ISSN 2722-2039

Adaptive Android APKs Reverse Engineering for Features Processing in Machine Learning Malware Detection

Benjamin Aruwa Gyunka [a], Aro Taye Oladele [b], Ojeniyi Adegoke [c,1,*]

[a] Department of Branch Operations Central Bank of Nigeria, Kano, Nigeria
[b] Department of Mathematical and Computing Sciences, KolaDaisi University, Ibadan, Nigeria
[c] Department of Computer Science, Maldives National University, Male, Maldives
[1] adegoke.ojeniyi@mnu.edu.mv
* corresponding author

ARTICLE INFO

Article history
Received November 25, 2021
Revised August 7, 2022
Accepted October 17, 2022

Keywords
Android
Android Application Package (APK)
reverse engineering
feature
attribute
malware
machine learning

ABSTRACT

The key component that makes the detection of android malware possible is the availability of the right triggers and pointers, which are found in the Android application packages, known as features or attributes. These are fundamental in the training of the different machine learning algorithms to produce the required detection model. The process of extracting these attributes or features, from the Android application packages, is known as reverse engineering. This paper delved into the experimental detail processes of applying reverse engineering procedure, using Sublime Text 2 and Androguard Plugin, on Android Application packages for the extraction of, particularly permissions, which are the targeted features. The study further discussed the cleaning stages, using NotePad++, Microsoft Excel Worksheet, and MS Word, to sort out all the relevant and important features by removing all the noisy ones. A total of 1500 Android apps were downloaded from both benign and malicious sources and used for the experiment. The cleaned or important features extracted from these application packages at the end of the reverse engineering processes are 162 in total and these were further used to form a feature binary matrix of size 1500 by 163 (including the class features).

1. Introduction

Smartphones and Tablets are cutting-edge technological advancements of the 21st Century that have unequivocally introduced rising security challenges in the field of computing and information systems. The emergence and proliferation of these Hand-held Devices are said to have almost reached an epidemic proportion [1]. The world's population study by the United Nations [2] revealed an estimated population of people worldwide as 7.71 billion and according to GSM Association (commonly known as Global System for Mobile Communication or simply GSMA) real-time intelligence data, there are now over 9.3 billion mobile connections worldwide as at September 2019, surpassing the current world population [3], [4]. These devices have the in-built capabilities of remaining constantly connected to the internet (due to the possession of features such as Wi-Fi, voice, data, GPS, etc.) and have continued, at an exponential rate, to win the confidence and trust of their users. Their emergence has simplified lots of social networking activities, commercial and banking transactions so much that they have become an integral part of our lives. The study by Andrew [5] noted that people now tend to trust these devices more with their secrets and corporate life than they do trust fellow humans. The manufacturers of these devices usually have distinct Operating Systems for their gadgets. For example, iOS for all Apple devices, RIM (Research in Motion) for Blackberry

Devices, Symbian and Windows for Nokia Devices, and Google Android which is adopted by over 1300 Mobile Device manufacturing companies, especially Samsung and HTC [6]. Google Android has become and remains the most proliferated and popular mobile device operating system in circulation [7].

The increasing proliferation nature of the Android OS, through different kinds of devices, has attracted so much attention from both legitimate and illegitimate (malicious) users. Malware developers have seen this as an opportunity to exploit the users of these devices through different malicious attacks. The study by Feng, Anand, Dillig, and Aiken [8] observed that Android malware is being released into the economy by these mischievous developers in rapid succession. Malware, by a simple definition, is an application that performs a function that is completely in variance with its original legitimate design. This brings about serious questions of trust on the developers of application software and also on the ability of an application to strongly resist any form of tampering (i.e., repackaging) by malicious attackers. The different mediums through which Android malware can be spread include Bluetooth Connections, Memory Cards, Wireless Networks, Universal Serial Bus (USB) Connections, Third-Party Apps providers, and something even through Google Play Store [9].

The field of Android Malware detection has seen lots of different techniques and methodologies, but in recent times, Machine learning techniques have brought about quite a lot of positive impacts. One major aspect or step in machine learning deployments is the collection or extraction of usable features out of the total collected data (Android Application Packages). The Android Application Packages (APKs) are like zipped files which have the .apk as their extensions. To extract and process the different features, a process call reverse engineering will be adopted and deployed. This paper focuses on the entire process of reverse engineering each one of the .apk files collected for the research purpose. Different techniques were deployed for constructing all the permission attributes extracted. The paper discusses the benign and malware data and the experiment carried out to analyze and process the collected data samples. It examined the steps and techniques used in the processes of data collection, and the procedures for extracting permission-based features from Android applications (benign/malicious).

2. Literature Review

Much work has been carried out in researching ways and measures of efficiently defeating Android malware [10], [11], [12], [13], [14], [15]. However, the procedure of reversed engineering of the collated data for analysis of Android malware has remained fundamental of most machine learning methodologies adopted.

2.1 Reverse Engineering

Reverse engineering is a process of studying and analyzing the underlying source codes of any application or software to understand its functionalities and behaviors and to come up with a new or improved version [16]. This process opens up a platform for the investigation and analysis of the hidden characteristics, behaviors, or intents of any Android application and its undocumented internal principles. Eilam [17] defined it as the process of extracting the knowledge or design blueprints from anything man-made. He further explains that reverse engineering is usually conducted to obtain missing knowledge, ideas, and design philosophy when such information is unavailable. This is because, in some cases, the information is owned by someone who is not willing to share them, while in other cases; the information has been lost or destroyed. However, Aguilera [18] noted that the main purpose of this process is not to make changes or to replicate the system under analysis, but to physically dissect them to uncover the secrets of their design; understanding how they were built. Figure 1 illustrates the sketch of the step-by-step process of reverse engineering of an Android APK file for the extraction of required features.

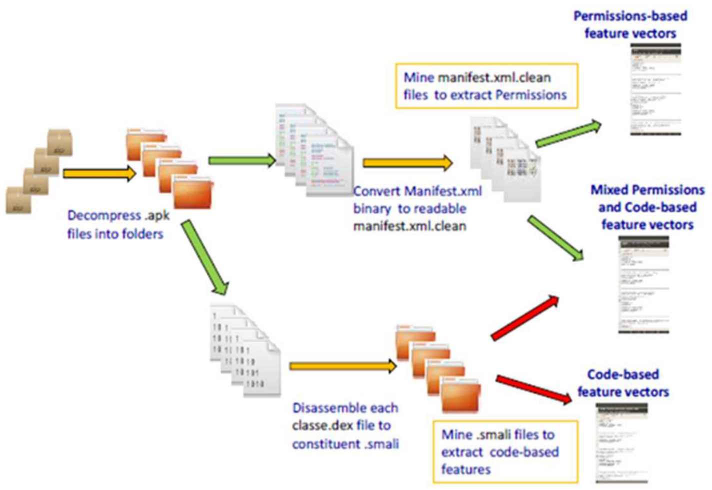

Permissions-based feature vectors
Mine manifest.xml.clean files to extract Permissions
Convert Manifest.xml binary to readable manifest.xml.clean
Decompress .apk files into folders
Mixed Permissions and Code-based feature vectors
Disassemble each classe.dex file to constituent .smali
Mine .smali files to extract code-based features
Code-based feature vectors

