Android Malware Detection Using Static Analysis, Machine Learning and Deep Learning

PhD Thesis

Fawad Ahmad

Computer and
Information Sciences

University of Strathclyde, Glasgow

Nov 22, 2020

Declaration

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination, which has led to the award of a degree. The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.50. The due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

Acknowledgements

I have received a great deal of support and assistance throughout my PhD research. I am deeply grateful to my supervisors for their insightful suggestions at every stage of the research. I would like to thank my first supervisor Dr Marc Roper whose knowledge and expertise was invaluable in shaping my research objectives, research questions, and methodology. I would also like to thank my second supervisor Dr Sotirios Terzis whose input to my literature review was extremely helpful.

I would like to express my sincere gratitude to Mr Peter Knapp, Managing Director of Orion Practice Management Systems Limited, for providing funding opportunities and support to pursue my PhD.

I would like to thank my family, and friends for their continuous support and providing much-needed distraction outside of my research.

Abstract

Android has been a dominant mobile operating system since 2012 as shown in Figure 1. This popularity coupled with a ubiquitous usage of smartphone in all aspects of our lives, e.g. online banking, social networking, and online shopping etc. have made Android a lucrative target for malware developers.

To combat the threat of malware stealing our private information, researchers have suggested various techniques for detecting Android malware. Broadly speaking, three primary techniques have been used for malware detection. Static Analysis, performed without running the application, has been used to generate signatures of malware, that can be used to differentiate between malware and benign applications. Another technique, Dynamic Analysis, has been used to create a behaviour profile of malware and benign applications by executing them in a controlled environment and monitoring their behaviour to detect malware. Hybrid Analysis has been used to utilise signatures generated from static analysis and behaviour profile created from dynamic analysis for detecting Android malware. In recent years, complementary techniques such as Machine Learning and Deep Learning have been used to extract features from the three primary analysis techniques and feed them to several algorithms for classification purposes. Deep Learning is a subfield of Machine Learning that relates to structuring algorithms in layers to mimic human neural network. The artificial neural network is used to solve complex problems using different algorithms.

In this dissertation, firstly, a systematic review is presented to amalgamate current approaches for detecting Android malware, and custom-built malware detection technologies. As a result of the literature evaluation, a taxonomy is suggested for Android malware detection. Furthermore, trends in the usage of the major analytical techniques and complementary techniques are shown. Research gaps in the Android malware detection area are identified for future research direction.

Secondly, Droid Fence, a custom-built web-based framework, for managing experiments is developed. Droid Fence automates the extraction of the required features from malware and benign applications directory by conducting static analysis via a frontend. Next, Droid Fence completes the automated process by storing the extracted features against each application record in a relational database, feeding them to the required machine learning and deep learning algorithms, storing the result into the database, and finally displaying the outcome of each experiment.

Thirdly, developed an approach that amalgamates a set of permissions, services, and six other features (usage of https, database, dynamic code, native code, reflection, and cryptography)

to generate a matrix that is used for detecting malware effectively. To the best of our knowledge, this is a novel approach that combines these features to detect malware. Droid Fence is evaluated on a dataset of 13191 applications consisting of 5787 malware and 7404 benign applications. Our results show that Droid Fence is very effective when it utilises a Sequential (Deep Learning) algorithm to detect malware, achieving accuracy, F1-measure, precision, and recall scores of 0.971, 0.967, 0.977, and 0.956 respectively. Our experiments, conducted using Droid Fence, demonstrates that deep learning Sequential algorithm scored consistently highly when compared against eight machine learning algorithms. However, the difference between the accuracy scores achieved by the Sequential (97.1%) and Random Forest Classifier (95.8%) is minimal in comparison with the remaining algorithms used in our experiments. We used a stratified k-fold cross-validation method, and the result was compared for four metrics: accuracy, F1 score, precision, and recall.

Finally, a conclusion and future research direction are suggested for both Android malware detection area and improvement in Droid Fence.

Table of Contents

Abstract ... 3
List of Figures ... 8
List of Tables ... 9
1. Introduction ... 10
1.1 Research Questions ... 12
1.2 Contribution ... 14
1.3 The organisation of the thesis ... 15
2. Android Operating System ... 17
2.1 Android Components ... 18
2.1.1 Android Platform Architecture ... 18
2.1.2 Android Applications ... 18
2.1.3 Java API Framework ... 19
2.1.4 Native C/C++ Libraries ... 19
2.1.5 Android Runtime ... 19
2.1.6 Hardware Abstraction Layer ... 20
2.1.7 Linux Kernel ... 20
2.2 Inter-Component Communication ... 20
2.3 Application Configuration ... 21
2.4 Android Security Model ... 22
3. Android Malware ... 24
3.1 The Evolution of Malware ... 24
3.2 Malware Family ... 26
3.3 Malware Attacks and Evasions ... 29
3.4 Malware Monetisation Strategies ... 33
4. Systematic Literature Review ... 34
4.1 Related Work ... 34
4.2 Research Method ... 35
4.2.1 Search Strategy ... 36
4.2.2 Digital Libraries ... 37
4.2.3 Inclusion and Exclusion Criteria ... 38
4.3 Summary ... 41
5. A Classification of Android Malware Security Analysis ... 42
5.1 Signature-Based Detection ... 43
5.2 Behaviour-Based Detection ... 47
5.3 Machine Learning-Based Detection ... 51
5.4 Taxonomy of Android Malware Detection ... 57

