

Android Malware Detection using Machine Learning Techniques on Low-Privileged Monitorable Features

A Major Project-II Progress Report

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

of

Master of Technology

in

Department of Computer Science and Engineering

by

Abhishek Shukla

(Roll No: 2K18/SWE/01)

under the guidance of

Dr. Divyashikha Sethia

Assistant Professor

Department of Computer Science and Engineering




DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY, DELHI
OCTOBER, 2020

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi - 110042

CERTIFICATE

I certify that the Project Dissertation titled **Android Malware Detection using Machine Learning Techniques on Low-Privileged Monitorable Features** which is submitted by Abhishek Shukla (Roll number: 2K18/SWE/01), Department of Computer Science Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a record of the project work carried out by the student under my supervision. To the best of my knowledge this work has not been submitted in part or full for any degree or diploma to this university or elsewhere.

Place: Delhi
Date: 30/10/2020



Dr. Divyashikha Sethia
Assistant Professor,
Department of Computer
Science and Engineering,
Delhi Technological
University

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi - 110042

DECLARATION

I, Abhishek Shukla, Roll No. 2K18/SWE/01, student of M.Tech (Software Engineering), hereby declare that the project report titled **Android Malware Detection using Machine Learning Techniques on Low-Privileged Monitorable Features** which is submitted by me to the Department of Computer Science and Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is original and not copied from any source without proper citation. This work has not been submitted anywhere for the award of Degree, Diploma, Fellowship or other similar title or recognition to the best of my knowledge.

Place: Delhi
Date: 30/10/2020

Abhishek
Shukla

Abhishek Shukla
Roll No. 2K18/SWE/01
Department of Computer
Science and Engineering,
Delhi Technological
University

ACKNOWLEDGEMENT

The success of a project requires help and contribution from numerous individuals and the organization. Writing the report of this project work gives me an opportunity to express my gratitude to everyone who has helped in shaping up the outcome of the project. I express my heartfelt gratitude to my project guide **Dr. Divyashikha Sethia**, Department of CSE for giving me the opportunity to do my project work under their guidance. Their constant support and encouragement has made me realize that it is the process of learning which weighs more than the end result. I thank my lab mates, without whose contribution this project would not have been possible. I also reveal my thanks to all my friends and my family for constant support.

ABSTRACT

The Android platform became one of the most vulnerable targets for cyberattacks in recent times due to a rapid surge in malware embedded apps. Researchers have investigated various machine learning techniques for Android malware detection but most of these techniques are inefficient against the novel malware. The various problems like code obfuscation, the requirement of device root privileges, simulated and small size datasets pose serious flaws to the existing solutions. This work evaluates several machine learning models for mitigating these issues using low-privileged monitorable features sampled in the SherLock dataset. The findings of this research conclude that the XGBoost classifier is the most accurate in detecting the malware compared to other classifiers with 93% overall values of precision, recall, and accuracy. In terms of FNR values, which signify the undetected malware, the XGBoost classifier also performs better than the other algorithms with values of 7.0%.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Motivation	3
1.3	Problem Statement	4
1.4	Proposed Solution	5
2	Literature Review	6
2.1	Android Platform Architecture	6
2.1.1	The Linux Kernel	6
2.1.2	Hardware Abstraction Layer	7
2.1.3	Android Runtime	7
2.1.4	Native C/C++ Libraries	7
2.1.5	Java API Framework	9
2.1.6	System Apps	9
2.2	Dalvik Vertual Machine	9
2.3	Android Application	10
2.3.1	Components of Application	10
2.4	Malware	11
2.5	Device Monitoring	11
2.6	Machine Learning Classifier	12
2.6.1	Naive Bayes classifier	12
2.6.2	K-nearest neighbour	13
2.6.3	Decision Tree	14
2.6.4	Random Forest	15
2.6.5	XGBoost	17
2.7	Data Analysis	17
2.7.1	Data Preprocessing	17
2.7.2	Data Analysis Phase	19
2.7.3	Feature Selection	19
2.8	Evaluation Metrics	20
3	Related Works	22
3.1	Static Analysis	22
3.2	Dynamic Analysis	24
3.3	Android malware detection using low-privileged monitorable features	26
3.4	Limitation of Methods	28

4	Methodology	29
4.1	Environmental configuration	29
4.2	Dataset	29
4.3	Feature Selection	32
4.4	Machine Learning Classifications	32
4.5	Evaluation Metrics	33
5	Results	35
6	Conclusion and Future Work	39

List of Figures

2.1	Android low level system architecture [1]	8
2.2	K-Nearest Neighbour [2]	13
2.3	Decision Tree [3]	14
2.4	Random Forest [4]	16
2.5	Data integration	18
2.6	Data Transformation	18
2.7	Data Reduction	18
4.1	Top 15 features obtained using Mutual Information Gain method	33
5.1	Measure of Accuracy, FNR, and FPR for all five classifiers	36
5.2	ROC curves for all five classifiers	37
5.3	Precision-Recall curves for all five classifiers	38

List of Tables

2.1	Confusion Matrix	20
3.1	Overview of some of the works for Android malware detection.	27
3.2	Limitations of Related Work	28
4.1	Files selected for use	31
5.1	Comparison of the performance of classification algorithms	35
5.2	Results For Benign Applications	37
5.3	Results For Malicious Applications	38

Chapter 1

Introduction

1.1 Overview

In recent times, smartphones turn out to be an indispensable part of human life as most of the day to day life computation is shifting towards smartphones. The smartphones are equipped with a variety of sensors to provide several applications to the end-users and generates a huge amount of sensitive and confidential data. Android OS, due to its open-source distribution, emerged as blazing popularity for smartphones in the last few years. This predominant operating system platform has established itself not only in the mobile world but also in the Internet of Things devices and turns out to be the most common operating system for smartphones with a market share of 86.1% by the end of 2019 [5].

The widespread adoption as well as the huge explosion of Android applications and contextually sensitive nature of smartphone devices has increased concerns over Android malware as it is very harmful to both the user data and device. The increase in users of Android smartphones has led to an increase in malicious applications that target mobile devices. Criminals are seeking to manipulate vulnerabilities on people's devices for their benefits. Due to the wide-scale acceptability of Android devices, a plethora of Google play store apps as well as third-party apps available for various applications to the end-users, but this popularity also brings challenges in the form of malware embedded apps. These malware-embedded apps aim to jeopardize the privacy and security of users by permitting unauthorized access to sensitive and confidential resources.

Malware can be harmful to the users in many ways like it can steal private information such as contact details, call details, message details. It may also harm the devices by exploiting the mobile resources which may lead to starvation of resources for benign apps. Similarly, spyware embedded internet browsing apps may keep an eye on internet activities, network traffic details, and sometimes even spot misleading advertisements and pop-ups within the web apps. Malware can also attack an android device by performing a privilege escalation attack [6] by which an unauthorized person can gain control of the phone via the backdoor. Furthermore, malware can perform other attacks like phishing and ransomware as well as it can affect the device by draining the battery and infecting the network interface card. So, there is a requirement of efficient security mechanisms for malware detection in Android devices.

The Google Play store as well as various other third-party app stores, including the app store provided by the smartphone manufacturers, provides an environment for the hosting and distribution of the Android apps. But uploading the new apps on these app

stores is quite easy and lenient as there is no strict standardization and security check mechanism for identifying the malicious apps. Google play store follows a very simple security mechanism known as Bouncer [7]. It is a third-party program that continuously scans the google play store repository for identifying malicious apps. However, this may reduce the number of uploaded malicious apps but still, it fails to detect most of the vulnerable apps uploaded on the play store. The other security mechanisms such as the Android permission system, integrated with the Android OS, control the permission to access the resources by giving the individual permissions to apps statically at the time of installation. But the issue with the Android permission system is that almost all the end-users blindly grant permission to apps during the installation. As a result, the intent of the installed apps may not be recognized by the end-user and it may jeopardize the privacy and security of the user by granting unauthorized access to sensitive and confidential resources.

In the earlier time, when malware is posing serious security issues, researchers have tried to apply the PC (personal computer) based security solutions to smartphones. Botha et al. [8] have applied the security mechanism used for PC to smartphones and claim that smartphones fail to perform well with these methods due to extensive resource utilization. Hence there is an increasing need for sophisticated, advanced, robust, and automated malware detection systems to detect malicious applications. Machine learning methods are very recent and well-established techniques for malware detection on the Android platform. Researchers have extensively classified the study of Android malware detection using machine learning techniques into two ways, static and dynamic analysis based on the features set acquired from the apps [9].

Static analysis refers to malware analysis before run-time execution by analysing the malware installation packages. Static investigation gathers set of features from applications by dismantling them using reverse engineering [10] without the run time execution. The features such as permissions set, APIs, application components, filtered intents, op-codes, and strings used by the apps are extracted by dismantling and parsing the installation.apk and AndroidManifest.xml files. [11]. The reasonable resource requirement and less time consuming make the static analysis best suited for devices having low memory and less processing capacity. The downside of the static analysis is that it can't avoid the code obfuscation [12] [13], as well as it can't detect the injection of non-Java code, network activity, and the modification of objects at run-time as they are only visible during execution [14]. A major drawback of static analysis is that most of the work for malware detection is based on analysis in virtual environments, like analysing on PC, rather than analysing on real mobile devices. Also there is an increasing surge in malware that uses strategies to prevent detection in virtual environments, making analytical approaches in virtual environments less effective than those approaches focused on real devices.

Whereas dynamic analysis refers to malware analysis during run-time, i.e. during execution of the application, and the malicious application is identified using the features based on system calls, memory usage, network connections, power intake, as well as individual's interaction with the user interface (UI). An open-source tool, called DroidBox [15], can be used to extract these features from the apps on emulated environments. As compared to static analysis, dynamic analysis is very effective against the code obfuscation but still, it is suffered from the problem of ant-virtualization [16], [17], code coverage constraints [18], [14], and device root privileges. Device root privileges require some adjustments to the kernel of codes which may lead the security architectures of operating systems to lose their efficacy [19].

Some other techniques like hybrid analysis combine both the static and dynamic analysis to mitigate each other's limitations but still, it is not effective in terms of performance overhead, decrease transparency, increase chances of code bugs, and maybe less portability [14]. So because of these unrealistic demerits, most research works does not provide a realistic assessment and performance to their detection methods.

So to overcome the limitations of Static and Dynamic analysis, researchers have proposed the machine learning approach to malware detection using device-based low-privileged monitorable features. Shabtai et al. [20] have utilized Logistic Regression, Naive Bayes, and Decision Tree with various low-privileged monitorable features but the results of this method are not effective to a large extent due to small training dataset. Memon et al. [21] have also worked with low-privileged monitorable features and utilized various algorithms like Decision Tree, Random Forest, Support Vector Machines, Gradient Boosting etc. but their results show a high false positive rate (FPR) and low accuracy. Wassermann et al. [22] have also used low privileged monitorable features for malware detection and obtains a good classification accuracy using the Decision Tree, but it may suffer from the overfitting of data in the training phase.

To mitigate the limitations of the above approach, in this research we applied a machine learning approach to malware detection using the SherLock dataset [23] which is a labeled dataset consist of device-based low-privileged monitorable features. We employed the Mutual Information gain method for feature selection and utilized Random Forest [24] and XGBoost classifier [25] for avoiding the overfitting of data in the training phase. Our experiment shows very effective results compared to the above methods in terms of low FNR and FPR values with high accuracy and recall rate. Our work compares the performance of various machine learning techniques on the features sampled from real-time sensors. The Ben-Gurion University provides a significant smartphone data known as Sherlock dataset [23], which contains the malware related data and global device data from 47 users since 2016. Malware related data provides logs of actions taken by various malicious applications and record the hidden malicious activities performed by them. The global device data are the system metric logs, like usage of the CPU, usage of memory and usage of batteries etc. Monitoring the device metrics does not need any changes like rooting the android devices, i.e. changing the Operating system to allow the monitoring at kernel level. In this work we develop a malware detection methods for unrooted android devices, as more than 95 percent of android devices are unrooted, and train the following classifiers for malware detection purpose: i) Naive Bayes, ii) K-nearest neighbor, iii) Decision Tree, iv) Random Forest and v) XGBoost. The classifiers are trained to determine whether an application is performing benign or malicious activities on the phones, provided the system metrics of a phone at a given moment.

1.2 Motivation

The popularity of smartphones and other types of mobile devices such as tablets has risen considerably in the past several years. The large quantity and variety of mobile applications and the increased functionality of the mobile devices themselves accompany this fact. Android OS emerged as a blazing popularity since the last few years for smartphones and tablets with an estimated market share of 86.1% [5].