Figure 1. Android APK Reverse Engineering Steps [19]

According to Eilam [17], software or Applications reverse engineering integrates several arts: code-breaking, puzzle solving, programming, and logical analysis. Reverse engineering can be done in two different ways; via the static analysis techniques or the dynamic analysis technique. Static analysis technique involves analyzing a .apk file by going through a manual step-by-step stage of decompiling and studying the code and physical components or features of an application using the different static analysis tools. This technique is sometimes called code analysis and the static features collected from this procedure are the permissions and the API calls which can be found in the AndroidManifest.xml file. The collection or extraction of features through this procedure involves a manual process. Dynamic analysis, on the other hand, involves a process of studying the behavior of the program when under execution in a controlled environment and it is also referred to as behavioral analysis [20]. Features that can be extracted via this process include network traffic, battery usage, and IP addresses [21].

2.2 Tools used for Reverse Engineering APKs

The open nature of the Android platform has allowed for a variety of tools developed by different researchers for analysis. Amongst the several existing tools, the following are the most popularly available and used;

a. AXML2jar: for converting the manifest file into a readable format.
b. ApkTool: for decoding resources to the original form and can be used to rebuild it back after modification. It is also useful for transforming binary Dalvik bytecode (classes. dex) into Smali source.
c. Dex2Jar: for converting Dalvik bytecode (DEX) to java bytecode (JAR) and then allows the usage of any existing Java decompiler with the resulting JAR file.
d. Androguard: a python-based tool useful for disassembling and decompiling Android APK files. It is suitable for the Static Analysis of an Android application.
e. Smali/Baksmali: used to disassemble the .dex file Android SDK.
f. APK Inspectors: a GUI apk analysis tool.
g. Jd-gui: helps in converting a .jar file to .java.

The two distinct tools used in this study for reverse engineering APKs are Androguard and Sublime Text 2. The Sublime Text 2 was first installed in the workstation after which the Andrograd Plug-in was copied and extracted into C:\User\<User_name>\AppData\Roaming\Sublime Text 2\Packages. The Sublime Text 2 was then restarted to make both tools integrate and ready for use.

2.3 Related Word

Chen et al. [22] looked into ways in which sophisticated attackers can sabotage the effectiveness of deploying machine-learning classifiers for malware detection and analysis. They pointed out that these attackers mostly aim at polluting the dataset used for training the classifiers so that the detection model to be produced will be a weak one thus leading to detection inaccuracies. Dong [23] worked using permissions as his primary feature to develop a novel detection system for Android malware. To perform the reverse engineering process, the author developed a tool to decompile the Apps to source code and manifest files automatically. The study by Adebayo [24] centered his work on using an apriori algorithm to generate candidates (flagbearers) from the future set of Android applications for classification into the malicious or benign application. The work aimed at building a malware detection system by improving the apriori algorithm using particle swarm optimization and permission-based features of Android applications to improve the classification system and detection rate of malicious applications. The reverse engineering of the Android applications was done using apk-tool, Androgurad, Sublime Text, and dex2jar. The features extracted and constructed were used to train seven different machine learning classifiers. Apvrille and Apvrille [25] focused their work on how to identify and defeat the class of unknown malware known as the “0-day” malware. They developed a framework called SherlockDroid. To detect unknown malware, the SherlockDroid works by filtering masses of applications and only keeps the most likely to be malicious for future inspections. Apart from crawling applications from marketplaces, SherlockDroid extracts code-level features, and then classifies unknown applications using Alligator.

Yerima, Sezer, and McWilliams [19] developed and analyzed proactive Machine Learning approaches based on Bayesian classification aimed at uncovering unknown Android malware via static analysis. Permissions and code-based properties such as API calls, both Java system based and Android system based, Linux and Android system commands were also extracted from the sampled applications. They employed a Java-based custom-built APK analyzer for the reverse engineering of all the APK files. Yerima et al. [26] extracted Permissions, API calls, Naive Linux System commands features using a Java-based Android package profiling tool for automated reverse engineering of the APK files. The tool incorporated Baksmali for disassembling the .dex files into multiple files with smali extensions. Sahs and Khan [27] extracted permissions and Control Flow Graphs (CFG) as features and used the One-Class Support Vector Machine (SVM) for classification. The authors made use of the open-source project Androguard to reverse-engineering the different apk files and process the extraction of features.

3. Methodology

The developed adaptive android APKs reverse engineering models for feature processing in machine learning malware detection involved several steps. The steps and procedures taken to gather a set of data to perform reverse engineering are further discussed in the following sub-sections.

2.1 Dataset Collection

The malware datasets samples were downloaded from Contagio Mobile and Malware Genome while the benign datasets samples were downloaded from the official Android Apps Market and Google Play Store. The features used in the learning processes were extracted primarily from the collected data samples through reverse engineering procedures.

A total sample size of 1500 applications was downloaded for the investigation and analysis process. These involved 500 benign applications and 1000 malicious applications as shown in Table 1. The benign applications downloaded constituted 33.333% of the entire collection while the malicious applications samples made up 66.667%.

Table 1. Percentages of APKS Downloaded

File Type	Source	Quantity	Percentage	Usage
Benign Apps	Google Play Store	500	33.3333%	Analysis
Malicious Apps	Contagiominidump, Contagiodump	1000	66.6667%	Analysis
Total		*1500*	*100%*	*2362.5*

2.2 Analysis of the APK files Downloaded for Feature Extraction

VirusTotal, an online malware/virus scanning engine that uses a combined strength of over fifty (50) Antivirus/Antimalware software working together simultaneously, was used to scan each of the benign applications downloaded to fully certify their benign status. All the apps that were tested proved to be truly benign. Tables 2 and Table 3 show the file analysis of all the APK files that were downloaded and utilized for feature extractions. The benign and malicious applications were categorized into their different families and types per their respective numbers.

Table 2. Statistic of Malware Families Downloaded

MALWARE FAMILIES USED AND THEIR NUMBERS			
Family	No. of Samples	Family	No. of Samples
DroidDream	154	Fakebanker	3
AnserverBot	150	Coinkrypt	3
Geinimi	130	Godwin	3
PJApps	98	Android.Samsapo	2
MisoSMS	94	Dendroid	2
NickiSpy	98	KungFuvariant	2
SpyEra_aka_Tigerbot	85	Phospy	2
Zitmo_akaZeus	76	UsbCleaver	2
Sandroid	44	Android_Crackme#4	1
RootSmart	32	DroidKungFu.	1
IBanking	6	com.Beauty.Girl-1	1
BMaster	6	AntiObamaScan	1
SMSGoogle	3	com.Beauty.Breast-1	1

Table 3. Statistic Of Benign Applications Downloaded By Categories

Benign Applications Categories		
Application Category	Number of Samples	Apps Market
e-Commerce Apps	40	Google Play Store
Educational Apps	101	Google Play Store
Games Apps	64	Google Play Store
m-Banking Apps	77	Google Play Store
Music/Video Players	74	Google Play Store
News Apps	71	Google Play Store
Photo Editors	12	Google Play Store
Social Media	61	Google Play Store

2.3 Features Extraction Phase

Static reverse engineering procedures were used to mine features from the manifest.xml of each APK file. Figure 2 shows each stage of the apk reverse engineering and features extraction processes.