5.5 Techniques for Malware Detection ... 58
5.5.1 Static Analysis Techniques for Malware Detection ... 59
5.5.2 Dynamic Analysis Techniques for Malware Detection ... 63
5.5.3 Hybrid Analysis Techniques for Malware Detection ... 68
5.5.4 Summary ... 73
5.6 Technologies for Android Malware Detection ... 74
5.7 Discussion, Trends, and Research Gap ... 81
5.7.1 Trends in Analysis Techniques Usage ... 82
5.7.2 Research Gap in Android Malware Detection ... 84
6. Droid Fence – Overview and Background on Features ... 91
6.1 Overview ... 91
6.1.1 Experiment Manager ... 92
6.1.2 Feature Extraction ... 93
6.1.3 Detection System ... 94
6.1.4 Database ... 96
6.2 Droid Fence - Background on selected features ... 98
6.2.1 Permissions ... 98
6.2.2 Services ... 99
6.2.3 Reflection ... 100
6.2.4 Dynamic code loading ... 101
6.2.5 Native Code ... 102
6.2.6 Database ... 103
6.2.7 HTTPs ... 104
6.2.8 Cryptographic code ... 104
6.2.9 Feature Extraction Deception ... 105
7. Droid Fence – Methodology & Performance Evaluation ... 107
7.1 Experimental Settings ... 107
7.2 Methodology ... 108
7.3 Machine Learning Algorithms ... 108
7.3.1 Logistic Regression ... 109
7.3.2 Linear Discriminant Analysis ... 110
7.3.3 K Nearest Neighbor Classifier ... 111
7.3.4 Decision Tree Classifier ... 112
7.3.5 Gaussian NB ... 113
7.3.6 Support Vector Classifier ... 114
7.3.7 XGB Classifier ... 115
7.3.8 Random Forest Classifier ... 116

7.3.9 Sequential (Deep Learning) ... 117
7.4 Evaluation Metrics ... 118
7.5 Features ... 119
7.6 Dataset Collection ... 123
7.7 Experiment 1 - Performance Comparison ... 124
7.8 Experiment 2 - Performance Comparison ... 129
7.9 Result Comparison between Experiment 1 and 2 ... 135
7.10 Experiment 3 – Performance Comparison ... 136
7.11 Experiment 4 – Performance Comparison ... 141
7.12 Comparison with related methods ... 147
8. Conclusion and Future Work ... 150
8.1 Research Questions ... 150
8.1.1 Research Question 1 (RQ1) ... 151
8.1.2 Research Question 2 (RQ2) ... 152
8.1.3 Research Question 3 (RQ3) ... 155
8.1.4 Research Question 4 (RQ4) ... 157
8.1.5 Research Question 5 (RQ5) ... 157
8.1.6 Research Question 6 (RQ6) ... 158
8.1.7 Research Question 7 (RQ7) ... 158
8.1.8 Supplementary Questions ... 158
8.2 Future Work ... 159
9. References ... 161