As a side effect of this blazing popularity of android operating system, centralized marketplaces like Google Play have evolved exponentially. Such marketplaces allow de-

velopers to conveniently upload their own apps and users can download those apps directly to their mobile devices from app store. In addition to official platform vendors markets like Google manufacturers like Samsung and HTC, a large number of unofficial third party marketplaces also available. Most of these marketplace have thousands of apps, and million of these apps are downloaded monthly. This rapid growth rate of android application comes at a cost. Attackers have realized that to perform cyber attacks on smartphones, rogue and malicious apps can be used and hence smartphone malware has become popular in the recent past. The open design of Android allows users to install applications from the third party vendor that do not necessarily available on Google Play Store. With over 1 million apps available for download from the official Google store, and probably another million scattered across third-party app stores, we can estimate that more than 20,000 new apps are launched each month. This requires that malware researchers and store managers have access to a scalable solution to quickly analyze new apps and identify malicious applications and isolate them. A large amount of research has been going on Android malware detection, but none of them provide a viable solution to acquire a thorough understanding of unknown applications. Most of these work is based on the System call analysis and tracking of only specific API invocations. These specific malware detection methods need rooting of devices and need change in kernel level code. So there is requirement of a sophisticated malware detection system which works with low privileged monitorable features on unrooted devices.

1.3 Problem Statement

The Android system popularity has resulted in a significant increase in the spread of Android malware, as discussed in section 1.1. This shows Android malware development over 2014. These malware is mainly distributed and managed by third-party markets, but even Google's Android Market can not guarantee that all of its listed applications are free of threats. Examples of Android malware include phishing applications, banking Trojan, spyware, bots, root exploits, SMS scams, and fake installers. There are Trojan apps that download their malicious code after installation so that Google technology can not easily detect those apps when they are published on the Google Android Market.

Most malware detection strategies are based on conventional content signature driven techniques, using a list of signature malware definition and comparing each application to the known signature database for malware. The downside of this method is that users are only protected from malware detected by the most recent updated signatures, but are not protected from the new malware. So the efficiency of this method is based on the exposure of database to new malware signatures. The earlier studies of malicious patterns concluded that signature-based approaches never keep up with the speed at which malware is created and evolved. The other security mechanisms such as the Android permission system, integrated with the Android OS, control the permission to access the resources by giving the individual permissions to apps statically at the time of installation. But the issue with the Android permission system is that almost all the end-users blindly grant permission to apps during the installation. As a result, the intent of the installed apps may not be recognized by the end-user and it may jeopardize the privacy and security of the user by granting unauthorized access to sensitive and confidential resources.

1.4 Proposed Solution

One of the solution to this problem is to utilize the power of Machine Learning techniques on phone sensor data set. In this work, our aim is to identify a machine learning solution to Android malware detection based on the low privileged sensor data. The proposed solution aims to provide a practical solution to android malware by avoiding the device root privileges requirement and offers the resistance against code obfuscation, anti virtualization and, code coverage constraints. In this work, we utilize the Mutual Information gain method for feature selection and employed various classifiers like Naive Bayes, K-nearest neighbor, Decision Tree, Random Forest and XGBoost. We carried out our work in following steps maintained as follows.

- Pre-process the existing sensor data (In this work Sherlock Data set [23] is used).
- Implement Classification model.
- Classify new set of Data and evaluates the model.

Chapter 2

Literature Review

It's important to understand the architecture of Android operating system and working of Android applications before discussing the details of this research work. Along with this, there is a detail discussion on the machine learning techniques used, data pre-processing techniques, and evaluation metrics utilized to inspect the classification model. In this chapter a brief introduction to the Android architecture, malware family and different types of malware attacks is given. In section 2.1 a high-level overview of the Android system architecture is given. It briefly discusses the architecture and different component layers of android system. Section 2.2 deals with the virtual machine that is responsible for operating the Android applications. Section 2.3 discuss key modules used in Android apps. This section discusses activities, services, recipients and attempts, the building blocks of Android applications. Finally, section 2.4 discuss briefly about different types of Android malware and malware acts on the the Android platform. Section 2.5 provides the detail about the features obtained from Android devices and Android applications. Further, section 2.6 discusses the different machine learning techniques used in this work. Section 2.7 and 2.8 discusses about the data analysis and evaluation metrics respectively.

2.1 Android Platform Architecture

Android is an open source software platform based on Linux, built for a wide range of devices and form factors. The key components of Android platform is shown in below diagram. In this figure, violets items are modules written in native machine code (C/C++), while green items are modules interpreted and executed by the Dalvik Virtual Machine(DVM). The bottom red layer contains the components of the Linux kernel and executes in kernel space. Using a bottom-up approach, we address briefly the different abstraction layers in the following subsections.

2.1.1 The Linux Kernel

The Android platform is based on the Linux kernel. The Android Run time (ART),for example, depends on the Linux kernel for basic functionalities including multi threading and low-level memory management. Linux kernel helps Android to take advantage of core security features and encourages handset makers to build hardware drivers for well-known kernel. Android uses a sophisticated version with some special additions to the Linux Kernel. These includes wake-locks, memory protection scheme that is more proactive in

saving memory, the Binder IPC driver, and other functionality that are essential for a mobile embedded platform such as Android.

2.1.2 Hardware Abstraction Layer

The Hardware abstraction layer primarily offers standard interfaces that present the functionality of device hardware to high level Java API platform. The HAL consists of several library modules, each of which implements an interface for a particular type of hardware item, such as the camera or the bluetooth module. The Android system loads the library package for that hardware unit when a Framework API makes a call for device hardware to access.

2.1.3 Android Runtime

The Android Runtime is a middleware component consists of the Dalvik Virtual Machine(DVM or DalvikVM) and a collection of core libraries. Dalvik VM is accountable for executing applications developed using the Java programming language. For devices that run Android version 5.0 (API level 21) or above, each application runs its own Android Runtime (ART) instance with its own process. ART is compiled to run multiple virtual machines on low-memory phones by executing DEX files, a Bytecode format specifically designed for Android that is optimized for minimal memory requirement. Android have two different types of main libraries.

1. Dalvik Virtual Machine specific libraries
2. Java programming language interoperability libraries

The first set allows specific VM information to be processed or modified, and is used primarily when byte code needs to be loaded into memory. The second set provides Java programmers with the common environment, which comes from Apaches modules. It implements most of the Java common packages like `java.lang` and `java.util`. Some of the key features of Android Run Time include the following:

- Ahead-of-time (AOT) and just-in-time (JIT) compilation
- Optimized garbage collection (GC)
- Better debugging support with a dedicated sampling profiler, detailed diagnostic exceptions and crash reporting

2.1.4 Native C/C++ Libraries

Many core components and services of the Android system, such as ART and HAL, are developed from native code, which require native libraries written in C and C++ language. Mostly these are external libraries with only slight improvements such as OpenSSL, WebKit and bzip2. The Android platform provides Java framework APIs to expose applications to some of those native libraries' functionality. You can use the Android NDK to use any of these native application libraries directly from your native code if you are creating an app that requires C or C++ code.

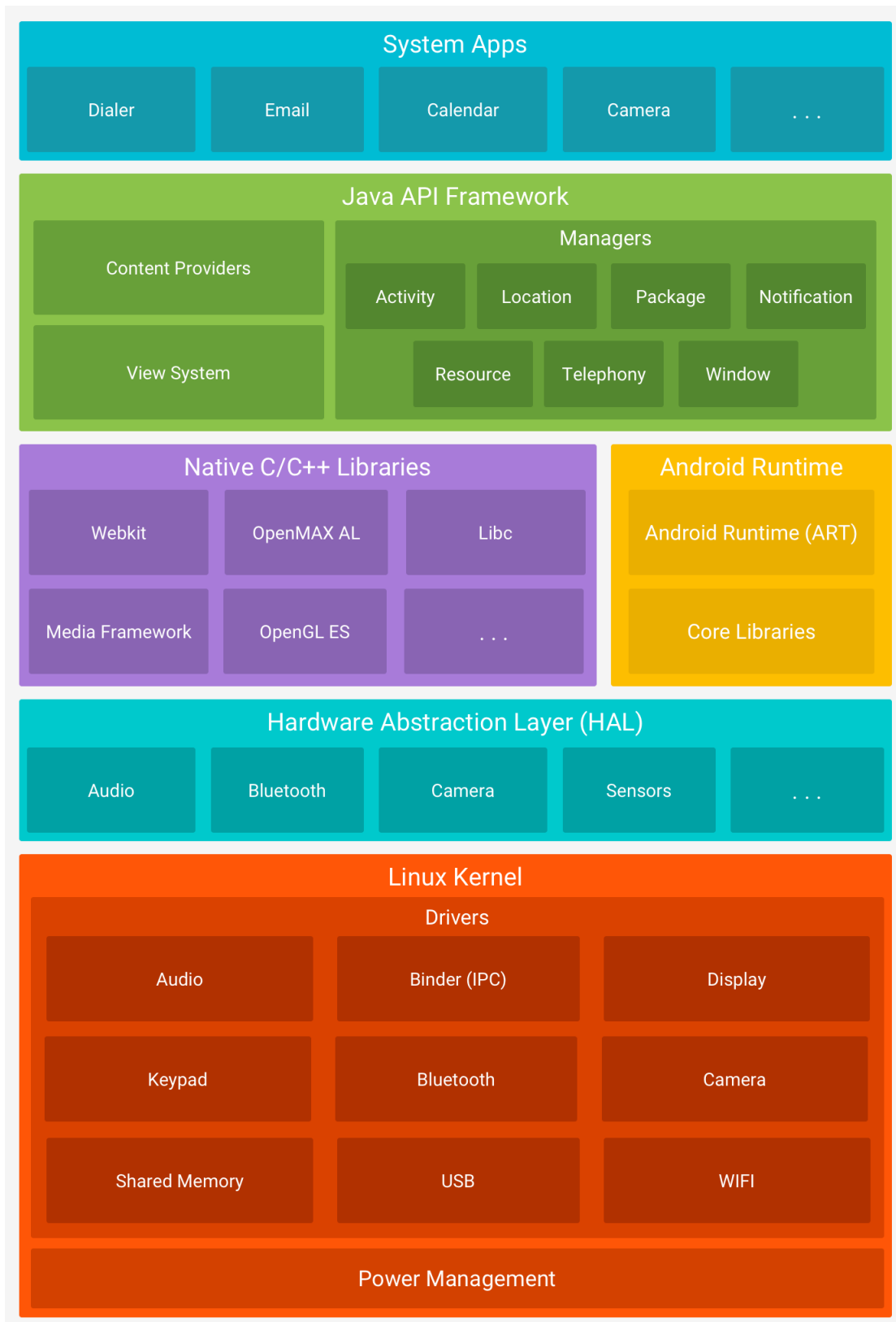


Figure 2.1: Android low level system architecture [1]

2.1.5 Java API Framework

The entire Android OS feature set is available to you via Java-written APIs. These APIs comprise the building blocks that you need to develop Android apps by simplifying the re-usability of core, modular framework components, and services. Some modules handle simple phone functions such as accepting phone calls or text messages or controlling power consumption. A few things need a little more consideration.

1. Activity Manager: If the device is running out of memory it is liable for terminating background processes. This also has the ability to detect unresponsive applications when it fails to respond within 5 seconds to an input event (such as a key press or a screen touch). This then triggers the Not Responding Request message.

2. Content Provider: Content Providers are one of Android's main building blocks. They are used for data exchange between different applications and components. For example, since contact list data required to be accessed by various application and thus must be stored in a content provider.

3. Telephony Manager: The Telephony Manager conveys information about the telephony resources, such as the unique user identification (IMEI) of the handset or the actual cell location. This is also accountable for managing phone calls.

4. Location Manager: The Location Manager provides access to system location services that enable applications to regularly update the geographical location of the device using the GPS sensor of the devices.

2.1.6 System Apps

Applications or integrated on top of the Application Framework and are responsible for the interfacing between end-users and the device. Android comes with a collection of core apps like email application, Text message, calendars, browser for online surfing, contact list and many more. Apps installed with the platform have no special privilege among the other installed apps which the user chooses. And so a third-party app can be the default web browser, SMS messenger or even the default keyboard for the user with some exceptions will be there such as the system Settings application etc.

2.2 Dalvik Vertual Machine

Usually, Java source code is compiled and transmitted as Java byte code and is interpreted and executed by a Virtual Machine (VM) at run time. But Google has decided to use a different byte code and Virtual Machine format called Dalvik for Android platform. Java byte code is converted to Dalvik byte code during the compilation process of Android applications which can later be executed by the specially designed Dalvik Virtual Machine. The bytecode that the DVM interprets is so-called DEX bytecode (Dalvik EXecutable code). DEXcode is obtained using the dx tool on converting the Java bytecode.

2.3 Android Application

Mobile apps are delivered as APK files. APK files are signed ZIP files that comprises the byte code of the application along with all its data, tools, third-party libraries and a manifest file describing the detail functionality of the application. Permissions for files in an application are then registered so that only the application itself can able to access them. In addition, every program is given its own Virtual Machine when it is started which ensures code is segregated from other applications

2.3.1 Components of Application

We now discuss a number of core components of the framework which is used to create Android applications.