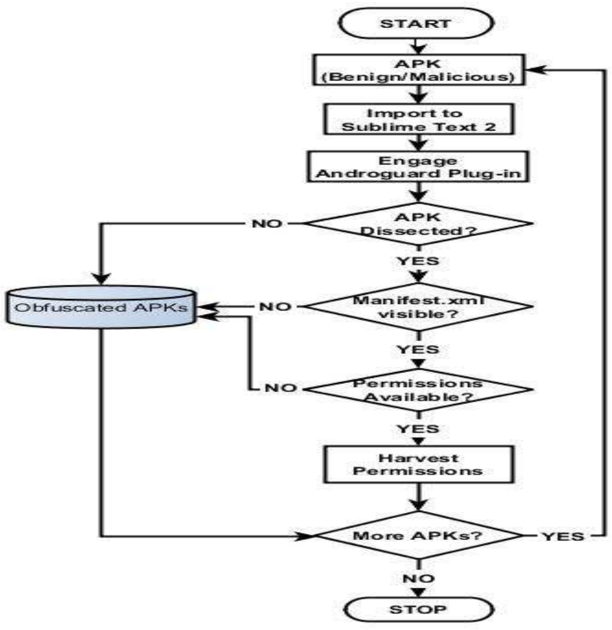


Figure 2. Reverse Engineering Stages for APK Files

Android Permissions were the only features considered in this study for the following reasons:

a. The process of gathering them has a low computing overhead;
b. The different features exhibited by most Applications are mostly declared at the permission section;
c. They have proven from research to positively influence Android malware detection rate to a very high degree [28], [29]

2.4 Cleaning of Extracted Features

After feature extraction, the next phase was to clean the features corpus by selecting the most relevant and unique ones through the removal of duplicates and those that appeared in single apks and then fine-tuning the needed ones to acceptable format as shown in Figure 3.

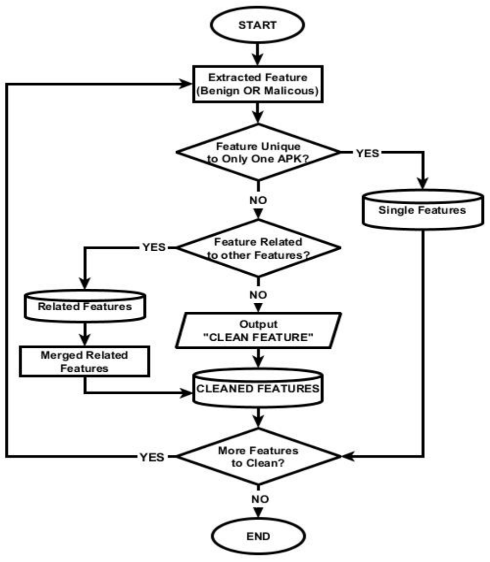


Figure 3. Features Cleaning Processes

2.5 Creation of Training Dataset

The features collected were transformed into binary/unary vectors format and represented in a matrix form. The process involved the deployment of mathematical set theory, having the Android application repository represents the Universal sets U_apps which comprises of Malicious applications M_apps and Benign applications B_apps such that both M_apps and B_apps are proper subsets of the universal set U_apps.

$$\left(B_{apps} \cup M_{apps}\right) \subset A \tag{1}$$

Where M_apps and B_apps represents features from Malicious and Benign applications and they are defined as follows:

$$M\_apps = (M\_1,M\_2,M\_3,M\_4,\ldots,M\_i) \tag{2}$$

$$B\_apps=(B\_1,B\_2,B\_3,B\_4,\ldots,B\_j\ ) \tag{3}$$

The interception of both subsets is defined as follows;

$$F\_(M \cap B)=M\_apps \cap B\_apps \tag{4}$$

The union of both subsets is defined as follows;

$$F\_(M \cup B)=M\_apps \cup B\_apps \tag{5}$$

To have a set that contains only the elements or features of the malicious applications, the following set difference was used.

$$F\_(M \backslash MB)=M\_apps \backslash M\_apps \cap B\_apps \tag{6}$$

To have a set that contains only the elements or features of the benign applications, the following set difference was used

$$F\_(B \backslash MB)=B\_apps \backslash M\_apps \cap B\_apps \tag{7}$$

To be able to compose a feature vector that has N-dimensional space (N)^D, it became necessary to use a combine feature of union sets F_(M∪B) as follows:

$$F\_(M \cup B)=(M\_1 \cup B\_1,M\_2 \cup B\_2,\cdots,M\_j \cup B\_j\ ) \tag{8}$$

Where (F_(M∪B))_zy∈L^j |y=1,2,⋯,k and z=1,2,⋯,j

The size of the input training matrix was defined as "The total number of applications used (both malicious and benign), also known as instances, by the total number of extracted features from the applications", also known as attributes, and the matrix was represented as follows;

$$F_{M \cup B} = \begin{bmatrix} (F_{M \cup B})_{11} & \cdots & (F_{M \cup B})_{1j} \\ \vdots & \ddots & \vdots \\ (F_{M \cup B})_{k1} & \cdots & (F_{M \cup B})_{kj} \end{bmatrix} \tag{9}$$

Where;

k=Total Number of Applications (instances) in the Data set

j=Total Number of cleaned Features (attributes)plus the class used

This was derived from the malicious and benign features Mapps and Bapps with each defined as follows;

Given that (M_apps)_yz∈L^j,for y=1,2,⋯,k;z=1,2,⋯,j the matrix representation is

$$M_{apps} = \begin{bmatrix} (M_{apps})_{11} & \cdots & (M_{apps})_{1j} \\ \vdots & \ddots & \vdots \\ (M_{apps})_{k1} & \cdots & (M_{apps})_{kj} \end{bmatrix} \tag{10}$$

Also, for (B_apps)_yz∈L^j,for y=1,2,…,k;z=1,2,…,j the matrix is represented as follows;

$$B_{apps} = \begin{bmatrix} (B_{apps})_{11} & \cdots & (B_{apps})_{1j} \\ \vdots & \ddots & \vdots \\ (B_{apps})_{k1} & \cdots & (B_{apps})_{kj} \end{bmatrix} \tag{11}$$

The majority of machine learning approaches work best using numerical vectors so the features in the matrix would need first to be mapped into a binary/unary vector space. From equation (8), a joint set F_(M∪B), was defined as M_j∪B_j represents the corresponding feature.