List of Figures

Figure 1: Android OS market share of smartphone sales to end users from 2009 to 2021 [13] 17
Figure 2: The Android Software Stack [14] 18
Figure 3: Global market share [28] 25
Figure 4: Detection of evasion techniques by several papers 30
Figure 5: Scope of the Survey 39
Figure 6: Number of selected papers by publishing year 41
Figure 7: Categorisation of Android malware security analysis 42
Figure 8: A Taxonomy of Android malware detection 57
Figure 9: Malware Detection techniques by the number of surveyed papers 82
Figure 10: Trends in Analysis techniques referenced in surveyed papers 83
Figure 11: Trends in Machine/Deep learning techniques referenced in surveyed papers 84
Figure 12: Droid Fence - Overview 92
Figure 13: Droid Fence Feature Extraction - Application info 94
Figure 14: Droid Fence - Data Preparation 95
Figure 15:Droid Fence - Database ER Diagram 96
Figure 16: Android Manifest - Permission Declaration 99
Figure 17: Android Manifest - Service Declaration 100
Figure 18: Android - Reflection Example Code 101
Figure 19: Android - Dynamic Class Loading [173] 102
Figure 20: Android - Database Access [176] 103
Figure 21: Android - HTTPS Connectivity [177] 104
Figure 22: Android - Cryptography Code [178] 105
Figure 23: Experiment Details 109
Figure 24: Six specified features used in Experiment 2, 3, and 4 122
Figure 25: Accuracy Metric for Experiment 1 for Training and Validation Data Set 125
Figure 26: F1 Score Metric for Experiment 1 for Training and Validation Data Set 126
Figure 27: Precision Metric for Experiment 1 for Training and Validation Data Set 127
Figure 28: Recall Metric for Experiment 1 for Training and Validation Data Set 127
Figure 29: Confusion Matrix for Validation Data Set - Experiment 1 129
Figure 30: Accuracy Metric for Nine Algorithms for Training and Validation Data Set 130
Figure 31: F1 Score Metric for Nine Algorithms for Training and Validation Data Set 131
Figure 32: Precision Metric for Nine Algorithms for Training and Validation Data Set 132
Figure 33: Recall Metric for Nine Algorithms for Training and Validation Data Set 132
Figure 34: Confusion Matrix for Validation Data Set 134
Figure 35: Accuracy Metric for Experiment 3 for Training and Validation Data Set 137
Figure 36: F1 Score Metric for Experiment 3 for Training and Validation Data Set 138
Figure 37: Precision Metric for Experiment 3 for Training and Validation Data Set 138
Figure 38: Recall Metric for Experiment 3 for Training and Validation Data Set 139
Figure 39: Confusion Matrix for Validation Data Set - Experiment 3 141
Figure 40: Accuracy Metric for Experiment 4 for Training and Validation Data Set 142
Figure 41: F1 Score Metric for Experiment 4 for Training and Validation Data Set 143
Figure 42: Precision Metric for Experiment 4 for Training and Validation Data Set 143
Figure 43: Recall Metric for Experiment 4 for Training and Validation Data Set 144
Figure 44: Confusion Matrix for Validation Data Set - Experiment 4 146

List of Tables

Table 1: List of malware families and their descriptions [25] [34] [36] [37] [38] [39].............. 28
Table 2: List of papers addressing each detection evasion technique 31
Table 3: The breakdown of search terms and their results... 40
Table 4: Publications utilising the required features 88
Table 5: Droid Fence - Database Tables Descriptions 97
Table 6: Top Forty Permissions used in Experiment 2, 3, and 4 120
Table 7: Top Twenty Services used in Experiment 1, 2, 3, and 4....................................... 121
Table 8: Dangerous permissions used in Experiment 1 123
Table 9: Detection Result Comparison of Nine Algorithms on Validation Data Set 128
Table 10: Detection Result Comparison of Nine Algorithms on Validation Data Set 133
Table 11: Best Algorithms in Experiment 1 and 2. 135
Table 12: Confusion Matrix for Experiment 1 and 2.. 136
Table 13: Detection Result Comparison of Experiment 3 on Validation Data Set 139
Table 14:Detection Result Comparison of Experiment 4 on Validation Data Set 144
Table 15: Comparison with related methods 149

1. Introduction

Android, having more than 2.9 million applications on Android Google Play alone, has emerged as the leading mobile platform [1]. The markets for mobile applications, for instance, Android Google Play, have fundamentally transformed how consumers receive software, with daily updating and inclusion of many applications. The rapid expansion of the applications market, coupled with the pervasive nature of applications provided on such platforms, has seen a parallel rise in the sophistication and number of security threats targeting mobile platforms [2].

Studies undertaken in the last few years indicate that mobile markets have vulnerable or malicious applications, thus compromising millions of gadgets [2]. Malware has threatened computer systems for many years. With the invention of Android systems and significant market share, it was only a matter of time before malware developers developed them for the Android platform. Android market share has increased significantly in the last few years; with the growth attracting multiple malware attacks which keep on changing concerning complexity and scope [2]. Since its release in 2008, the Android platform has experienced immense growth with significant share over the years. The popularity and extensive usage of the platform are associated with a heightened interest from malware developers with varied malicious interests. Multiple aspects and vulnerabilities of the platform have been exploited with continued improvement and enhancement of security features. Different frameworks with varying capabilities and limitations have progressively been developed. The popularity of the Android platform has seen market share globally reach more than 86% according to recent statistics. Estimates place the figure at more than 1.5 billion monthly Android users. Android is the leader in the market, with a dominance of 86.6% when compared to 13.4% of Apple iOS [3]. Policies employed by Google have enabled the Android platform to experience this kind of growth and acceptance. Its open policy has allowed millions of applications to be available on the platform with a high level of tolerance regarding verification and release. Many culprits are responsible for Android malware growth, and some of them are not technical; for instance,

the absence of a regulatory body in open markets along with the lack of consequences for the people providing applications with malicious potential or vulnerabilities. The scenario is likely to increase because mobile applications are becoming ubiquitous and complex.