Activities

An activity stands for a single screen with a specific user interface. Apps are likely to have more than one activity numbers each with a different purpose and activity interlinked with each other. Each activity is independent of the others and can be started by other applications too if permitted by the app. For example, a music player could have one activity showing a list of available albums and another activity showing the song being played with buttons, to pause, enable shuffling, or quickly forwarding.

Services

Services are elements running in the background for long running operations and do not provides a user interface. For example, the music streaming app will have a music service that is responsible for playing background music whilst the user is interacting with other application. Other components of the app like an activity or a broadcast receiver can initiate services.

Content providers

Content providers are used for data sharing across multiple applications. They manage shareable set of data among different applications. For example, contact data is kept in a content provider so that it can be queried by other applications when necessary.

Broadcast receiver

A broadcast receiver listen and respond to specific system broadcast announcements. Broadcast receivers have no graphical user interface and are often used to act as entry point to other components. For example, they may invoke a background services to do some function based on occurrence of a specific event.

2.4 Malware

The term malware represents a contraction of malicious software. In simple terms malware is any piece of software that was written to damage devices, leak information and generally cause mess. Viruses, trojans, spyware, and ransomware are among the most common malware styles. In this section, we take a closer look at the characteristics of mobile malware and discuss how it is distributed.

Types of Malware

- **Virus:** Viruses as like their biological namesakes, attach themselves to clean files and infect other clean files. They can be spread in uncontrolled way and can damage the core functionality of the system and can make changes, delete or corrupt files. Commonly they come as an executable file.
- **Trojan:** This kind of malware masks itself as legitimate software, or is concealed in legitimate software that has been compromised. It begins to act discreetly and create security holes to let other malware in, in your security.
- **Spyware:** spyware is malware intended for spying on you. It resides in the background and keep eyes on about what you are doing online, including your passwords, credit card numbers, browsing activities, and more.
- **Worms:** Worms attack whole computer networks using network interfaces, either locally or over the internet. It uses each computer that has been infected consecutively to infect others.
- **Ransomware:** Typically this kind of malware locks down your mobile device and files, and jeopardizes to erase everything unless you pay a ransom.
- **Adware:** Although not always malicious in nature, intensive advertising software can bolster your security just to serve you ads — which can provide an easy way in for other malware.
- **Botnet:** Botnets are used to infect the networks of machine, which are made to work together under an attacker's control.

2.5 Device Monitoring

Device monitoring is described as the analysis and monitoring of the features by the detection method to identify the behaviour of the system is malicious or not. The device can be monitored in different ways depending on the types of the features used by mobile malware detection method. The features can be act as the input to the malware detection method. Features could be categorized into three categories:

- Hardware Features
- Software Features

- Firmware Features

Hardware features are features that can be tracked and are device-specific, such as battery storage, Processor and memory utilization. Software features are functions which can be tracked during program execution or by analysis of software package such as permissions and network traffic. Firmware features are features from programs that use read-only memory. Most firmware features require the Android OS to have rooting privileges.

2.6 Machine Learning Classifier

Machine learning is the quintessential skill of this digital age. As we dissect the process how a machine learns to classify and the inputs or the raw materials needed for learning the specifics of the desired task, features or attributes forms the basis of what we feed in the learning algorithm. In the task of android malware detection machine learning plays an important role. There might be many techniques available to do such task. We will discuss various methods that can be used to achieve such task.

2.6.1 Naive Bayes classifier

Naïve Bayes (NB) is classification techniques based on statistic, which uses the Bayes theorem to determine the probability of a given sample point that belongs to a certain class. The Bayes theorem calculates the likelihood of a hypothesis H being true given that some evidence e, according to the below given formula.

$$P(c/x) = \frac{P(c) * P(x/c)}{P(x)}$$

$$P(c/x) = P(x1/c) * P(x2/c) *P(xn/c)$$

where,

P(c) denotes the prior probability of class (c)

P(x) denotes the prior probability of predictor

P(x/c) denotes the posterior probability of predictor (x) given class (c)

P(c/x) denotes the posterior probability of class (c) given predictor (x) attributes

The classification model is called naive because it implies conditional independence, trying to make it less computationally expensive to compute the above formula, particularly for data sets with several features. While Naive Bayes implies conditional independence, still it performs well in those mathematical perspective where conditional independence is violated. Naive Bayes classifier is advantageous as it fast, simplistic and mature algorithm, insensitive to irrelevant feature data and a disadvantage is that it requires the assumption that features are independent.

2.6.2 K-nearest neighbour

K-nearest neighbour is a distance based classification technique in which new data points are classified by looking at their k- number of neighbours. The new data point is being associated to the class which has majority among the entire k neighbours. Thus decision boundary of majority class is improved by some margin. This process continues till all the points are being classified. KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970's as a non-parametric technique.

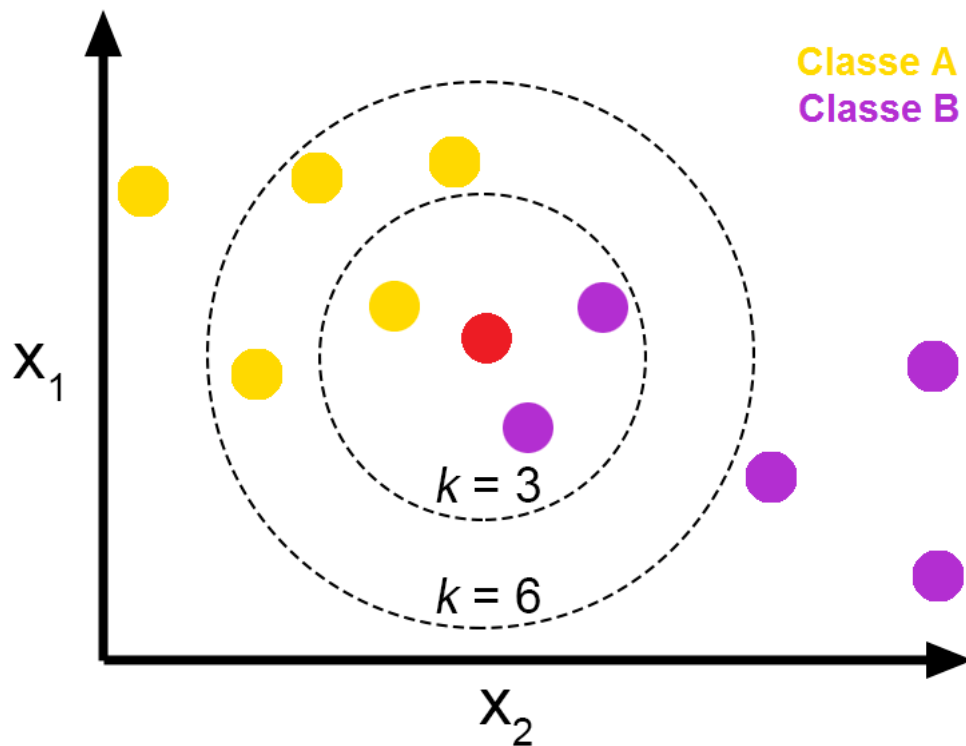


Figure 2.2: K-Nearest Neighbour [2]

Algorithm

A class is classified by a majority support of its neighbours. The class being assigned to the most common amongst its K nearest neighbours measured by a distance function. There are three distance metrics used for calculating the neighbouring distance.

- Euclidean
- Manhattan
- Minkowski

Euclidian Distance :The Euclidean distance of two points p and q in n-dimensional system is given by

$$d(p, q) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

Manhattan Distance: The manhattan distance between two points is defined as:

$$d(M, P) \equiv |M_x - P_x| + |M_y - P_y|$$

Minkowski Distance: The Minkowski distance between two variables X and Y is defined as

$$d(X, Y) = \left(\sum_{i=1}^n |X_i - Y_i|^p \right)^{1/p}$$

such as, when $p = 1$ it represents Manhattan distance and when $p = 2$ it represents Euclidean distance. As p is a real value it may attain any value but normally it sets to a value from 1 to 2. The above theorem does not determines a valid distance metric for values of p less than 1, as it violate the triangle inequality.

2.6.3 Decision Tree

A decision tree is a non-parametric supervised model constructed using the recursive instance space partition method. It is a tree-like graph having internal nodes representing an attribute with rules, edges are the responses to the rules, and the leaves represent the class label. The model is constructed by learning the simple decision rules derived from the data features and can be used for both classification and regression purposes. Since these decision rules are mostly in the form of if-then-else statements so, the deeper the tree, the more complicated the rules would be and thus provides the best-fit model.

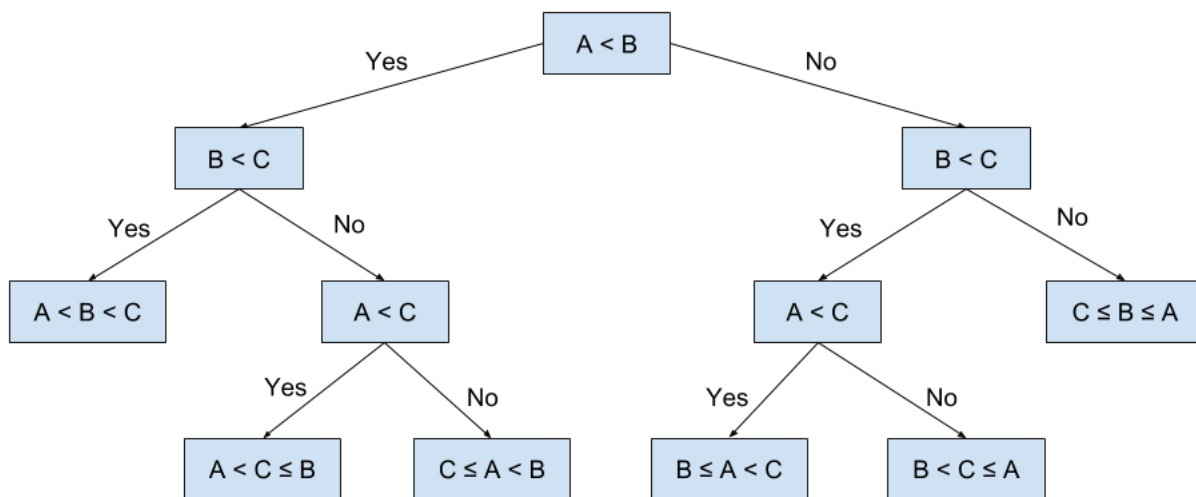


Figure 2.3: Decision Tree [3]

Algorithm

The baseline algorithm used in the decision trees is known as the ID3 algorithm that generates a decision tree by using the greedy, top-down strategy. Briefly, the algorithm steps are:-

- Pick the best attribute "A" from the initial dataset.
- Allocate A as the decision attribute for the NODE.
- Produces a new descendant of the NODE for each value of "A".
- Arrange the training instances to the descendant leaf node.
- If the instances are correctly defined, then STOP else iterate over the new leaf nodes.

The most important thing in the decision tree algorithm is choosing the best attribute in every iteration. For the ID3 algorithm, the best attribute is that which has the most information gain, which is a metric that reflects how well an attribute separates the data into classification classes and can be expressed as follows in terms of entropy.

$$InformationGain = 1 - Entropy \quad (1)$$

Where, the entropy can be calculated using:

$$Entropy = - \sum_{x \in \mathcal{X}} p_i \log_2 p_i \quad (2)$$

where, p_i = Probability of $class_i$.

Entropy is utilised to measure the purity of node such as lower entropy value signifies higher node purity. The Entropy of the Homogeneous node is zero hence its Information Gain is maximum equal to 1 as evident from the above equations.

2.6.4 Random Forest

Random forest is a tree-based algorithm composed of several decision trees and their performance is combined to boost the generalisation ability of the model. The ways of combining the trees are known as ensembling which is defined as combining the weak learners (individual trees) to build a strong learner. Random Forest can be utilized to solve the problem of regression and classification. In regression problems the dependent variable is constant whereas in classification problems the dependent variable is categorical. Random forest employ bootstrapping with decision tree models in attempts to create several decision tree models with resampled dataset and initial variables to reduce the variance and improves the generalization capability of model. This process is repeated several times before making final predictions on each observation and final prediction is the function of each prediction such that it can simply be the mean of each outcomes.