$$F_{M\cup B} = \{M_1 \cup B_1, M_2 \cup B_2, \dots, M_j \cup B_j\} \tag{12}$$

The F_(M∪B) was utilized and |F_(M∪B) | was defined as dimensional vector space, in which case each dimension is either 0 or 1. The feature (M∪B) was mapped to this space by constructing a vector V:

$$V = \{v_1, v_2, .., v_n\} \tag{13}$$

Such that:

$$v_i = \begin{cases} 1, & f_i \in (F_{M\cup B})_{zy} \\ 0, & f_i \notin (F_{M\cup B})_{zy} \end{cases} \tag{14}$$

Thus, the binary feature vector was translated into V={0,0,1,0,1,1,0,..} and the resulted (F_(M∪B))_zy is m x n vector-matrix which contained 1 and 0 as its elements. One (1) represented the presence of a feature in the given sample and zero (0) represented the absence of a feature in the sample. The feature sets were therefore a set of bits 0 and 1. This implied that, if an application contained a feature, then 1 was recorded for that feature and if it does not contain that feature, 0 was recorded. This implies that the following applies:

$$(M_{apps})_{yz}, (B_{apps})_{yz}, (F_{M\cup B})_{zy} \in (0,1) \tag{15}$$

Where {0,1}^n is a feature space of binary vectors for somen. Figure 4 shows the binary transformation process for the different features used.

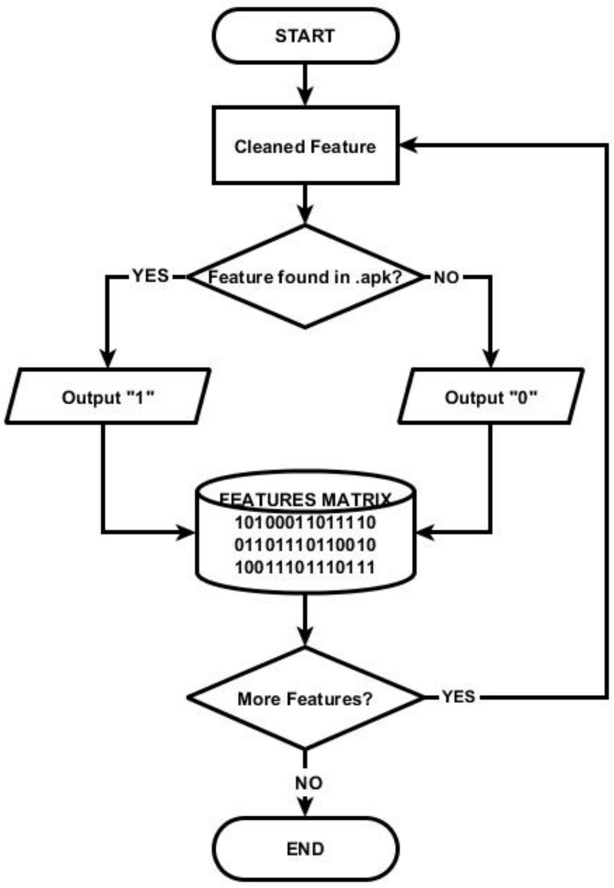


Figure 4. Feature Vectors Binary Transformation

2.5 Data Transformation Tools

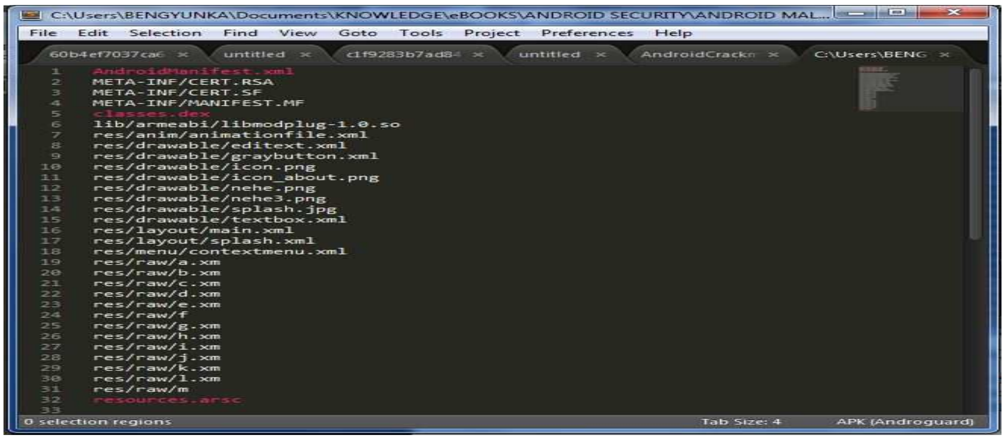
The features utilized in this study were text-based or categorical. The feature extraction was performed manually using Sublime Text 2 and Androguard and dumped in Microsoft Word 2017. These dumps were further processed into Microsoft Excel 2017 where they were separated into different categories and types. Excel provided a very essential platform for effective data cleaning, separation, and categorization in which the features were processed and transformed into numeric and binary digits formats. Notepad and Notepad++ were also very instrumental in the data cleaning and categorization processes.

4. Results and Discussion

This section provides and discusses all the results obtained from the reverse engineering of all the application packages obtained.

4.1 Results of Reversed Engineering Analysis

The result of an APK file that was dissected using Sublime Text 2 and Androguard plug-in is shown in Figure 5. Three key components shown in the result are the AndroidManifest.xml Classes.dex and resources.arsc. These components provide very critical definitive characteristics for any APK file. The AndroidManifest.xml houses the apps' permissions.