Some of the policies applied by Google also pose security hazards to users. Open policies are allowing third parties to have an unofficial application store where users can download applications without verifying security and authenticity. Digital certificates usage on the Android platform is not strictly managed, meaning that some application developer cannot be traced to their initial developers through digital signatures [4]. The lack of digital certificates usage makes it easy for malware developers to release cracked versions of authentic applications as well as Trojan horses which may be disguised as regular applications. As a result, the Android ecosystem has become one of the most targeted platforms by malware architects with many malicious intentions [5]. Even though the malware threat is apparent, Google policymakers maintain that the open policy applied by the corporation has done more good than harm as it has benefited millions of developers as well as security architects seeking to safeguard the platform. There have been around 1.9 million instances of Android malware discovered in the first half of 2019, which means an infected application has been published every eight seconds. [5] With more utilities being developed for the Android platform, it is bound to continue being a target for malware.

Attackers mainly seek to access private information for individuals and entities which is used to orchestrate identity theft and hacking incidences. Attackers can obtain private information from users' private profiles such as online wallet access details, call logs and contact information which can all be used for malicious purposes. Due to the adverse consequences of Android malware and millions of potential victims, malware detection has been an ongoing security issue which has received significant interest from multiple stakeholders. Multiple detection and security measures have been proposed and developed with varying degrees of success and reliability. For instance, a typical detection technique is the signature-based warning mechanism which compares individual applications with known malware signatures

[6]. However, the method is limited with regards to detecting mobile malware, which keeps on emerging at a rapid rate, because a database may not contain their signature. This conventional technique has necessitated the invention of other methods such as static, dynamic, hybrid, and machine learning which are more efficient in detecting malware [7]. Even with robust detection techniques, malware architects still find ways to avoid detection, hence making the process even more complicated.

Security may be considered a dynamic target. The significant part of the society has recognised the tedious form of protection used on computers as a virtually unavoidable effect in contemporary times. However, compared to the personal computer world, mobile presents an emerging field. Currently, mobile gadgets constitute a critical part of the daily lives of people because they allow them to access various ubiquitous services [8]. The existence of such mobile and universal services has increased considerably because of different types of connectivity offered by mobile gadgets; for instance, Wi-Fi, Bluetooth, General Packet Radio Service, and Global System for Mobile Communications. Android also provides fully-developed features for exploiting cloud-computing resources [8].

The security of Android has emerged as an exciting research topic. Such research attempts have examined the security threats of Android from different viewpoints. They are spread across different research communities, thus resulting in a literature body that cuts across various publication venues and domains. A significant quantity of the reviewed literature is published within the software security and engineering domains. However, research about the security of Android also runs parallel with that for analysing programming language, mobile computing, and HCI which considers topics such as the usability of security approaches.

1.1 Research Questions

The thesis attempt to answer the following research questions.

- **RQ1:** How can we classify the work on Android application security analysis reported in the literature? Chapter 5 will endeavour to answer the first research question.

- **RQ2:** What is the current state of Android malware detection techniques and technologies? Chapter 5 will attempt to answer this research question.
- **RQ3:** What challenges, gaps, and patterns might be deduced from the existing research attempts, which will inform further research? Chapter 5 will attempt to answer this research question.

The third research question has helped identified further research questions to devise our research path.

- **RQ4:** Is it possible to devise an efficient process for running experiments, decompiling the APK, obtaining the required features, and viewing and comparing the results? Chapter 6 – Droid Fence Overview and Background on features – will attempt to answer fourth research question.
- **RQ5:** Will the use of neglected features identified in literature review along with Android permissions and services allow us to build an Android malware detection model capable of achieving over 90% accuracy whilst keeping the False Positive Rate (FPR) less than 3%? Chapter 7 – Droid Fence Methodology & Performance Evaluation – will attempt to answer research questions five.
- **RQ6:** How does the performance of the proposed deep learning algorithm (in terms of accuracy, F1 score, precision, and recall) compared to that of existing machine learning algorithms? Chapter 7 – Droid Fence Methodology & Performance Evaluation – will attempt to answer research questions six.
- **RQ7:** Does the approach developed as part of RQ5 performs better (in terms of Accuracy) than comparative methods? Chapter 7 – Droid Fence Methodology & Performance Evaluation – will attempt to answer research questions seven. RQ6 and RQ7 are different because the former compares the algorithms used in our

experiments whereas the latter compares our approach with the comparative methods in literature.

