., & Kubota, A. (2011). Kernel-based Behavior Analysis for Android Malware Detection. *2011 Seventh International Conference on Computational Intelligence and Security*. doi: 10.1109/cis.2011.226

Mercaldo, F., Nardone, V., Santone, A., & Visaggio, C. A. (2016). Download malware? no, thanks. *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering - FormaliSE 16*. doi: 10.1145/2897667.2897673

Karbab, E. B., Debbabi, M., & Mouheb, D. (2016). Fingerprinting Android packaging: Generating DNAs for malware detection. *Digital Investigation*, *18*. doi: 10.1016/j.diin.2016.04.013

Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware images. *Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec 11*. doi: 10.1145/2016904.2016908

Nath, H. V., & Mehtre, B. M. (2014). Static Malware Analysis Using Machine Learning Methods.*Communications in Computer and Information Science Recent Trends in Computer Networks and Distributed Systems Security*, 440–450. doi: 10.1007/978-3-642-54525-2_39

Afonso, V. M., Amorim, M. F. D., Grégio, A. R. A., Junquera, G. B., & Geus, P. L. D. (2014). Identifying Android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, *11*(1), 9–17. doi: 10.1007/s11416-014-02267

Yerima, S. Y., Sezer, S., & Muttik, I. (2015). Android malware detection: An eigenspace analysis approach.*2015 Science and Information Conference (SAI)*. doi: 10.1109/sai.2015.7237302

Sahs, J., & Khan, L. (2012). A Machine Learning Approach to Android Malware Detection. *2012 European Intelligence and Security Informatics Conference*. doi: 10.1109/eisic.2012.34

Milosevic, N., Dehghantanha, A., & Choo, K.-K. R. (2017). Machine learning aided Android malware classification. *Computers & Electrical Engineering*, *61*, 266–274. Doi 10.1016/j.compeleceng.2017.02.013

VirusTotal. (n.d.). Retrieved from http://www.virustotal.com/.)

Palmer, D. (2017, August 23). Over 500 Android apps with a combined 100 million downloads found to secretly contain spyware. Retrieved from https://www.zdnet.com/article/500-android-apps-found-to-secretly-contain-data-stealing-spyware/.

Chan, P. P. K., & Song, W.-K. (2014). Static detection of Android malware by using permissions and API calls.*2014 International Conference on Machine Learning and Cybernetics*. doi: 10.1109/icmlc.2014.7009096

[Patanaik, C. K., Barbhuiya, F. A., & Nandi, S. (2012). Obfuscated malware detection using API call dependency. *Proceedings of the First International Conference on Security of Internet of Things - SecurIT 12*. doi: 10.1145/2490428.2490454

Leeds, M., Keffeler, M., & Atkison, T. (2017). A Comparison of Features for Android Malware Detection. *Proceedings of the SouthEast Conference on - ACM SE 17*. doi: 10.1145/3077286.3077288

Abraham, A., Andriatsimandefitra, R., Brunelat, A., Lalande, J.-F., & Tong, V. V. T. (2015). GroddDroid: a gorilla for triggering malicious behaviors. *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. doi: 10.1109/malware.2015.7413692

Hersh, W. (2005). Evaluation of biomedical text-mining systems: Lessons learned from information retrieval. *Briefings in Bioinformatics*, *6*(4), 344–356. doi: 10.1093/bib/6.4.344

Gbrindisi. (n.d.). gbrindisi/malware. Retrieved from https://github.com/gbrindisi/malware/blob/master/android/gmbot/project/SlempoService/src/org/slempo/service/utils/MessagesContentSender.java.

(n.d.). Retrieved from https://www.hybrid-analysis.com/sample/0d8645e25046b66799350b572d4c83a0fc63584794f00849aa7165e8af29cb25?environmentId=200.

(n.d.). Retrieved from https://www.hybrid-analysis.com/sample/0ec8ef71bf113535b169721d8b2b72b53530476d1f746077ca85303ce385529e?environmentId=200.

Android Trojan reads Whatsapp-Messages. (2018, October 1). Retrieved from https://gotech.world/android-trojan-reads-whatsapp-messages/.

ch0psticks. (n.d.). An analysis of android. Retrieved from http://happyhacking.info/post/malware-analysis-android-adsms.

(n.d.). Retrieved from https://www.hybrid-analysis.com/sample/0c7304075f0fdded9db3915139c00b86b98eba698a53bea4a4b1b7d3c80a004c?environmentId=200.

Kreider, N. K. N. (1962, May 1). getText() vs getPassword(). Retrieved from https://stackoverflow.com/questions/9798066/gettext-vs-getpassword.

Oracle JavaMail SMTP Header Injection. (n.d.). Retrieved from https://packetstormsecurity.com/files/126721/Oracle-JavaMail-SMTP-Header-Injection.html.