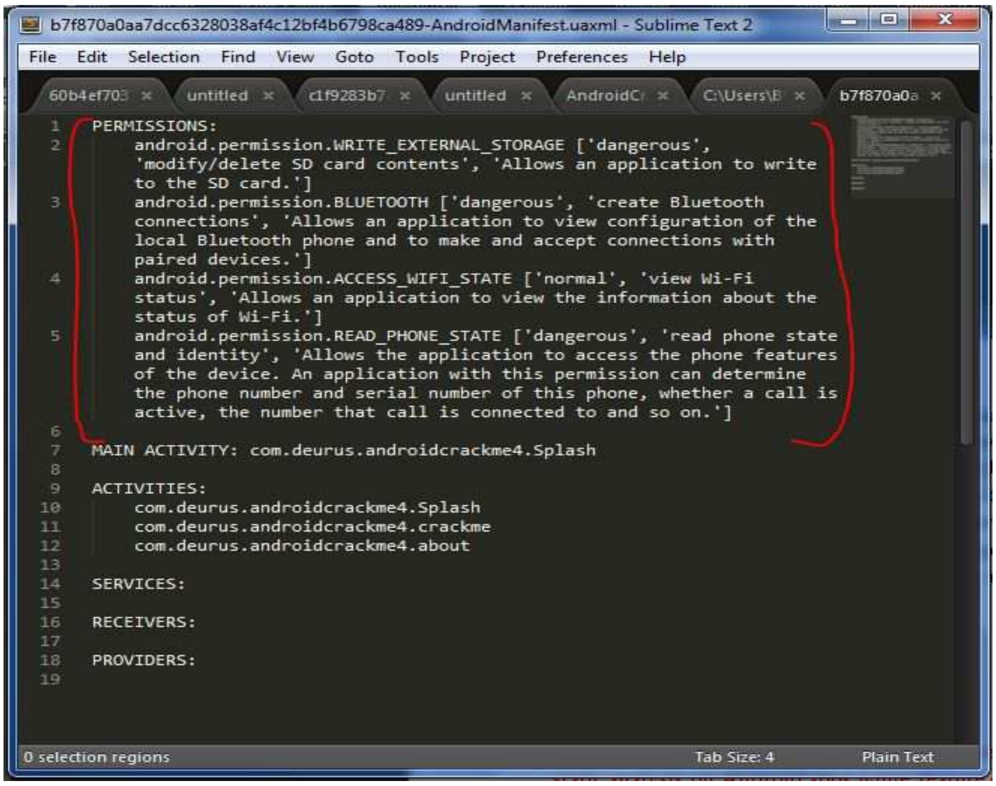


```
AndroidManifest.xml
META-INF/CERT.RSA
META-INF/CERT.SF
META-INF/MANIFEST.MF
classes.dex
lib/armeabi/libmodplug-1.0.so
res/anim/animationfile.xml
res/drawable/editext.xml
res/drawable/graybutton.xml
res/drawable/icon.png
res/drawable/icon_about.png
res/drawable/nehe.png
res/drawable/nehe3.png
res/drawable/splash.jpg
res/drawable/textbox.xml
res/layout/main.xml
res/layout/splash.xml
res/menu/contextmenu.xml
res/raw/a.xm
res/raw/b.xm
res/raw/c.xm
res/raw/d.xm
res/raw/e.xm
res/raw/f
res/raw/g.xm
res/raw/h.xm
res/raw/i.xm
res/raw/j.xm
res/raw/k.xm
res/raw/l.xm
res/raw/m
resources.arsc
```

C:\Users\BENGYUNKA\Documents\KNOWLEDGE\eBOOKS\ANDROID SECURITY\ANDROID MAL...
File Edit Selection Find View Goto Tools Project Preferences Help
60b4ef7037ca6
untitled
c1f9283b7ad8
untitled
AndroidCrackn
C:\Users\BENG
0 selection regions
Tab Size: 4
APK (Androguard)

Figure 5. Androgurad Analysis of APK file

A further dissecting of the AndroidManifest.xml, using the same tools, reveals all the permission features contained in the app as shown in Figure 6.



```
PERMISSIONS:
    android.permission.WRITE_EXTERNAL_STORAGE ['dangerous',
    'modify/delete SD card contents', 'Allows an application to write
    to the SD card.']
    android.permission.BLUETOOTH ['dangerous', 'create Bluetooth
    connections', 'Allows an application to view configuration of the
    local Bluetooth phone and to make and accept connections with
    paired devices.']
    android.permission.ACCESS_WIFI_STATE ['normal', 'view Wi-Fi
    status', 'Allows an application to view the information about the
    status of Wi-Fi.']
    android.permission.READ_PHONE_STATE ['dangerous', 'read phone state
    and identity', 'Allows the application to access the phone features
    of the device. An application with this permission can determine
    the phone number and serial number of this phone, whether a call is
    active, the number that call is connected to and so on.']

MAIN ACTIVITY: com.deurus.androidcrackme4.Splash

ACTIVITIES:
    com.deurus.androidcrackme4.Splash
    com.deurus.androidcrackme4.crackme
    com.deurus.androidcrackme4.about

SERVICES:

RECEIVERS:

PROVIDERS:
```

b7f870a0aa7dcc6328038af4c12bf4b6798ca489-AndroidManifest.uaxml - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
60b4ef703
untitled
c1f9283b7
untitled
AndroidCr
C:\Users\B
b7f870a0a
0 selection regions
Tab Size: 4
Plain Text

Figure 6. Inside AndroidManifest.xml

A result of a manually harvested permission dump of a single apk (belonging to the BMaster malware family) is shown in Figure 7. The result illustrated how practically a single application can contain any numbers of permissions defined in it. This manual extraction was performed for each one of the 1500 APK files and it is shown in Fig 8 as a text box:

```
    android.permission.READ_SYNC_SETTINGS ['normal', 'read sync settings', 'Allows an application to read the sync settings,
such as whether sync is enabled for Contacts.']
    android.permission.WRITE_APN_SETTINGS ['dangerous', 'write Access Point Name settings', 'Allows an application to modify
the APN settings, such as Proxy and Port of any APN.']
    android.permission.READ_SECURE_SETTINGS ['dangerous', 'Unknown permission from android reference', 'Unknown
permission from android reference']
    android.permission.RECEIVE_BOOT_COMPLETED ['normal', 'automatically start at boot', 'Allows an application to start itself
as soon as the system has finished booting. This can make it take longer to start the phone and allow the application to slow down the
overall phone by always running.']
    android.permission.BLUETOOTH ['dangerous', 'create Bluetooth connections', 'Allows an application to view the configuration
of the local Bluetooth phone and to make and accept connections with paired devices.']
    android.permission.ACCESS_WIFI_STATE ['normal', 'view Wi-Fi status', 'Allows an application to view the information about
the status of Wi-Fi.']
    android.permission.INTERNET ['dangerous', 'full Internet access', 'Allows an application to create network sockets.']
    android.permission.CHANGE_CONFIGURATION ['dangerous', 'change your UI settings', 'Allows an application to change the
current configuration, such as the locale or overall font size.']
    android.permission.BLUETOOTH_ADMIN ['dangerous', 'Bluetooth administration', 'Allows an application to configure the local
Bluetooth phone and to discover and pair with remote devices.']
    android.permission.ACCESS_FINE_LOCATION ['dangerous', 'fine (GPS) location', 'Access fine location sources, such as the
Global Positioning System on the phone, where available. Malicious applications can use this to determine where you are and may
consume additional battery power.']
    android.permission.HARDWARE_TEST ['signature', 'test hardware', 'Allows the application to control various peripherals for the
purpose of hardware testing.']
    android.permission.DEVICE_POWER ['signature', 'turn the phone on or off', 'Allows the application to turn the phone on or off.']
    android.permission.ACCESS_NETWORK_STATE ['normal', 'view network status', 'Allows an application to view the status of
all networks.']
    android.permission.GET_TASKS ['dangerous', 'retrieve running applications', 'Allows an application to retrieve information about
currently and recently running tasks. May allow malicious applications to discover private information about other applications.']      /
    android.permission.WRITE_SYNC_SETTINGS ['dangerous', 'write sync settings', 'Allows an application to modify the sync
settings, such as whether sync is enabled for Contacts.']
    android.permission.WRITE_SETTINGS ['dangerous', 'modify global system settings', "Allows an application to modify the
system's settings data. Malicious applications can corrupt your system's configuration."]
    android.permission.FLASHLIGHT ['normal', 'control flashlight', 'Allows the application to control the flashlight.']
    android.permission.READ_PHONE_STATE['dangerous', 'read phone state and identity', 'Allows the application to access the
phone features of the device. An application with this permission can determine the phone number and serial number of this phone,
whether a call is active, the number that call is connected to and so on.']
    android.permission.WRITE_SECURE_SETTINGS ['signatureOrSystem', 'modify secure system settings', "Allows an application
to modify the system's secure settings data. Not for use by normal applications."]
    android.permission.VIBRATE ['normal', 'control vibrator', 'Allows the application to control the vibrator.']
    android.permission.SYSTEM_ALERT_WINDOW ['dangerous', 'display system-level alerts', 'Allows an application to show
system-alert windows. Malicious applications can take over the entire screen of the phone.']
    android.permission.CAMERA ['dangerous', 'take pictures and videos', 'Allows application to take pictures and videos with the
camera. This allows the application to collect images that the camera is seeing at any time.']
    android.permission.WAKE_LOCK ['dangerous', 'prevent phone from sleeping', 'Allows an application to prevent the phone from
going to sleep.']
    android.permission.CHANGE_WIFI_STATE ['dangerous', 'change Wi-Fi status', 'Allows an application to connect to and
disconnect from Wi-Fi access points and to make changes to configured Wi-Fi networks.']
    android.permission.MODIFY_PHONE_STATE ['signatureOrSystem', 'modify phone status', 'Allows the application to control
the phone features of the device. An application with this permission can switch networks, turn the phone radio on and off and the
like, without ever notifying you.']
    android.permission.GET_ACCOUNTS ['normal', 'discover known accounts', 'Allows an application to access the list of accounts
known by the phone.']
```