1.2 Contribution

The thesis makes the following contributions:

a. A systematic review of current approaches for analysis of Android malware, Android malware analysis, and custom-built malware detection technologies. The contribution is made as part of research into RQ1 and RQ2.
b. Presented a systematic literature review of research within the Android malware detection area through a suggested taxonomy. The contribution is made as part of research into RQ1 and RQ2.
c. Located gaps, patterns, and trends through comparative analyses and observations across Android malware security. The contribution is made as part of research into RQ3.
d. Developed an approach that amalgamates a set of permissions, services, and six other features (usage of https, database, dynamic code, native code, reflection, and cryptography) to generate a matrix that is used for detecting malware effectively. To the best of our knowledge, this is a novel approach that combines these features to detect malware. The contribution is made as part of research into RQ5, RQ6 and RQ7.
e. Developed Droid Fence, a web-based framework, which allows users to run experiments to extract various static features, store them in a relational database, and apply different machine learning and deep learning algorithms to detect malware and commit the result into a database. The contribution is made as part of research into RQ4.
f. Droid Fence provides a web frond-end to view, evaluate, and compare the results of nine algorithms stored against each experiment. The contribution is made as part of research into RQ4.

g. Our experiment demonstrates that deep learning Sequential algorithm scored consistently highly when compared against eight machine learning algorithms. However, the difference between the scores achieved by the Sequential and Random Forest Classifier is minimal in comparison with the remaining algorithms used in our experiments. We used a stratified k-fold cross-validation method, and the result was compared for four metrics: accuracy, F1 score, precision, and recall. The contribution is made as part of research into RQ5, RQ6 and RQ7
h. Offered suggestions for building research agendas for further developments. The contribution is made as part of research into RQ4, RQ5, RQ6 and RQ7.

1.3 The organisation of the thesis

The rest of the thesis is organised as follows. In Chapter 2 – Android Operating System – an overview of the Android operating system will be provided. Chapter 3 – Android Malware – will provide a brief history of Malware and their family, how malware have evolved over the year, how malware are monetised, and their attack and evasion techniques. Chapter 4 – Systematic Literature Review – will provide a description of the related surveys, our methodology, research questions, and the scope of our systematic literature review.

Chapter 5 – A Classification of Android Malware Security Analysis – will provide answers to our research questions through suggested taxonomy and the outcome of the literature review. The chapter will endeavour to answer the first research question, i.e., How do we classify Android application security analysis provided in the literature? The Chapter 5 will further attempt to answer the next two research questions: What is the current state of analysing Android malware detection techniques and technologies? And what challenges, gaps, and patterns might be deduced from the existing research attempts which have directed our research.

Chapter 6 – Droid Fence Overview and Background on features – will attempt to answer fourth research question (RQ4) and will provide an overview of our custom-built technology Droid

Fence and how it works. It will also offer a summary of the features that we are using in our experiment. Chapter 7 – Droid Fence Methodology & Performance Evaluation – will attempt to answer research questions five (RQ5), six (RQ6), and seven (RQ7) by providing detail on methodology, our experimental settings, data set collections, our results and the comparison with related methods. It will also provide a brief overview of the machine learning algorithms that are utilised in our experiments.

Chapter 8 – Conclusion and Future work – will present the conclusion of our thesis and the future direction in two domains: Android Malware Detection and enhancing Droid Fence.

2. Android Operating System

This section succinctly presents an overview of the Android operating system and its security mechanism. Android is an open-sourced mobile operating system developed and maintained by Google. Android is designed on top of a Linux kernel, and its source code is released under Apache license. Google acquired Android Inc. - the company that developed the Android operating system initially - in August 2005; this was a strategic step by Google to dive into the mobile market [9]. Google released Android in 2007 [10] paving the way for HTC to release the first commercially produced Android device called the ‘HTC Dream’ in Sep 2008 [11]. Android has since become a ubiquitous operating system with over two billion activated Android-powered devices and over two billion monthly active Android users [12]. Figure 1 below shows that Android has a dominant market share of smartphones sales to end users.