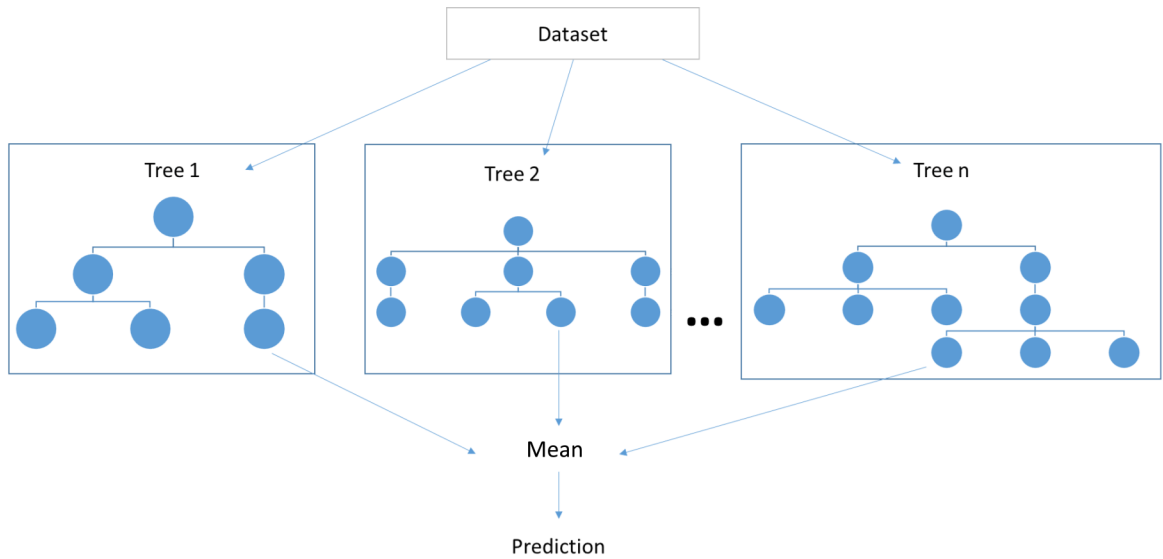


Figure 2.4: Random Forest [4]

Algorithm

The basic steps followed in Random Forest algorithm is briefly explained below.

- The first step is to generate random samples using the Bagging (Bootstrap Aggregating) algorithm. Suppose we have a dataset D_1 having n rows and p columns, a new dataset (D_2) is generated by randomly sampling n cases with replacement from the original data such that $1/3$ of the rows from D_1 are left out, known as Out of Bag (OOB) sample.
- In the second step, the models are trained using D_2 dataset and utilize the OOB sample for determining unbiased estimate of the error.
- Now for constructing the tree, at each node P columns out of p columns, such that $P \ll p$, are randomly selected. The default choice of P is $p/3$ for regression and \sqrt{p} for classification tree.
- Unlike decision tree, no pruning occurs in a random forest and every tree is generated completely. Pruning is a method for preventing overfitting in decision trees by selecting a subtree which results in the lowest rate of error testing. Cross-validation can be used to evaluate the test error rate for a subtree.
- Bootstrapping is employed to grow several decision tree models and final prediction is obtained by averaging or voting the output of every decision tree model.

2.6.5 XGBoost

XGBoost is nowadays the most popular machine learning algorithms to provide better solutions than other well-known ML algorithms for the problem of regression or classification regardless of the form of data. It is an ensemble learning method that provides a comprehensive approach for integrating the predictive power of multiple learners and results in a single model that generates aggregated output from several models. The models that constitute the ensemble method could be either from the same learning algorithms or from different learning algorithms and these models are known as base learners. Bagging and boosting are two common approaches to ensemble learners and can be used with several mathematical models, but the most commonly utilized with decision trees. For certain algorithms like decision trees including Classification and Regression Trees (CART) that have a high variance, bagging (bootstrap aggregation) is a common method to reduce the variance. The decision trees are very prone to the training data used as the resulting decision tree will be quite different if the training data is resampled with different sizes and different initial variables and so the predictions can be quite different in turn also. Bagging is the extension of the Bootstrap technique to, a high-variance machine learning algorithms, usually a decision trees. When the bagging is applied with the decision trees, the concern of overfitting the training data with the individual trees is not considered and hence the individual decision trees are allowed to grow deeper and the pruning of trees is avoided. The boosting is a process in which trees are created sequentially so that each of the corresponding trees attempts to reduce the errors associated with the previous trees. Each tree improves to its counterparts and the residual errors are updated. Therefore the tree that evolves next in the series will improve from a modified version of the residual. The base learners involve in boosting are weak learners with high bias and better predictive power. Each of these weak learners adds some crucial predictive knowledge and allows the boosting technique to effectively combine these weak learners to create a strong learner which takes down both bias and variance.

2.7 Data Analysis

Data analysis is the application of statistical process for examining, filtering, transforming and modelling of data in order to uncover valuable knowledge, to draw insights and facilitate decision-making. Data collection and analysis have several nuances and approaches which covers a vast application in various areas of business, science and social science.

2.7.1 Data Preprocessing

This is the very basic step in modeling any machine learning model as in this phase the data is converted into a format that can be processed easily and effectively. That is done by cleaning dirty data for improving the quality of data. Various types of preprocessing techniques are as follows:-

- **Data cleaning**

Redundancy is removed, missing values are handled and inconsistency is taken care of.

- **Data Integration**

Combining related data together to form a single dataset is called Data Integration. Usually different type of datas are stored in different dataware house. So at the time of processing we need to integrate the data.

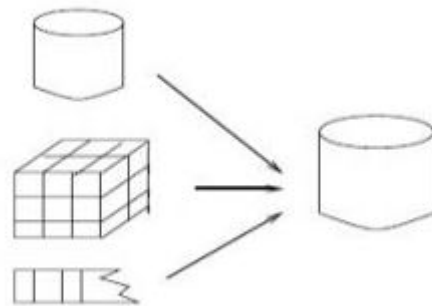


Figure 2.5: Data integration

- **Data Transformation**

Transforming data into usable form or into another form for normalizing it with respect to other attributes.



Figure 2.6: Data Transformation

- **Data Reduction**

Removing unnecessary data from the dataset or by reducing the feature set and reducing the dimensionality.

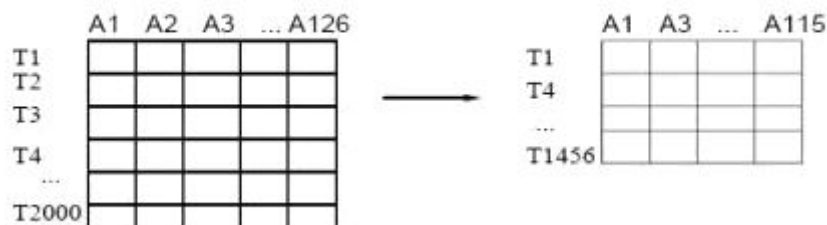


Figure 2.7: Data Reduction

2.7.2 Data Analysis Phase

This phase refers to identification of techniques to be used, irregularities in the data are identified and feature set are selected which are most crucial in decision making

- **Selection of methods**

In this step, out of so many techniques available, a most suitable technique is identified and data irregularities are identified on the basis of which classification is to be made.

- **Identification of decision making features**

In this step out of many features only those crucial features which are useful for designing the model are selected.

- **Declaration of rules and conditions for selected methodology**

In this step rules are recognized and declared on the basis of which decision making is to be done.

2.7.3 Feature Selection

Information Gain method

Feature selection is performed to get rid of the insignificant and repetitive information present in the dataset. It reduces the dimension of the dataset to avoid overfitting of data hence improves the accuracy and reduces the complexity and running time of the model. The mutual information between the discrete random feature variable $X = \{x_1, x_2, \dots, x_k\}$ and discrete random target variable $Y = \{y_1, y_2, \dots, y_d\}$ is expressed as:

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \cdot \log \frac{p(x, y)}{p(x) \cdot p(y)} \quad (1)$$

The mutual information $I(X; Y)$ can quantify any kind of relation between variables and if its value is large, it means two variables are closely else they are not closely related. The value of mutual information can not be negative, if its value is zero it means two variables are independent of each other.

The mutual information $I(X; Y)$ is also related to entropy (H) and can be equivalently expressed as:

$$I(X; Y) = H(X) - H(X|Y) \quad (2)$$

where $H(X)$ is the entropy of feature X and $H(X|Y)$ is the conditional entropy of X when Y is introduced. The entropy of feature variable X can be calculated using:

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log p(x) \quad (3)$$

where $p(x)$ is the mass probability function for the random variable X . Similarly, the conditional entropy of X against Y can be calculated as:

$$H(X|Y) = - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x|y) \log p(x|y) \quad (4)$$

where $p(x|y)$ is the conditional probability of x against y . So from the above equations, we conclude that reduction in entropy of feature X will enhance the significance of feature.

Wrapper and Embedded method

Feature selection is performed to get rid of the insignificant and repetitive information present in dataset. It reduces the dimension of dataset to avoid over-fitting of data hence improves the accuracy and reduces the complexity and running time of the model. We use the Recursive Feature Elimination (RFE) method and regularization algorithm like Select From Model to find out the importance of every features and further these features are used to train the classification model. In Recursive Feature Elimination method greedy search approach is followed to find out the set of features which have effective performance with the specified ML algorithms. Whereas in Select From model there is in built feature selection method associated with ML algorithms.

2.8 Evaluation Metrics

Evaluation metrics are used to evaluate the classification modeladjustbox. There are various evaluation metrics used to inspect the classification model like confusion matrix, cross-validation, AUC - ROC (receiver operating characteristic) curve, and Precision - Recall curve. These metrics help to evaluate the effectiveness of model depending on the requirement of problems.

Confusion Matrix

This matrix gives information about actual and predicted class and used for the performance evaluation of the supervised machine learning models. The confusion matrix is shown in table 2.1 which has four cells names as TP (True Positive), TN (True Negative), FP (False Positive), and FN (False Negative) where true cases determines number of correct predictions and false cases are wrong predictions. There are some standard terms derived from the confusion matrix which acts as performance evaluation metrics for any model which are stated as follows.

		Predicted Class	
		Malicious	Benign
Actual Class	Malicious	True Positive (TP)	False Negative (FN)
	Benign	False Positive (FP)	True Negative (TN)

Table 2.1: Confusion Matrix

- The recall or true positive rate (TPR) is defined as

$$Recall = \frac{TP}{TP+FN}$$

- The false positive rate (FPR) is defined as

$$FPR = \frac{FP}{TN+FP}$$

- The false negative rate (FNR) is defined as

$$FNR = \frac{FN}{TP+FN}$$

- The accuracy is defined as

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

- The precision is defined as

$$Precision = \frac{TP}{TP+FP}$$

- The F1-score is defined as

$$F1 - score = 2 * \frac{Recall * Precision}{Recall + Precision}$$

Receiver Operating Characteristic curve

To compare the detection performance of classifiers, the receiver operating characteristic (ROC) curve is utilized which is a plot between TPR and FPR for different thresholds of the detection methods. The AUC (area under the curve) value of ROC curves gives the sense of how effectively the data can be separated by the classification model. It represents the total area under the ROC curve and its value closer to 1 implies better classification. So, the value of AUC closer to 1 indicates better classification power of that particular model.

Precision-Recall curve

Precision-Recall curve is a plot between precision and recall for different thresholds of the detection methods. This plot is used to represent the tradeoff between precision and recall such as high AUC (area under the curve) value of Precision-Recall curve shows high precision (low false positive rate) and high recall (low false negative rate).

Chapter 3

Related Works

In this chapter, we discussed relevant research work done on android malware detection using machine learning techniques. The literature contains various methods which adapt different strategies to detect malware applications. Basically most of the android malware detection work can be grouped in two categories, static analysis and dynamic analysis. Both approaches have their advantages and disadvantages as Static analysis is unrealistic and vulnerable to obfuscation where as dynamic analysis is generally faster as compared to static analysis and less resource intensive but requires change in kernel of the operating system. Some other methods combine static and dynamic analysis, known as hybrid analysis, to improve detection accuracy. Along with this, there are some new approaches to malware detection using device-based low monitorable features. Below, we give a quick review of existing approaches that belong to these categories. In this segment, we also address the work done using the identification of behavioral and signature based malware.

3.1 Static Analysis

One of the significant techniques for analysing the malware is through static analysis that performs detection of malware applications before installing or before running on the devices. Static analysis gathers set of features from applications by unpacking and dismantling them without installing and executing on device. The static analysis based features are extracted from the installation.apk files and parsing the classes.dex and AndroidManifest.xml files. This provides the permission set required and API call sequence used by the application. It is useful with respect to less computational expense, low asset usage, light-weights and less time expending in nature. The static examination can be utilized on gadgets which are low in memory and less power consuming in nature. Recent work on Android malware detection which uses machine learning techniques with static features are discussed as follows.

Chin et al. [26] proposed a method called Comdroid which is used for detecting the malicious application using communication based vulnerabilities in Android. In addition to an open API, the Android operating system also features a rich inter application message passing system for transferring messages between applications. This promotes cooperation between applications and reduces the burden on developers by encouraging reuse of the components. Unfortunately, the passing of message is also the subject of an application attack. So Comdroid utilize this concept and analyze the interaction between Android application and determine Security threats in application modules.