Figure 7. Permission Features Dump from a Single Application

It was observed that some of the applications recorded no visible permissions while some others could not be decompiled by the reversed engineering tools. From the one thousand five hundred (1500) APK files decompiled, comprising of 1000 malicious and 500 benign, only one thousand four hundred and five (1405), comprising of Nine Hundred and Fifty-Two (952) malicious and Four Hundred and Fifty-Three benign (453), were successfully decompiled and contained visible permission features as shown in Table 4.1. From the remaining Ninety-Seven (95) APK files, eight (8) benign APK files were decompiled but had hidden or no visible permissions registered while Thirty Nine (39) benign APK files could not be decompiled by the Androguard. For the malicious APK files, four (4) were decompiled but had hidden or no permissions registered, while Forty Four (44) could not be decompiled by Androguard. The APK files that could not be decompiled could be for the reasons of obfuscation and encryption techniques deployed by malware writers to evade detection and to conceal their malicious codes from any kind of analysis. Table 4 gives a concise breakdown of the percentage compositions of the different instances (applications) used.

4.2 Duplicate features

Duplicate features are those found more than once in a particular android application or those erroneously placed twice in an application during the feature compilation. These features were identified and removed from the dataset.

4.3 Merged Features Output

Related features are those that are closely related in their permission definition and so can be merged according to their permission definitions. A total of three hundred and forty (340) benign related features and twenty-five (25) malicious related features were identified and extracted to Notepad. The related features were merged to produce an additional twelve (12) unique benign features and six (6) unique malicious features. A sample of related features extracted and ready for merging are as followed;

com.amazon.device.messaging.permission.RECEIVE
com.nokia.pushnotifications.permission.RECEIVE
com.nokia.pushnotifications.permission.RECEIVE
com.google.android.c2dm.permission.RECEIVE
com.amazon.device.messaging.permission.RECEIVE

Single Features Result

Single features are those that were found only in individual .apk files and not in any other. A total of one hundred and thirty-three (133) benign single features and forty-nine (49) malicious single features were extracted. A sample is shown below:

android.permission.READ_SYNC_STATS
android.permission.NETWORK_PROVIDER
android.permission.PROCESS_OUTGOING_SMS
android.permission.GLOBAL_SEARCH_CONTROL
android.permission.RECORD_VIDEO

Table 4. Analysis of The APKS Used

Android applications packages	Quantity	%
Total APKs utilized – both Benign and Malicious	1500	100
Total APKs with features	1401	93.6667
Total APKs without features	95	6.3333
Malicious APK files	1000	66.6667
Benign APKs	500	33.3333
Valid Benign APK	453	30.2
Valid Malicious APK	952	63.4667
Invalid Malicious APK	48	3.2
Invalid Benign APK	47	3.1333

Total Benign & Malicious Apps utilized: the whole total collection of applications downloaded and utilized.
Total APKs that contained features: the whole collection of apps that have visible permissions after being reversed engineered.
Total APKs that had no features: The whole collection of apps that had no visible permissions after being reversed engineered.
Malicious APK files: The whole collection of only malicious apps utilized in the study.
Benign APKs: The whole collection of only benign apps utilized in the study.
Valid Benign APK: The whole collection of only benign apps that contained permissions.
Valid Malicious APK: The whole collection of only malicious apps that contained permission.
Invalid Malicious APK: The number of only malicious apps that had no permissions.
Invalid Benign APK: The number of only benign apps that had no permissions.

4.4 Analysis of the Extracted Permission-based Features

Table 5 provides an overview of the number of permissions extracted from the different categories of applications and their formulation into diverse feature structures. The results show that a total of 763 permission-based features were extracted from all the downloaded applications via reversed engineering. 175 of these were extracted from the malicious applications which constituted 22.936% of the total extracted features. The benign applications had the largest number of 588 features, which is 77.064% of the total collections. Of these statistics, 107 was obtained as the total number for completely cleaned malicious features while 127 was obtained as the total number of completely cleaned benign features. The union of these completely cleaned benign and malicious features produced a total of 162 features (that is 21.232% of the total extracted features) required cleaned features ready to be used in the machine learning experiments implementation.

Table 5. Extracted Permission-Based Features Percentages

Feature Code	Features	Quantity	Extracted Features (%)	Used Features (%)
$\boldsymbol{AF}$	All Features	763	100	
$\boldsymbol{M}$	All Malicious	175	22.9357798	
$\boldsymbol{B}$	All Benign	588	77.0642202	
$\boldsymbol{M_S}$	Single Malicious Features removed	49	6.42201835	
$\boldsymbol{B_S}$	Single Benign Features removed	133	17.4311927	
$\boldsymbol{C}$	Class	1		
$\boldsymbol{TUF}$	Total Used Features	163	21.6251638	
$\boldsymbol{B_R}$	Related Benign Features removed	340	44.5609436	
$\boldsymbol{M_R}$	Related Malicious Features removed	25	3.27653997	
$\boldsymbol{PB_R}$	Processed RBF	12	1.57273919	7.40740741
$\boldsymbol{PM_R}$	Processed RMF	6	0.78636960	3.70370370
$\boldsymbol{M/(M_S \cup M_R)}$	Cleaned Malicious	101	13.2372215	62.3456790
$\boldsymbol{B/(B_S \cup B_R)}$	Cleaned Benign	115	15.07208388	70.9876543
$\boldsymbol{M_t}$	Total cleaned Malicious	107	14.02359109	66.0493827
$\boldsymbol{B_t}$	Total cleaned Benign	127	16.64482307	78.39506173
$\boldsymbol{M_t \cap B_t}$	M & B	66	9.04325037	40.74074074
$\boldsymbol{M/M_t \cap B_t}$	Malicious Alone	38	4.98034076	23.45679012
$\boldsymbol{B/M_t \cap B_t}$	Benign Alone	58	7.60157274	35.80246914
$\boldsymbol{M_t \cup B_t}$	M OR B	162	21.2319790	100