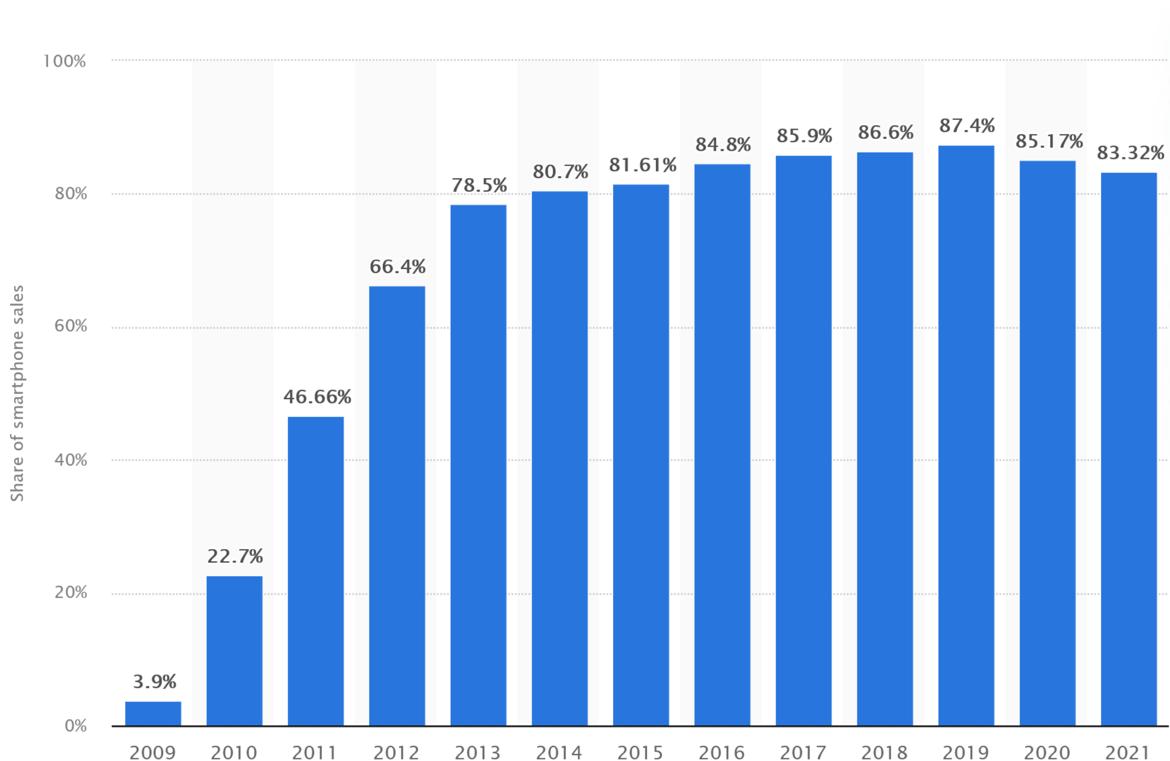

Figure 1: Android OS market share of smartphone sales to end users from 2009 to 2021 [13]

2.1 Android Components

The Android operating system consists of various components, and we will explain each element in subsequent subsections below.

2.1.1 Android Platform Architecture

Android refers to a platform intended for mobile gadgets; its architecture is classified into six layers, as shown in Figure 2 below. We will briefly examine all six layers in the following subsections [14].

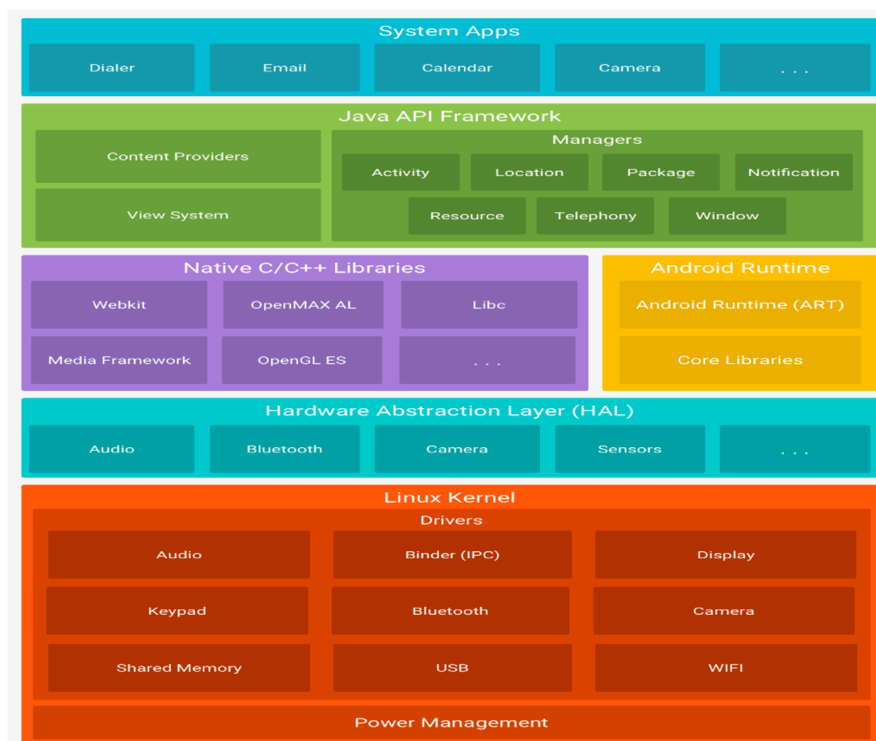

Figure 2: The Android Software Stack [14]

2.1.2 Android Applications

System applications, developed by the Android team, and all third-party applications are installed in the topmost layer of the Android software stack. Android contains a set of core applications which allows users to perform some basic operations such as SMS messaging, calendars, contacts, making or attending phone calls and more. System applications do not have any precedence over third-party applications if user would like to use a different

application for a given purpose instead of using the default system one. Android provides facility to use individual applications for a basic functionality offered by a system application, e.g., any third-party application can become a default application for sending and receiving SMS messages.