Wei et al. [27] suggested a techniques called ProfileDroid which is a multilayer pro-

cessing method for monitoring and evaluating the application. This approach inspect the applications at four layers: The first one is at level of static analysis or app configuration, the second one is at level of user interaction, the third one is at the level of operating system, and the last one is at network level.

Guivarch et al. [28] have suggested a method called DroidMat which apply k-means and k-nearest neighbor (KNN) clustering algorithms on the features obtained using static analysis to identify the apps as malicious or not. This method considers the static details, such as app permission, deployment of different components, intent messages, and API calls for characterization of the Android apps.

Arp et al. [29] have proposed a model for the lightweight system which uses the Support Vector Machine (SVM) model on the features based on network information and API calls. This approach is very reliable in detecting various families of malware with an accuracy of 93% at a FPR of 1%, but it fails to detect that malware which shows the reflection and byte-code encryption

Yerima et al. [30] have applied the Random Forest ensemble learning approach to the features based on API calls, embedded commands, and permissions set. This approach achieved an accuracy of 97.6% with a true positive rate (TPR) of 97.3% and a false positive rate of 2.3% but it is exposed to the problem of code obfuscation and zero-day malware attacks.

Varsha et al. [31] have employed various algorithms like the Rotational Forest, SVM, and Random Forest with static features such as permissions, application components, filtered intents, opcodes, and strings extracted from executable and manifest files. The author reported a respective accuracy of almost 97% and 99% using Random Forest, on the features selected using Entropy-based Category Coverage Difference (ECCD) method and Weighted Mutual Information (WI) method but still, it is vulnerable to code obfuscation and zero-day attacks.

Fan et al. [32] have employed various algorithms like Decision Tree, Random Forest, PART, and KNN to detect the piggybacked apps using the features constructed from sensitive subgraph (SSG) and concluded that Random Forest achieves best results with a TPR of 95% at an FPR of 0.7%. The downside of this approach is a small size imbalanced dataset and problem of code obfuscation techniques, such as encryption and reflection.

Wang et al. [33] have applied various algorithms like Logistic Regression, Decision Tree, Random Forest, and Linear SVM on the features generated using static analysis and concluded that Logistic Regression yields the best performance with the TPR of 96% at FPR of 0.06%. This method uses a small size dataset so that it covers only limited malware attacks and shows inefficiency against the code obfuscation and zero-day attacks.

Xu et al. [34] have utilized the Long Short Term Memory (LSTM) network on automatically extracted features from XML files and bytecode and reported accuracy of 97.74% with a true positive rate of 97.96% and a false positive rate of 2.54%. Although this method is very effective against code obfuscation but still, it shows some limitations against the obfuscated malicious apps generated by Call Indirection.

Zhu et al. [35] have applied Ensemble Rotation Forest with permission set, sensitive APIs, system events, and permission-rate as input features and found an accuracy of 88.26% at the TPR of 88.40% with the precision of 88.16%.

DAPASA [32] aimed at detecting piggybacked malware on benign apps by using sensitive subgraphs to create five characteristics features invocation patterns. The features are then fed into machine learning algorithms like Random Forest, Decision Tree, k-NN, and PART and the best detection performance is observed with random forests.

Sharma et al. [36] used API calls and permissions for the development of malware detection schemes based on Naive Bayes and k-NN. Similarly, Cerbo et al. [37] has proposed a method to utilize the android permissions present in manifest file as features set to identify malicious applications. Other approaches, such as suggested by Hao et al. [38] performs static analysis to leverage the information embedded in byte code of application for predicting its behaviour.

Most of the Android malware static-analysis research primarily uses features such as md5 hash, signature, data flows, permissions, API (Application Programming Interface) calls extracted from manifest file of android application. These features lack the understanding of APK code organizations and object hierarchy, and therefore may be inadequate in detecting and forecasting the behaviors and maliciousness of an android applications.

Li et al. [39] have suggested a static analysis method based on Characteristic Tree and sought to apply a fresh approach to characterization of the API used for Android application at different levels of resolutions including packages, classes, functions and interface. A tree structure called "Characteristic Tree" is used to store certain knowledge about the use of APIs on various layers of tree structure and a comparison algorithm is developed to calculate the similarity of characteristic tree. This new detection process gives more rigorous insights into classifying and detecting Android malware of different kinds and code families.

3.2 Dynamic Analysis

Another techniques to detect Android malware is based on dynamic analysis where the malware could be detected during execution of the application. In dynamic analysis, features are extracted by collecting data from DroidBox or an application sandbox during the execution of the applications and utilized the features like system calls, network connection, memory usage, power utilization and intent call for malware detection. There are various strategies suggested by researchers for dynamic analysis on malware detection some of the recent work is discussed below.

Zhao et al. [40] have proposed a method, named AntiMalDroid which uses SVM as a classification model and dynamically picked software behavior signature as input features to train the model for malware detection. This approach achieves a detection rate of 93.3%, 90%, and 90% for three types of malware such as Geinimi, DroidDream, and Plankton respectively. The major limitation with this approach is the small data sample and more time consumption.

Wu et al. [41] have applied SVM with the useful static and dynamic features extracted from the logged data and obtained an accuracy of 86.1% with an F-score of 0.85. This approach is comparatively inefficient with low accuracy, more time consumption, and suffered from the problem of malware anti-emulation techniques.

Afonso et al. [42] have investigated various algorithms like Naive Bayes, Random Forest, SVM, and J48 with system call sequence, and Android API call trace as features and concluded that Random Forest obtains the best accuracy score of 96.66%. But this method may fail to detect some malware that does not show their behavior in the emulated environment.

Alzaylaee et al. [43] have evaluated the performance of malware detection on real devices instead of emulators by utilizing the various algorithms like Naive Bayes, Random

Forest, SVM, PART and multilayer perceptron on the dynamically obtained features. They have concluded that phone-based analysis, using the Random Forest classifier, shows better results with F-measure of 0.926 and TPR of 93.1%. Although, this work reduces the problem of malware anti-emulation but still, there is a requirement of an effective solution based on an alternative set of dynamic features using large sample datasets.

Mahindru et al. [44] have extracted a set of 123 dynamic permissions from the 11000 apps and evaluates these features using several classifiers like Simple Logistic, Naive Bayes, Decision Tree, and Random Forest. They have concluded that Simple Logistic is a better performer with an accuracy rate of 99.7% but this approach suffered from the problem of zero-day malware attacks as the sample size is very small.

Cai et al. [45] have used dynamic features based on method calls and intent-component communication and observed an F1-score of 97.39% with a precision of 97.53% and recall of 97.34% using Random Forest as the classifier.

Feng et al. [46] have applied the ensemble of multiple base classifiers like Linear SVM, Decision Tree, Random Forest, Extremely Randomized Trees, and Boosted Trees with dynamic behavior features like cryptographic operations, network operations, file operations, and system calls. The author has reported an accuracy of 98.18% with a true positive rate of 98.20% and a false positive rate of 1.61%. Similar to the limitations of dynamic analysis, this method also suffered from the malware anti-emulation and environmental shortcomings.

Xiao et al. [47] have proposed a method that uses system call sequence as input features with the LSTM network as a classifier and achieves an accuracy of 93.7% with a precision of 91.3%, recall of 96.6% and low FPR of 9.3%. This method achieves a reliable recall rate but lags in FPR value, as well as it is not much efficient as time and power consumption is high.

Ni et al. [48] have suggested a real-time malicious activity detection system that tracks API calls, permissions set and other real time features, such as device operations and utilized these features to model SVM and Naive Bayes algorithms for malware detection work.

DroidDolphin [49] has also used Support Vector Machine with features that were dynamically acquired. These features are the information related to memory utilization, network information, binder function information etc. Similarly, Enck et al. [50] simply performs dynamic taint analysis to monitor the flow of sensitive and confidential data via the third-party applications and detect any leakage of data to remote servers.

There are some other techniques known as hybrid analysis that leverages the approach of static and dynamic analysis for feature extraction and model training. Lindorfer et al. [51] have utilized SVM with RBF kernel and two linear classifiers with L_1 - and L_2 -regularized Logistic Regression to perform hybrid analysis and obtained an accuracy of 99.76%, 99.85%, and 99.83% for respective classifiers. In the same way, Su et al. [52] applied hybrid analysis to 1200 (900 benign and 300 infected) samples and evaluated several classifiers like Naive Bayes, SVM, and KNN and concludes that SVM performs better than the other classifiers with an accuracy of 97.4%. Similar to the limitations of the static and dynamic analysis, these methods also suffered from the problem of code obfuscation, malware anti-emulation, evasion, and high time consumption.

In recent time, application of machine learning and deep learning methods to Android malware detection and categorization have become an active research field. This section examines the related works of neural network and recurrent neural network for static and dynamic analysis in the identification and classification of Android malware. In

the Android application there is a file named `AndroidManifest.xml` which defines all the permissions and the API calls made by the application [53]. Along with this there are various other approaches to collect the feature set like Isohara et al. [54] have devised a mechanism for kernel based log collector to collect system calls and filters events and claimed that the system calls are more effective with the experiments of Android malware detection. These collected features can be possibly utilized in two ways for malware detection :

- Extract the features such as permission sets, API calls, intent filters etc. and apply classical machine learning techniques
- Convert the permission sets, API calls, Sensitive API, and Strings etc. into vector using word embedding and apply deep learning methods

The second approach is also utilized by many researchers but due to limited mobile computing power and limited battery backup these methods are not practically feasible. Some of the work devised by researchers in the field of Android malware detection using deep learning are as follows. Yuan et al. [55] has suggested a deep learning method for Android malware detection by utilizing more than 200 features from 3 categories such as permissions, Sensitive API and dynamic actions extracted from static and dynamic analysis and achieves an accuracy of 96%. Chen et al. [56] have proposed DroidVecDeep in which they extract 240 features from 4 main static type feature such as permissions, actions, sensitive API, and strings and transform these features into vector using word2vec. Further these vectors are modelled using a Deep Belief Networks which is a deep learning methods and achieves a promising accuracy of 99.1%.

3.3 Android malware detection using low-privileged monitorable features

Apart from malware detection work based on static and dynamic features extracted from apps, researchers have discussed some new approaches to malware detection based on low-privileged monitorable features. These methods mitigate the basic limitations of static and dynamic analysis. Shabtai et al. [20] have utilized Logistic Regression, Naive Bayes, and Decision Tree with various low-privileged monitorable features like CPU usage, battery consumption, and the amount of data packets sent over the network. The results of this method are not effective to a large extent because the models are trained on a small dataset so it didn't perform well for every malware family. Zheng et al. [57] have applied various classical techniques like Naive Bayes, Decision Tree, Random Forest and modern techniques like recurrent neural networks with long short-term memory (LSTM) on low-privileged monitorable features [23] and conclude that Random Forest outperformed the LSTM network. Wassermann et al. [22] have discussed the machine learning approach for the malware detection and app identification using SherLock data [23] and concluded that Decision Tree is the best performer with high accuracy. The downside of this experiment is that the Decision Tree may show the overfitting of data. Memon et al. [21] have utilized the SherLock data [23] with various machine learning models like Decision Tree, Random Forest, Support Vector Machines, Gradient Boosting etc. on the features selected using chi-square method. They claimed that Extended Gradient Boosting is the best performing classifier with the accuracy of 90%. The same algorithm achieves a highest precision value

of 0.91 and highest recall value of 0.90. The Extended Gradient Boosted model also shows the best F1-Score of 90.89% but this method has some limitations like it have utilized only quarter 3 dataset, as well as FPR value achieved in this experiment is 9.2% which is comparatively high. A brief overview of some of the work for Android malware detection is given in table 3.1.

Paper	Method	Features Set	Algorithms	Results
Zhu et al. [35]	Static	-permissions -sensitive APIs -system events -permission-rate	Ensemble Rotation Forest	Acc is 88.2% TPR is 88.4%
Xu et al. [34]	Static	-extract features from xml files and bytecode	LSTM (RNN)	Acc is 97.7% FPR is 2.5%
Cai et al. [45]	Dynamic	-method calls -ICC intent -app resources -system calls	NB, DT, RF, SVM, KNN	RF is best. TPR is 97.3% Prc is 97.5%
Xiao et al. [47]	Dynamic	-system call sequences	LSTM (RNN)	Acc is 93.7% TPR is 96.6% Prc is 91.3%
Zheng et al. [57]	Dynamic	-T4 Probe [23]	RF, LSTM	RF is best. Acc is 95.8%
Memon et al. [21]	Dynamic	-T4 Probe [23]	DT, KNN, NB, GBT, SVM, RF	GBT is best. Acc is 90% FPR is 9.2%
Proposed Work	Dynamic	-T4 Probe [23]	NB, DT, KNN, RF XGBoost	XGBoost is best. Acc is 93% FNR is 6.9% FPR is 6.5%

Table 3.1: Overview of some of the works for Android malware detection.