AF: Total number of features extracted from all apps.
M: Total features extracted only from malicious apps.
B: The total number of features extracted from only the benign apps.
M_S: Single features found in only one malicious apps that had been removed from the feature sets.
B_S: Single features found in only one benign apps that had been removed from the feature sets.
C: The category that provided the expected output for each instance of an applications.
TUF: The total number of features cleaned plus the class category.
B_R: The number of benign features that are closely related to one another that can be merged together.
M_R: These are malicious features that are closely related to one another that can be merged together
PB_R: The number of features gotten from merging related benign features.
PM_R: The number of features gotten from merging related malicious features.
M/(M_S∪M_R): The total number of malicious apps after single and related features are removed.
B/(B_S∪B_R): These are set of benign features after removal of single and related features.
M_t: Total cleaned malicious features minus single features plus merged related malicious features.
B_t: Total cleaned benign features minus single features plus merged related benign features.
M_t∩B_t: The total number of features that are found in both benign and malicious apps.
M/M_t∩B_t: Number of purely malicious cleaned features.
B/M_t∩ B_t: Number of purely benign cleaned features.
M_t∪ B_t: The total cumulative number of cleaned benign and malicious features.

After the extraction and cleaning of features - which included removal of single features, removal of duplicate features, and the merging of similar features, was completed, a reduced features vector space was obtained as shown in Table 6, which would fasten the processing time and improve the quality of learning results.

Table 6. Statistics of Cleaned Features

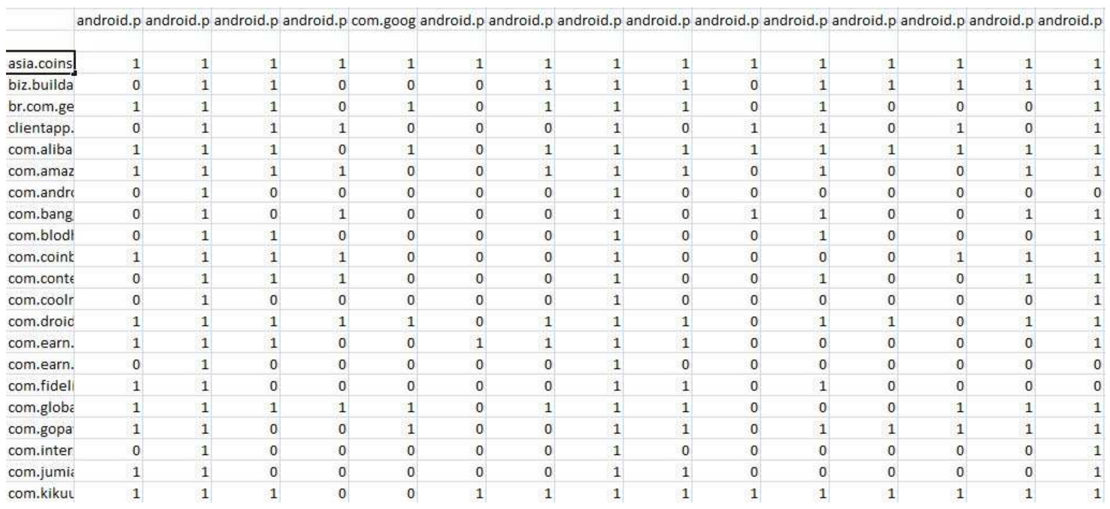
Features Type	Quantity
Single features removed from Malicious	49
Single features removed from Benign	133
Single features removed from both Malicious & Benign	0
Malicious ($\boldsymbol{M}$)	175
Benign ($\boldsymbol{B}$)	588
Malicious Alone ($\boldsymbol{M/M_t} \cap \boldsymbol{B_t}$)	38
Benign Alone ($\boldsymbol{B/M_t} \cap \boldsymbol{B_t}$)	58
M & B ($\boldsymbol{M_t} \cap \boldsymbol{B_t}$)	66
M OR B ($\boldsymbol{M_t} \cup \boldsymbol{B_t}$)	162
Class ($\boldsymbol{C}$)	1
Features usable for Vector analysis	163

The results revealed six sets of features combinations as shown in Table 6 as

a. Features found in malicious applications (M)
b. Features found in benign applications (B)
c. Features common to malicious and benign (M_t∩B_t)
d. Features unique or found only in malicious applications but not in benign (M/M_t∩B_t)
e. Features unique or found in only benign applications but not in malicious applications (B/M_t∩B_t)
f. Features found in malicious or benign applications ((M_t∩B_t) ∪ (M/M_t ∩ B_t) ∪ (B/M_t ∩ B_t)).

4.5 Feature Vectors Matrix Formulation

Cleaned features were transformed into N-dimensional binary vectors to enable suitability in the machine learning experiments. To form the N by M dimensional matrix of the binary vector space, the 162 cleansed features, plus one (1) Class attribute for output were used against the 1405 dataset to form a 1405 X 163 Matrix space. Applying Equation (3.14) on this cleansed data set, the binary feature vector matrix was generated as shown in Figure 8.



	android.p	android.p	android.p	android.p	com.goog	android.p	android.p	android.p	android.p	android.p	android.p	android.p	android.p	android.p	android.p
asia.coins	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
biz.builda	0	1	1	0	0	0	1	1	1	0	1	1	1	1	1
br.com.ge	1	1	1	0	1	0	1	1	1	0	1	0	0	0	1
clientapp.	0	1	1	1	0	0	0	1	0	1	1	0	1	0	1
com.aliba	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1
com.amaz	1	1	1	1	0	0	1	1	1	0	1	0	0	1	1
com.andr	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
com.bang	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1
com.blodl	0	1	1	0	0	0	0	1	0	0	1	0	0	0	1
com.coinb	1	1	1	1	0	0	0	1	0	0	0	0	1	1	1
com.conte	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1
com.coolr	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1
com.droid	1	1	1	1	1	0	1	1	1	0	1	1	0	1	1
com.earn.	1	1	1	0	0	1	1	1	1	0	0	0	0	0	1
com.earn.	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
com.fideli	1	1	0	0	0	0	0	1	1	0	1	0	0	0	0
com.globa	1	1	1	1	1	0	1	1	1	0	0	0	1	1	1
com.gopa	1	1	0	0	1	0	0	1	1	0	1	1	1	1	1
com.inter	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1
com.jumia	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1
com.kikuu	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1

Figure 7. Vectors Feature Matrix transformation

This matrix will then be translated into the format required for training the classifiers depending on the Machine Learning suites that would be used for the learning (classification) processes.