2.1.3 Java API Framework

The Java programming language is mainly used for writing Android applications through a broad range of application programming interfaces (API) that the Android software development kit (SDK) provides. This layer consists of modular and reusable core components and services which are used to create Android applications. These reusable components include *View System* for building application's user interface by using UI components such as list boxes, grids, buttons etc., *Resource Manager* for supplying access to static non-code assets such as graphics, localised date-time or string variables, *Notification Manager* for allowing applications to present alerts to users, *Activity Manager* for controlling the life cycle of applications and supplying navigation, and finally, *Content Provider* for enabling applications to share data across applications [14].

2.1.4 Native C/C++ Libraries

The native C/C++ layer is in charge of supplying support to many core Android features such as ART and HAL. These fundamental features are built from native code, and they require access to native C/C++ libraries to work seamlessly. The Android platform ensures this access via Java framework APIs. For instance, Java OpenGL API can be used to access OpenGL ES for adding drawing and graphics manipulation support in an application [14]. Third-party application developers requiring access to C/C++ code can utilise the Android Native Development Kit (NDK) for accessing native libraries.

2.1.5 Android Runtime

Android Runtime (ART) is the managed runtime environment utilised by some Android system services and third-party applications. The Dalvik virtual machine (Dalvik VM) was the Android

runtime environment preceding Android version 5.0, API level 21 [15]. Applications running on Android version 5.0, API level 21 or later utilise ART for faster startup and on-going execution, as each application runs in its process, which in turn has its instance of the Android Runtime. ART pre-compiles the bytecode into native code, at installation time, by using a method called Ahead of time (AOT), this removes the need to execute applications in interpreted code, resulting in faster execution. ART also has an optimised garbage collector (GC) as well as enhanced debugging support [15].

2.1.6 Hardware Abstraction Layer

The hardware abstraction layer (HAL) provides functionalities for application developers to access the device's hardware capabilities. Applications can leverage Java API framework to use multiple library modules, which implements an interface for different hardware components, exposed by HAL. This layer also provides developers to build their drivers.

2.1.7 Linux Kernel

The Linux Kernel is the lowest level layer in the Android software stack and is the foundation of the Android platform architecture. The Kernel is responsible for low-level operating system tasks such as memory management, multi-threading, network stack, process isolation, storage management security management and more.

2.2 Inter-Component Communication

As one of its security mechanisms, Android protects applications from one another and system resources against applications through a sandboxing mechanism. Such insulation of applications on which Android relies for protection of applications needs interactions to take place via a message-passing tool, known as inter-component communication (ICC). In Android, ICC is mainly undertaken through intent-filters, which represent the types of requests to which a specific component may respond. An intent message refers to an event that should be available for actions to be done alongside the data supporting those actions. Component invocations exist in a variety of flavours, such as inter-app, intra-app, implicit, or explicit. The

ICC of Android allows for late runtime binding involving components within different or similar applications, where calls are not explicit within the code rather than enabled via event messaging, a significant characteristic for event-oriented systems. It has emerged that the ICC interaction mechanism for Android introduces numerous security issues [16]. For instance, intent-event messages that interacted in components may be tampered with or intercepted because there is no typical authentication or encryption applied to them [17]. Additionally, there is no mechanism for blocking an ICC callee against misrepresenting its caller's intentions toward third parties [18].

2.3 Application Configuration

The manifest refers to a necessary configuration file (AndroidManifest.xml), which accompanies all Android applications. It specifies many things, including the principal components that make up the application; this includes their abilities and types, along with enforced and required permissions. The values of the manifest file are attached to the Android application during the time of compilation, and modification cannot be undertaken during run-time.

Apart from sandboxing, permission enforcement is another mechanism that the Android framework provides for the protection of applications. Indeed, permissions constitute the Android security model's strong point. The permissions defined within the application manifest allow secure access to sensitive resources and cross-app interactions. When users install applications, the Android system asks the user's agreement to requested permissions before installation. If the user declines to provide the application the required consent, the installation of the application is cancelled. Further, with in-built permissions that the Android system provides to shield different resources of the system, any Android application may also describe its permissions for self-protection.

Because the access control model of Android is at the individual applications level, there is no mechanism for checking the security state of the seemingly benign applications colluding with

each other. This type of access control model triggers numerous security challenges that include application collusions [19] and re-delegation attacks [20], which have been found to exist within the applications in the market [17] [21]. Application collusion attacks occur when malware developers split malicious code among two or more applications and initiate attack when a user installs all those applications that contain part of the malicious code. Intra applications communication feature of Android is used to communicate among the affected applications. Re-delegation attack is a form of application collusion where an application with fewer privileges colludes with an application that has higher rights to perform unauthorised operations [20].