Note: NB=Naive Bayes; LR=Linear Regression; DT=Decision Tree; RF=Random Forest; KNN=K-Nearest Neighbors; LSTM=Long Short Term Memory; RNN=Recurrent Neural Network; GBT=Gradient Boosting Trees; SVM=Support Vector Machine; Acc=Accuracy; Prc=Precision.

3.4 Limitation of Methods

Analysis Types	Detection Methods	Limitations
Static	Signature Based	Identifies malware from previously existing signatures so that undiscovered malware can not be detected.
	Permission Based	May identify benign apps as suspicious because of very minor differences in permission set, so have high FPR.
	Dalvik Bytecode	It require high computation power on the device thus going to drain the battery and exhaust the memory.
	Anamoly Based	May categorize a benign apps as malicious if there are some warning signs in a benign application, such as more battery consumption, high traffic so high FPR.
Dynamic	Taint Analysis	Requires high computation on the device so it takes a lot of time for real time detection and make device slow.
	Emulation Based	Requires high utilization of memory and resources Device start to hang.
Hybrid	ML approach	Training costs is very high, FPR is high due to data set problems, data set is very old and slow identification.

Table 3.2: Limitations of Related Work

Chapter 4

Methodology

This section describes the methodology of the experiment and discussed a novel approach to Android malware detection using a significant smartphone data set composed of low-privileged monitorable features. The experiment is performed to employ various machine learning algorithms on the features obtained after the feature selection process and their performance is evaluated using various metrics.

4.1 Environmental configuration

This experiment is performed using an Apache Spark cluster computing framework. The Spark cluster was running on Elastic Map Reduce (EMR) platforms of Amazon Web Services (AWS) and the data was stored in AWS Simple Storage Service (S3). The Spark cluster is made of 12 nodes, out of which 11 nodes were arranged as slave nodes. Each node is installed with PySpark and composed of 8GB RAM and Quad-core processor.

4.2 Dataset

The dataset [23], used in this work is provided by the University of Ben-Gurion, called SherLock dataset. This significant smartphone data is generated from a continuing long-term data collection experiment by providing Samsung Galaxy S5 to 50 volunteers. The two Smartphone agents are involved in this data collection experiment: SherLock and Moriarty.

- **Sherlock:** SherLock is a data collection agent which captures various device metrics (such as battery usage, CPU usage and memory usage etc.) from a wide range of sensors and applications at a high sampling rate.

- **Moriarty:** Moriarty perpetrates varied cyber attacks on the user and records its activities in order to provide labels to SherLock dataset.

The primary objective of the dataset is to help safety professionals and research groups to develop a innovative methods to detect malicious behavior in smartphones implicitly in those devices where sensor data can be accessed without the privileges of the superuser(root).

The SherLock dataset is very distinct and unique as compared to existing public datasets due to the following reasons:

1. Temporal Resolution: The SherLock data set contains low monitorable attributes sampled at a higher frequency. In addition, the data from the motion sensor has 80-fold temporal coverage over other public existing datasets.

2. The Collected Data: The SherLock dataset provides detail information of device metrics like Linux level memory usage, CPU utilization, and scheduler details from everyday running application. In addition, it also sampled a broader variety of aggregated motion sensors data.

3. Explicit Labels: The SherLock dataset provides explicit labelling for every type of malware attacks and captures the device metrics and labels for various malware performing malicious activities.

Datset description

The SherLock dataset is organized into form of probe. A probe is a data table in which multiple sensors sharing the same interval are grouped to collect the data. The data set is decomposed into 15 different probes. Of the 15 probes, 8 are PULL probes which captures sensor data in fixed time interval and 7 are PUSH probes which captures sensor data as soon as the new information arrives.

PULL probes are those table which are sampled at regular frequency. For example system metrics is collected in every 5 seconds and therefore recorded in T4 PULL probe. The various PULL probes which provides different information are explained below.

- T0 probe : Contains Telephone, System and Hardware Information.
- T1 probe : Contains Location, Cell tower, Wi-fi and Bluetooth scan information.
- T2 probe : Contains Accelerometer, Gyroscope and Magnetic field information.
- T3 probe : Contains information about audio and light data.
- T4 probe : Contains CPU, memory, Network traffic, Io interrupts etc. information.
- App probe: Contains information like memory, CPU etc. for each running apps.

PUSH probes are those table which are sampled at the occurrence of specific events. For example, as soon as a new SMS message arrived it was recorded by the SMS PUSH probe instantly. There are various PUSH probes which provides different information are explained below.

- App package : Contains information about installed, updated and removed apps.
- Broadcast : Contains information about Broadcast intent.
- Call log : Contains information about calls.
- Moriarty : Contains information about Malware actions and Malware sessions.
- SMS log : Contains information about SMS status.
- Screen status: Contains information about screen on/off.
- User present: Contains information about interaction of user with apps.

In this work we use the combination of hardware and software features to identify the action of installed app being malicious or not. So the system specific data such as CPU utilization, amount of data transmission over network as well as memory utilization etc. are used to identify the behaviour of application. Considering this requirement, the T4 probe is selected which provides system specific data and Moriarty probe is selected which provides explicit label to T4 probe for the android malware detection work as well as App probe is also selected to identify the data related to which application. Therefore we select T4 probe, Apps probe and Moriarty probe for the malware detection work. The three selected probes are described in more detail as follows:

Moriarty Probe : The Moriarty probe has been tracking the data of various malware attacks carries out by Moriarty agent installed on devices. The record of each Moriarty probe is a log of the action taken by the user’s malware application. The log includes information of the action and session and it also contains other features such as the malware version and the time-tamp. Moriarty probe can be used to provide the label of the T4 probe and apps probe data. The malware application actually started either in a benign session or in a malicious session. In benign session only benign actions are performed but inside a malicious session malware can performed either malicious action or benign action.

System Probe(T4) : The system probe or T4 probe monitor and collects global device data in every five second. The recorded feature categories as CPU utilization, battery utilization, memory utilization, network details, I/O interrupts and storage information etc. At a given moment, each record in T4 probe is a log of the user’s global device data.

App Probe : The app probe tracked app data for each application installed on the device at every fixed time interval. The only relevant data for this research is the malware packages app data. Each record of the apps probe is a log of data at given time from the user’s malware app.

File Name	Description
T4 probe	Data is sampled and recorded at every 5 seconds. It contains CPU utilization of each core and utilization of memory, virtual memory and cache. It describe information about number of processes in user mode, kernel mode, number of threads created, running and blocked processes, and internal and external storage.
Moriarty probe	This table is updated when clue is recorded by the malicious agent. It describes behaviours, action details and type at specific timestamp.
App probe	The app probe tracked app data for each application installed on the device at every fixed time interval. Each record of the apps probe is a log of data at given time from the installed apps.

Table 4.1: Files selected for use

Data Preparation

The experiment of ML-based malware detection schemes is performed on the large dataset to get better performance. But due to the limited computing resources, the experiment is conducted on the quarter 3 of 2016 dataset with a size of 35 GB. As already stated system data is sampled in T4 probe and label of those records is present in Moriarty probe. So labelled points were generated by joining the Moriarty and T4 data tables. Only such T4 records were taken to join which has both benign and malicious data collected during the Moriarty application session. To obtain the labeled data points, the T4 probe and Moriarty probe are joined based on the closest timestamp for every user. Further, to identify the app associated with every instance, the relational join is performed between the T4-Moriarty probe (obtained in the previous step) and the Application probe to include the application related information [23]. So the final dataset contains the information about the behavior of each application with the captured system metrics. In the next step, we perform the feature selection to obtain the most important features used for training the classification model. The file selected for experiments had 34.23 million records after combining Moriarty and T4 to create labeled points. The statistics of data shows a class imbalance and as malicious samples are very high as compared to benign samples. To rectify this imbalance, equivalent positives and negatives data points were chosen and dataset was split in the ratio of 70:30, respectively for training and testing the classification models.

4.3 Feature Selection

In this work, the mutual information gain method is used for selecting the features to train the classification model. The mutual information $I(X;Y)$ can quantify any kind of relation between variables and if its value is large, it means two variables are closely related else they are not closely related. The value of mutual information can not be negative, if its value is zero it means two variables are independent of each other. Relative importance of the top 15 features obtained using the mutual information gain method has shown in Fig. 4.1. After analysing and performing the feature selection work, the top 40 significant features from an initial set of 130 features are selected to train the classification model so that it reduces the training time and complexity of the model. The methods used by our system and the one presented by Memon et al. [21] are very similar. Their system uses the features selected using the Chi-square method having a p-value > 0.05 , whereas our system uses the top 40 features selected using the mutual information gain method. The results obtained in this work shows that the futures selected using the mutual information gain method gives the better result for the Android malware detection on the SherLock dataset.

4.4 Machine Learning Classifications

The most enticing machine learning strategies have been chosen based on the findings of the work done on dynamic malware detection as discussed in section 3.2, various machine learning algorithms like Random Forest, Naïve Bayes and K-Nearest Neighbor have showed the promising results in malware detection by maximising the (TPR) and minimising the low false flags (FPR and FNR). Furthermore, recent studies on the use of

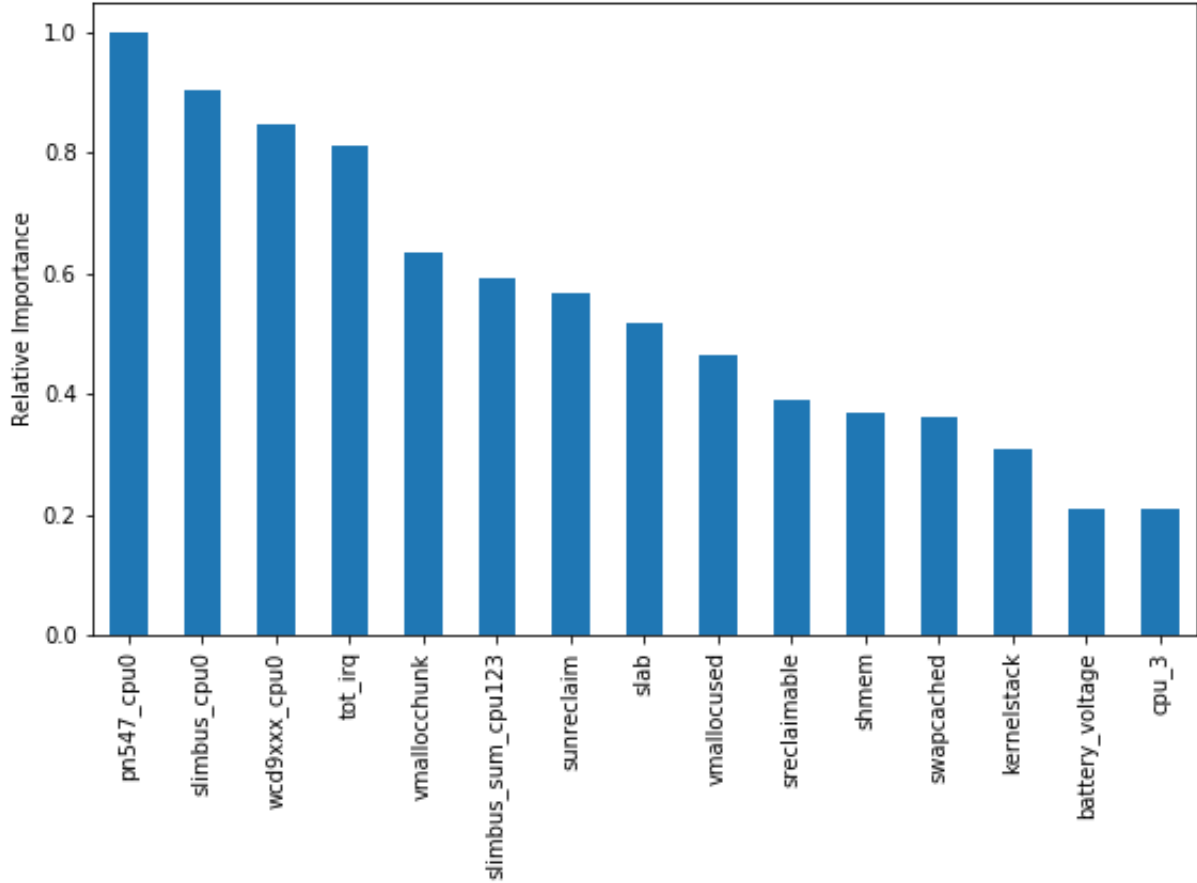


Figure 4.1: Top 15 features obtained using Mutual Information Gain method

Extended Gradient Boosting for mobile malware identification has shown encouraging results. Therefore, for the detection model, this thesis uses Naive Bayes (NB), K-Nearest Neighbor (K-NN), Decision Tree (DT), Random Forest (RF), and Extended Gradient Boosting (XGBoost) classifiers. Before selecting the classification model, the criteria of the domain such as high-dimensional feature space, sparse data, performance and computing constrains are analysed in detail. Since the performance of tree based algorithms for binary classification problem over high dimensional feature spaces is very significant and requires less computing power in real environment, so the selection of these algorithms in this work is justified. The top 40 features obtained after the feature selection process is used to train the models and their performance is investigated using the various evaluation metrics. The selected models are trained on the filtered data obtained after the pre-processing steps. The detection models aims to determine the class of application either malicious or benign and the performance of these trained models are evaluated using various evaluation metrics.