5. Conclusion

Feature engineering in Machine learning is one of the hardest aspects of Learning. Extracting the features was not the only step, but it also involved the rigorous process of constructing the features, through different cleaning methods, to obtain the most suitable set for the main process of training and testing of the learning classifiers. The result of reverse-engineering the Android applications was a dataset of streamlined (cleaned) features which were further transformed into a binary feature vector matrix. The binary feature vector, being the final state of features processing, presents a valuable set of data that is readily available to be used in the training phase of the machine classifiers. The feature gathering and processing phase is the most time-consuming stage of machine learning which, when carefully and patiently done correctly, makes the other phase of training and evaluation lots faster and correct. Hence, with this phase completed successfully, the next phase would be to use this dataset to fit the different learning classifiers.

References

[1] B. Mellars, A forensic examination of mobile phones. Digital Investigation, 1(4), 266–272, 2006, https://doi.org/10.1016/j.diin.2004.11.007

[2] United Nations. World Population Prospects 2019. In Department of Economic and Social Affairs. World Population Prospects 2019.

[3] K. Bankmycell. How Many Phones Are In The World? 2019.

[4] GSMA-Intelligence. Definitive data and analysis for the mobile industry, 2019

[5] H. Andrew, Android Forensice: Investigation, Analysis and Mobile Security for Google Android (1st ed.). Amsterdam: Syngress Publishing, 2011

[6] L. Tung, Android fragmentation: There are now 24,000 devices from 1,300 brands.2015.

[7] Gartner. Worldwide Smartphone Sales Grew 3.9 Percent in First Quarter of 2016. Retrieved on June 16, 2021 from https://www.gartner.com/newsroom/id/3323017

[8] Y. Feng, S. Anand, L. Dillig, and A. Aiken. Apposcopy : Semantics-Based Detection of Android Malware Through Static Analysis. Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), 16–22, 2014. https://doi.org/10.1145/2635868.2635869

[9] N. Aquilina. Cross-Platform Malware Contamination Cross-Platform Malware Contamination (Vol. 11). 2015, London.

[10] D. Arp, M. Spreitzenbarth, H., Malte, H., Gascon, and K. Rieck. Drebin, Effective and Explainable Detection of Android Malware in Your Pocket. Symposium on Network and Distributed System Security (NDSS), (February), 23–26, 2014, https://doi.org/10.14722/ndss.2014.23247

[11] S. Arzt, S., Rasthofer, C, Fritz, E., Bodden, A, Bartel, J, Klein, and P, Mcdaniel. FlowDroid : Precise Context, Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 259–269, 2014, https://doi.org/10.1145/2594291.2594299

[12] M, Lindorfer, S, Volanis, A, Sisto, M, Neugschwandtner, E, Athanasopoulos, F, Maggi, and S. Ioannidis. AndRadar: Fast discovery of Android applications in alternative markets. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 8550 LNCS, 51–71, 2014, https://doi.org/10.1007/978-3-319-08509-8_4

[13] A, Narayanan, L, Yang, L Chen, and L, Jinliang, Adaptive and scalable android malware detection through online learning. IJCNN 2016, 2484–2491, 2016, https://doi.org/10.1109/IJCNN.2016.7727508

[14] R, Raveendranath, V, Rajamani, A. J, Babu, and S. K. Datta, Android malware attacks and countermeasures: Current and future directions. 2014 International Conference on Control, Instrumentation, Communication and Computational Technologies, ICCICCT 2014, 137–143, 2014, https://doi.org/10.1109/ICCICCT.2014.6992944

[15] M, Spreitzenbarth, F. C, Freiling, F, Echtler, T, Schreck, and J, Hoffmann, Mobile-sandbox: Having a Deeper Look into Android Applications. Proceedings of the 28th Annual ACM Symposium on Applied Computing, 1808–1815, 2013, https://doi.org/10.1145/2480362.2480701

[16] R. J. Whelan, T. R., Leek, J. E, Hodosh, P. A., Hulin, and B. Dolan-gavitt, Repeatable Reverse Engineering with the Platform for Architecture-Neutral Dynamic Analysis. 22(1), 2016.

[17] E. Eilam, REVERSING Secret of Reverse Engineering (1st ed.), 2005, https://doi.org/10.1007/s13398-014-0173-7.2

[18] V. D. Aguilera, Android reverse engineering: understanding third-party applications. OWASP EU Tour. Bucharest: The OWASP Foundation, 2013.

[19] S. Y. Yerima, .S. Sezer, and G. McWilliams, Analysis of Bayesian Classification-based Approaches for Android Malware Detection. Information Security, IET, 8(July 2013), 25–36, 2014, https://doi.org/10.1049/iet-ifs.2013.0095

[20] K. Alfalqi, R. Alghamdi, and M. Waqdan, Android Platform Malware Analysis. 6(1), 140–146, 2015.

[21] B. Baskaran, and A. Ralescu, A Study of Android Malware Detection Techniques and Machine Learning. Proceedings of the 27th Modern Artificial Intelligence and Cognitive Science Conference 2016, Dayton, OH, USA, April 22-23, 2016., 15–23, 2016.

[22] S. Chen, M. Xue, L. Fan, S. Hao, L, Xu, H. Zhu, and B. Li, Automated poisoning attacks and defences in malware detection systems: An adversarial machine learning approach. Computers and Security, 73(Bo Li), 326–344, 2018, https://doi.org/10.1016/j.cose.2017.11.007

[23] Y. Dong, Android Malware Prediction by Permission Analysis and DataMining. The University of Michigan-Dearborn, 2017.

[24] O. S. Adebayo, Android-Based Malware Classification Using Apriori Algorithm with Particle Swarm Optimization. International Islamic University of Malaysia, 2017.

[25] L. Apvrille, and A. Apvrille, Identifying unknown android malware with feature extractions and classification techniques. Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015, 1, 182–189, 2015, https://doi.org/10.1109/Trustcom.2015.373

[26] S. Y. Yerima, S. Sezer, and I. Muttik, A New Android Malware Detection Approach Using Bayesian Classification. In Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on, 121–128, 2013, https://doi.org/10.1109/AINA.2013.88

[27] J. Sahs, and L. Khan, A Machine Learning Approach to Android Malware Detection. Intelligence and Security Informatics Conference, 141–147, 2012, https://doi.org/10.1109/EISIC.2012.34

[28] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, PIndroid: A novel Android malware detection system using ensemble learning methods. Computers & Security, 68, 36-46, 2017.

[29] A. H. Mostafa, M. M. Elfattah, and A. A. Youssif, Reduced Permissions Schema for Malware Detection in Android Smartphones. In Proc. Recent Advances in Computer Science, 19th Int. Conf. on Circuits, Systems, Communications and Computers (CSCC 2015) pp. 406-413), 2015.