4.5 Evaluation Metrics

In order to evaluate the features obtained after the feature selection process, we apply the five selected classification models in this experiment as stated earlier, and examines their effectiveness using various performance metrics introduced earlier. The experiment

intends to improve the results of Memon et al. [21] and the results of the experiment, for all five classifiers, are presented in chapter 5.

The goal of the research is to : (i) Analyze and compare the performance of employed malware detection model; (ii) Analyze and identify the samples that are incorrectly classified in each model; (iii) Identify the reasons about samples incorrectly classified in one process but classified correctly by another; and lastly (iv) Enquire about the method employed for detection purpose is accurately separating the malicious and benign sample or not. Since any approach has inherent drawbacks, the distinction between them is not obvious from usual estimation. Therefore, for comparing the performance of all the selected models various evaluation metrics are utilized.

In this assessment, Android malware is recognized as a positive instance and benign application is recognized as a negative instance. Here, let the number of Android malware that is correctly detected is denoted by TP (true positive); the number of Android malware that is not detected as malicious is denoted by FN (false negative); the number of benign apps that are correctly detected is denoted by TN (true negative), and the number of benign apps that are detected as malicious is denoted by FP (false positive). The effectiveness of the trained models is evaluated using various metrics, based on the confusion matrix, such as true positive rate (TPR), false-positive rate (FPR), false-negative rate (FNR), accuracy, precision, and F1-score. Along with the confusion matrix, ROC curve and Precision-Recall curve is also utilized to analyze the detection performance and precision-recall tradeoff respectively for the employed models.

Chapter 5

Results

In order to evaluate the features obtained after the feature selection process, we apply the five selected classification models in this experiment as stated earlier, and examines their effectiveness using various performance metrics introduced earlier. The experiment intends to improve the results of Memon et al. [21] and performed using a balanced dataset having 70% of data for training and 30% of data for testing the models. The results of the experiment, for all five classifiers, are presented in Table 5.1. The percentage accuracy achieved by all the five classifiers is presented in Fig. 5.1, which shows that the XGBoost attains the highest classification accuracy of 93.25%, Random Forest attains the second-highest accuracy of 92.19%, and the Decision Tree achieves the third-highest accuracy of 87.74%. The respective accuracy score of KNN and Naive Bayes classifier is 84.22% and 70.79%. The methods used by our system and the one presented by Memon et al. [21] are very similar. Their system uses the features selected using the Chi-square method having a p-value > 0.05 , whereas our system uses the top 40 features selected using the mutual information gain method. We show that the features selected using the mutual information gain method gives the better result for the Android malware detection on the SherLock dataset as the highest accuracy achieved by our system using the XGBoost (Extreme Gradient Boosting) classifier is 93.25%, whereas the highest accuracy achieved by Memon et al. [21] using Gradient Boosted Trees is 90.24%. Our methods show the better result for the Random Forest also with an accuracy of 92.19% as compared to the accuracy of 89.67% reported by the same classifier in Memon et al. [21]. However, the Decision Tree used by Memon et al. [21] obtained better results in terms of accuracy value 89.72% and FPR value 9.65% as compared to our approach having an accuracy value 87.74% and FPR value 12.25% using the same algorithm. FPR and FNR denotes

Table 5.1: Comparison of the performance of classification algorithms

Model	TPR	FPR	FNR	Precision	F1-Score	Accuracy	AUC
Naive Bayes	62.5%	21.3%	37.5%	73.65%	67.62%	70.79%	78.11%
KNN	83.65%	15.23%	16.34%	83.95%	83.80%	84.22%	91.48%
Decision Tree	87.74%	12.25%	12.25%	87.21%	87.47%	87.74%	87.72%
Random Forest	92.42%	8.01%	7.57%	91.65%	92.04%	92.19%	97.56%
XGBoost	93.02%	6.52%	7.0%	93.14%	93.08%	93.25%	97.87%

the total misclassified records but FPR is not considered as critical as FNR because the latter directly indicates the undetected malware, whereas FPR indicates the benign apps that have been detected as malicious. So a low value of FPR and FNR is desirable to make our detection system more accurate and less time-consuming. Fig. 5.1 shows the

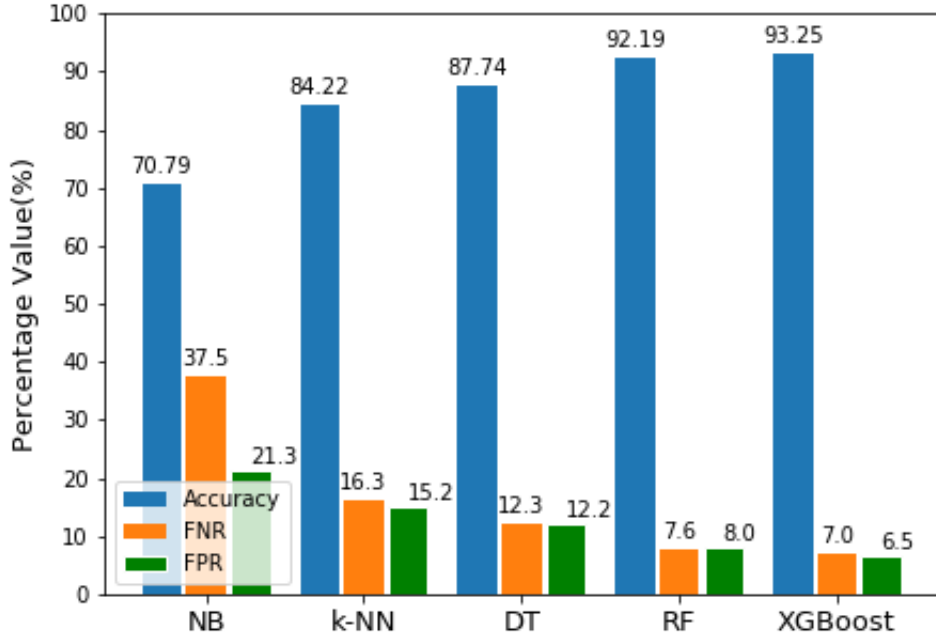


Figure 5.1: Measure of Accuracy, FNR, and FPR for all five classifiers

percentage value of FPR and FNR observed by all the classifiers. XGBoost has achieved the lowest FPR and FNR values compared to other algorithms, which is 6.5% and 7% respectively. The second-best performance is achieved by the Random Forest with an FPR of 8% and an FNR of 7.6%. The FNR values achieved by the other classification models as shown in Fig. 5.1 is 37.5%, 16.3%, and 12.3% respectively for Naive Bayes, KNN, and Decision Tree classifier. Similarly, the FPR values achieved by the other classification models are also shown in Fig. 5.1, which are 21.3%, 15.2%, and 12.2% respectively for Naive Bayes, KNN, and Decision Tree. From the results shown in Table 5.1, we conclude that the XGBoost classifier outperforms the other classification models in terms of the highest accuracy and the lowest FPR and FNR values, as it reduces the risk of undetected malware and lowers the false notification of malware detection. Comparing to Memon et al. [21], our approach shows a better performance in terms of FPR also as the lowest FPR value achieved in our proposed work is 6.52% using XGBoost classifier, whereas they achieved the lowest FPR of 9.2% using Gradient Boosted Trees. The Random Forest in our approach also shows a better FPR value of 8% compared to the FPR value of 10.16% achieved by the same algorithm in Memon et al. [21].

Further to assess and compare the detection performance of every selected classifier, the ROC curve and AUC values are computed and presented in Fig. 5.2. The ROC curves indicate that XGBoost and Random Forest have similar detection performance with a high value of TPR at a low value of FPR. The AUC values of the Random Forest and XGBoost classifier as shown in Fig. 5.2 are 97.6% and 97.9% respectively which are almost equal and hence both approaches have a similar detection performance. The detection performance of XGBoost shows a TPR of 0.93 at an FPR of 0.065 and Random Forest shows a TPR of 0.92 at an FPR of 0.08. From Fig. 5.2, we conclude that the XGBoost and Random Forest have better detection performance in terms of ROC-AUC values compared to other classifiers. The performance of Random Forest is very effective with this dataset due to two reasons [24]. First, the use of out-of-bag error as an estimate

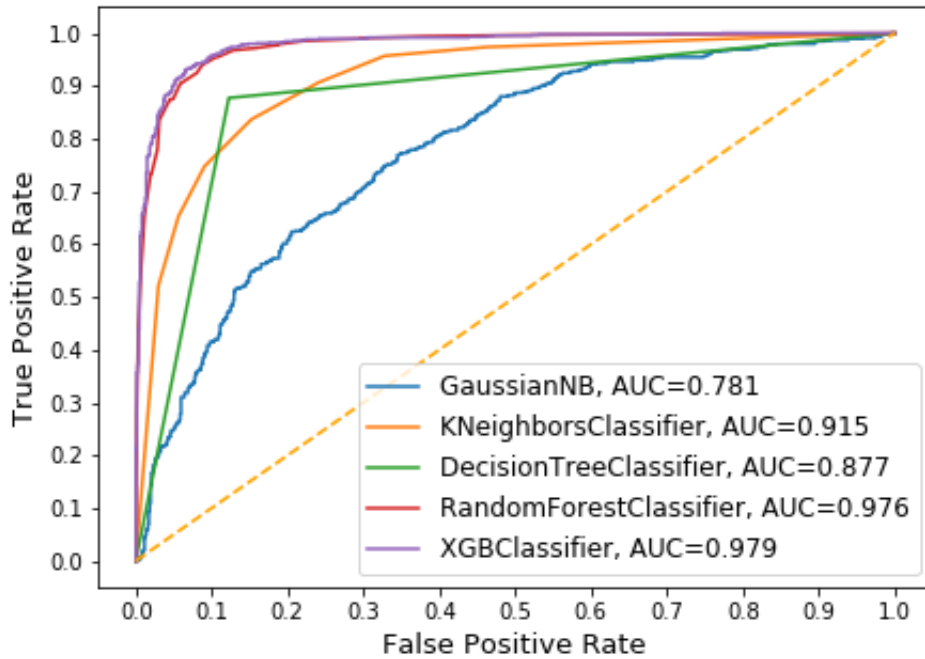


Figure 5.2: ROC curves for all five classifiers

for generalizing the error improves its performance. Second, being an ensemble classifier it prevents the overfitting of data as it yields the limited value of generalization error even after adding more trees to Random Forest. On the other hand, the performance of XGBoost is very effective, as it avoids the overfitting of data, reduces error rate, and performs faster than other classifiers, due to regularization nature and parallel processing implementation [25].

Model	Precision	Recall	F1-Score
NB	0.69	0.79	0.73
KNN	0.84	0.85	0.85
DT	0.88	0.88	0.88
RF	0.93	0.92	0.92
XGB	0.93	0.93	0.93

Table 5.2: Results For Benign Applications

We performed label wise experimental studies to make sure that the classification models work equally well in tracking both malicious and benign applications. Table 5.2 and Table 5.3 show that Naive Bayes has the lowest precision and recall value for both benign and malicious applications whereas, the XGBoost and Random Forest classifier has significant performance for both applications in terms of precision and recall value. As evident from Table 5.2 and Table 5.3, XGBoost attains the highest precision and recall value of 0.93 for both the malicious and benign classes whereas, Random Forest achieves the second-highest precision and recall value of almost 0.92 for both the classes. So in terms of the precision-recall trade-off for both classes, XGBoost performs slightly better than the Random Forest as precision and recall values get closer to 1 implies data is perfectly separable in two classes. Table 5.2 and Table 5.3 show that for both benign and malicious classes, XGBoost has the highest F1-score of 0.93 and Random Forest has the

Model	Precision	Recall	F1-Score
NB	0.74	0.62	0.68
KNN	0.84	0.84	0.84
DT	0.87	0.88	0.87
RF	0.92	0.92	0.92
XGB	0.93	0.93	0.93

Table 5.3: Results For Malicious Applications

second-highest F1-score of 0.92, which indicates that XGBoost has perfect precision and recall value. From the above discussions, we conclude that the top two performers for the detection of both malicious and benign applications are the XGBoost and Random Forest classifier.

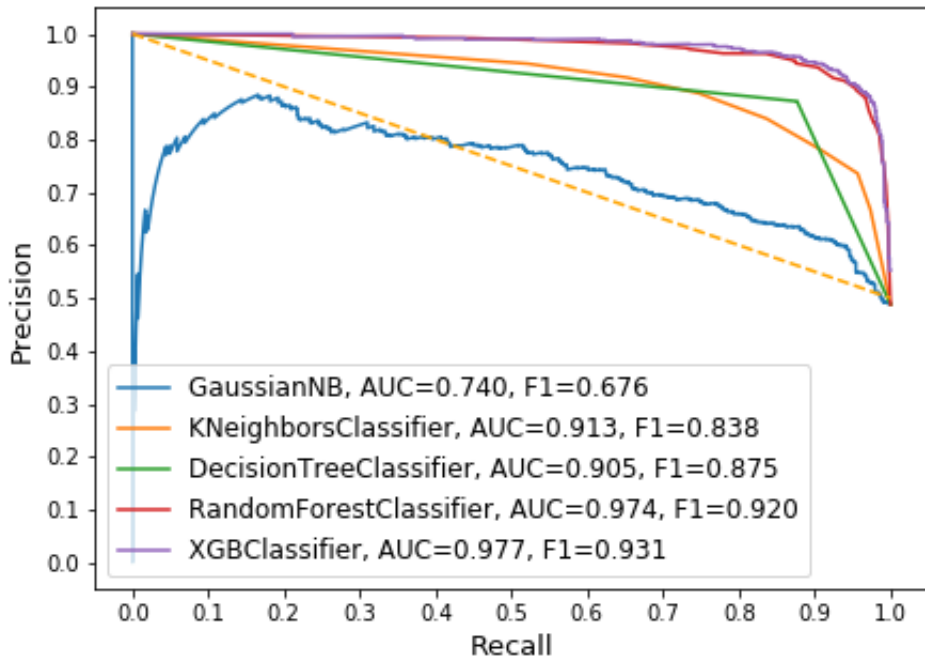


Figure 5.3: Precision-Recall curves for all five classifiers

To analyze the precision-recall tradeoff for all classification model, the Precision-Recall curve is computed as shown in Fig. 5.3 for different thresholds. The AUC (area under curve) value and F1 value for all classification models are also shown in the above Fig. 5.3 which shows that in term of precion and recall XGBoost is also the best performer with an AUC value of 97.7% and F1 value of 93.1%. The second best performance is shown by the Random Forest with AUC and F1 value of 97.4% and 92% respectively. Precision-Recall tradeoff is a very important parameter to measure the effectiveness of model. High precision value shows reduced false positive rate and high recall value shows reduced false negative rate. So the AUC value closer to 1 corresponds to high precision and high recall and thus reduces the error rate of the model. Hence XGBoost have minimum error rate compared to other models.

Chapter 6

Conclusion and Future Work

In this work, we have investigated the performance of various classifiers like Naive Bayes, KNN, Decision Tree, Random Forest, and XGBoost for Android malware detection using low-privileged monitorable features. To train our model, we have used system-specific features selected using the mutual information gain method from the T4 probe sampled in the SherLock dataset. The results obtained in this work shows that the futures selected using the mutual information gain method gives the better result for the Android malware detection on the SherLock dataset. The findings of this research show that the XGBoost and Random Forest are the top two performers with the respective accuracy of 93.25% and 92.19%. XGBoost classifier is the most accurate in detecting the malware with 93% overall values of precision, recall, and accuracy. In terms of FNR and FPR values, XGBoost also outperforms the other classifiers with respective values of 7.0% and 6.52%. Naive Bayes, KNN, and Decision Tree are not that effective as their accuracy score is 70.79%, 84.22%, and 87.74% respectively. If we consider the ROC curves, the detection performance of XGBoost is best with a TPR of 0.93 at an FPR of 0.065 and Random Forest has the second-best detection rate with a TPR of 0.92 at an FPR of 0.08, which indicates that XGBoost has the better classification power. So in this research, we employ various machine learning techniques and conclude that XGBoost and Random Forest classifiers are the top two performers for Android malware detection. After all there is always some scope for the improvement in any work. In this work there is also a scope of improving the results by utilizing the large amount of data. Along with this there is also a scope of selecting the set of good features which helps in improving the detection accuracy. Further, data pre-processing can help to achieve a better result by removing the anomalous data.

Bibliography

- [1] “The android software stack,” <https://developer.android.com/guide/platform>, accessed: 2020-10-29.
- [2] “K-nearest neighbors,” <https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d>, accessed: 2020-10-29.
- [3] “Classifying data with decision trees,” <https://elf11.github.io/2018/07/01/python-decision-trees-acm.html>, accessed: 2020-10-29.
- [4] “Random forests,” <https://towardsdatascience.com/random-forests-and-decision-trees-from-scratch-in-python-3e4fa5ae4249>, accessed: 2020-10-29.
- [5] “Smartphone market share,” 2020, (Accessed : July 2020). [Online]. Available: <https://www.idc.com/promo/smartphone-market-share/o>
- [6] R. H. Niazi, J. A. Shamsi, T. Waseem, and M. M. Khan, “Signature-based detection of privilege-escalation attacks on android,” in *2015 conference on information assurance and cyber security (CIACS)*. IEEE, 2015, pp. 44–49.
- [7] J. Oberheide and C. Miller, “Dissecting the android bouncer,” *SummerCon2012, New York*, vol. 95, p. 110, 2012.
- [8] R. A. Botha, S. M. Furnell, and N. L. Clarke, “From desktop to mobile: Examining the security experience,” *Computers & Security*, vol. 28, no. 3-4, pp. 130–137, 2009.
- [9] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, “Evolution, detection and analysis of malware for smart devices,” *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2013.
- [10] A. K. Gahalaut and P. Khandnor, “Reverse engineering: an essence for software re-engineering and program analysis,” *International Journal of Engineering Science and Technology*, vol. 2, no. 06, pp. 2296–2303, 2010.
- [11] C. Urcuqui-López and A. N. Cadavid, “Framework for malware analysis in android,” *Sistemas y Telemática*, vol. 14, no. 37, pp. 45–56, 2016.
- [12] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 421–430.
- [13] A. Aprville and R. Nigam, “Obfuscation in android malware, and how to fight back,” *Virus Bulletin*, 2014.

- [14] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, “The evolution of android malware and android analysis techniques,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–41, 2017.
- [15] “Droidbox: An android application sandbox for dynamic analysis.” [Online]. Available: <https://github.com/pjplantz/droidbox>
- [16] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, “Morpheus: automatically generating heuristics to detect android emulators,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 216–225.
- [17] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, “Rage against the virtual machine: hindering dynamic analysis of android malware,” in *Proceedings of the Seventh European Workshop on System Security*, 2014, pp. 1–6.
- [18] S. R. Choudhary, A. Gorla, and A. Orso, “Automated test input generation for android: Are we there yet?(e),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.
- [19] T. Isohara, K. Takemori, and A. Kubota, “Kernel-based behavior analysis for android malware detection,” in *2011 seventh international conference on computational intelligence and security*. IEEE, 2011, pp. 1011–1015.
- [20] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, *andromaly: a behavioral malware detection framework for android devices*. Journal of Intelligent Information Systems, 38(1):161–190, 2012.
- [21] L. U. Memon, N. Z. Bawany, and J. A. Shamsi, “A comparison of machine learning techniques for android malware detection using apache spark,” *Journal of Engineering Science and Technology*, vol. 14, no. 3, pp. 1572–1586, 2019.
- [22] S. Wassermann and P. Casas, “Bigmomal: Big data analytics for mobile malware detection,” in *Proceedings of the 2018 Workshop on Traffic Measurements for Cybersecurity*, 2018, pp. 33–39.
- [23] Y. Mirsky, A. Shabtai, L. Rokach, B. Shapira, and Y. Elovici, “*Sherlock vs moriarty: A smartphone dataset for cybersecurity research*”, in *Proceedings of the 2016 ACM workshop on Artificial intelligence and security*, ACM, 2016.
- [24] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [25] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.
- [26] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, *Analyzing inter-application communication in android*. In International conference on Mobile systems, applications, and services, 2011.
- [27] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, *Profiledroid: Multi-layer profiling of android applications*. In International conference on Mobile computing and networking, 2012.

- [28] C. Guivarch and S. Hallegatte, “*2c or not 2c?*” *SSRN Electronic Journal*, 2012.
- [29] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, “Drebin: Effective and explainable detection of android malware in your pocket.” in *Ndss*, vol. 14, 2014, pp. 23–26.
- [30] S. Y. Yerima, S. Sezer, and I. Muttik, “High accuracy android malware detection using ensemble learning,” *IET Information Security*, vol. 9, no. 6, pp. 313–320, 2015.
- [31] M. Varsha, P. Vinod, and K. Dhanya, “Identification of malicious android app using manifest and opcode features,” *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 2, pp. 125–138, 2017.
- [32] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, “Dapasa: detecting android piggybacked apps through sensitive subgraph analysis,” *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, 2017.
- [33] X. Wang, W. Wang, Y. He, J. Liu, Z. Han, and X. Zhang, “Characterizing android apps’ behavior for effective detection of malapps at large scale,” *Future generation computer systems*, vol. 75, pp. 30–45, 2017.
- [34] K. Xu, Y. Li, R. H. Deng, and K. Chen, “Deeprefiner: Multi-layer android malware detection system applying deep neural networks,” in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 473–487.
- [35] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, “Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model,” *Neurocomputing*, vol. 272, pp. 638–646, 2018.
- [36] A. Sharma and S. K. Dash, “Mining API calls and permissions for Android malware detection,” in *Cryptology and Network Security, Cham, Switzerland:Springer Int*, 2014.
- [37] F. D. Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, *Detection of malicious applications on android os*. In *Computational Forensics*, 2011.
- [38] H. Hao, V. Singh, and W. Du, *On the effectiveness of api-level access control using bytecode rewriting in android*. In *ACM SIGSAC symposium on Information, computer and communications security*, 2013.
- [39] Q. Li and X. Li, “Android malware detection based on static analysis of characteristic tree,” in *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2015.
- [40] M. Zhao, F. Ge, T. Zhang, and Z. Yuan, *Antimaldroid: An efficient svm-based malware detection framework for android*. In *Information Computing and Applications*, 2011.
- [41] W.-C. Wu and S.-H. Hung, “Droiddolfin: a dynamic android malware detection framework using big data and machine learning,” in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, 2014, pp. 247–252.

- [42] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, “Identifying android malware using dynamically obtained features,” *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015.
- [43] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, “Emulator vs real phone: Android malware detection using machine learning,” in *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, 2017, pp. 65–72.
- [44] A. Mahindru and P. Singh, “Dynamic permissions based android malware detection using machine learning techniques,” in *Proceedings of the 10th innovations in software engineering conference*, 2017, pp. 202–210.
- [45] H. Cai, N. Meng, B. Ryder, and D. Yao, “Droidcat: Effective android malware detection and categorization via app-level profiling,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.
- [46] P. Feng, J. Ma, C. Sun, X. Xu, and Y. Ma, “A novel dynamic android malware detection system with ensemble learning,” *IEEE Access*, vol. 6, pp. 30 996–31 011, 2018.
- [47] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, “Android malware detection based on system call sequences and lstm,” *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.
- [48] Z. Ni, M. Yang, Z. Ling, J. N. Wu, and J. Luo, *Real-time detection of malicious behavior in Android apps*. Proc. Int. Conf. Adv, 2016.
- [49] R. Riasat, “*Machine learning approach for malware detection by using apks*, 2017.
- [50] W. Enck, P. Gilbert, and B. G., *Chun*. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: L. P, 2009.
- [51] M. Lindorfer, M. Neugschwandtner, and C. Platzer, *MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis*. Proc, 2015.
- [52] M. Y. Su, J. Y. Chang, and K. T. Fung, *Machine learning on merging static and dynamic features to identify malicious mobile apps*. Proc, 2017.
- [53] R. A. Botha, S. M. Furnell, and N. L. Clarke, “From desktop to mobile: Examining the security experience,” *Computers Security*.
- [54] T. Isohara, K. Takemori, and A. Kubota, “Kernel-based behavior analysis for android malware detection.” IEEE Computer Society, 2011.
- [55] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, “Droid-sec,” *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM 14*, 2014.
- [56] T. Chen, Q. Mao, M. Lv, H. Cheng, and Y. Li, “Droidvecdeep: Android malware detection based on word2vec and deep belief network.” *TIIS*, vol. 13, no. 4, pp. 2180–2197, 2019.

- [57] Y. Zheng and S. Srinivasan, “Mobile app and malware classifications by mobile usage with time dynamics,” in *International Conference on Advanced Information Networking and Applications*. Springer, 2019, pp. 595–606.