2.4 Android Security Model

The Android security model is designed to respond to specific security needs of the platform. The Android security model can be divided into 5 domains each defining various security rules on the platform [22]. The first domain involves multiparty consent. In this regard, the android platform can be divided into the user, developer, and platform. Users control the data located in shared storage, while developers control the information contained in application folders. The platform controls data in special locations that are only reserved for the operating system. Each party must consent to provide the data whenever it is requested by a certain program and can revoke the privilege anytime. The rule ensures that all the stakeholders are aware of their data being used elsewhere.

The second domain is the android platform being an open ecosystem platform. It enables the developers to have power over the data they are willing to share with other programs thereby enhancing security. Further, mobile devices are required to be security compatible. Mobile phones must pass Google's Compatibility Test Suite in order for them to be compatible with Android operating system [22]. Manufacturers must adhere to various recommendations that help to guarantee the security of the devices when using the Android operating system.

The third domain involves the application programs acting as security controllers. In the traditional desktop, running a program with administrative user privileges ensures that it has total access to the resources of the system. The same case does not apply to the Android platform and applications are not considered to be able to fully authorise user actions. Such a design creates a sandbox environment for the applications to execute without an impact on others or the system settings and application. An application that needs data from other areas uses multiparty consent by requiring permission from the owner of the data such as the platform, application program, or from the user.

The fourth domain is the recommended platform, Google Play, to download applications. The platform is a service provided by Google that ensures that the company has control over the programs installed on user devices [23]. The control is mainly aimed to ensure the data security of users by ensuring that mobile applications fulfil various security related requirements. Furthermore, since Google Play is installed on the devices of Android users it acts as an anti-virus by checking applications for harmful code during downloads just before it is installed on the device. Additionally, Google Play scans an android phone periodically to ensure that application updates are not malware [23]. When applications with malware are detected during download, a user is usually warned about the issue to make them aware of the problem. The application also continues to be available on the Google Play store until it is removed after extensive review of the problems detected by play protect.

The last dimension is the fail-safe that restores a device to its factory settings, which are usually safe [22]. When a mobile phone becomes compromised by persistent malware, a user can reset the mobile phone to a safe state, which involves formatting the writable parts and returning it to a state that uses only verified system code. Thus, the Android security model consists of various distinct dimensions that help to guarantee the overall security of a device running the operating system.

3. Android Malware

In this section, a brief history of malware and its evolution are discussed. Our primary focus will be Android malware, but inevitably we will provide a brief overview of mobile malware in general. We will also discuss the attacks and evasion tactics that malware use to avoid detection.

3.1 The Evolution of Malware

The mobile malware have come a long way since the first mobile worm, Cabir, designed to infect Nokia 60 series was developed. The infection seems innocuous as the worm would display the word ‘Caribe’ on the screen. The infection would spread itself, using Bluetooth, to other nearby Bluetooth enabled devices such as printers, mobiles, etc. [24]. Symbian was a popular operating system at the time and would offer a vast market for mobile malware developers. Incidentally, 2005 also saw a sizable dip in Symbian market share, which could be the result of Cabir propagating in Symbian mobile phones [25]. In 2005, Cabir was followed up by CommWarrior which would proliferate itself using MMS as well as Bluetooth. The virus was designed for Symbian 60 platform and had infected over 100,000 mobile devices by sending greater than 450,000 MMS [24]. This propagation added monetary value to malware development as each MMS sent would incur a charge from a carrier. The financial incentive was further exploited by a trojan called RedBrowser, which was discovered in 2006. The trojan was designed to utilise premium rate SMS service as each SMS would typically cost $5 to the device's owner [24]. RedBrowser was a turning point in the evolution of mobile malware as it was the first malware that could contaminate mobile phones encompassing different operating systems by utilising weakness in universally supported Java 2 Micro Edition (J2ME) [24]. The next two years, 2007 and 2008, were idle periods regarding the evolution of new mobile malware threats and development of non-commercial malware almost died out [26]. An example of the non-commercial malware is the first mobile worm Cabir [24] which was not developed for monitory gains as discussed at the start of the chaprter. However, there was a

significant increment of exploiting premium rate services by several mobile malware [24]. The primary purpose of developing malware was to steal data as cybercriminals started targeting online gamers to acquire their passwords, in-game assets and virtual characters [26].

In 2009, the mobile botnet malware Yxes was discovered and hit the headlines as one of the first malware for Symbian OS 9 [27]. This discovery was another turning point in the evolution of mobile malware as well as technological innovation because it was the first malware to send an SMS and access the Internet [24]. Mobile malware has been rising exponentially since 2009 due to technological enhancements offering new approaches for profit utilisation, an increase in black market accessibility for selling and buying stolen information, and malware developers collaborating on exchanging malware code and system vulnerabilities [25]. Apvrille et al. [24] describe 2010 as an industrial age for mobile malware as it transitions from individuals to organised cybercriminals as monetary incentives grew significantly. The first quarter of the same year also witnessed an increase in the popularity of Android. Its worldwide market share increased to 88% from 1.6% in Q1 2009. Figure 3 below shows this popularity trend.

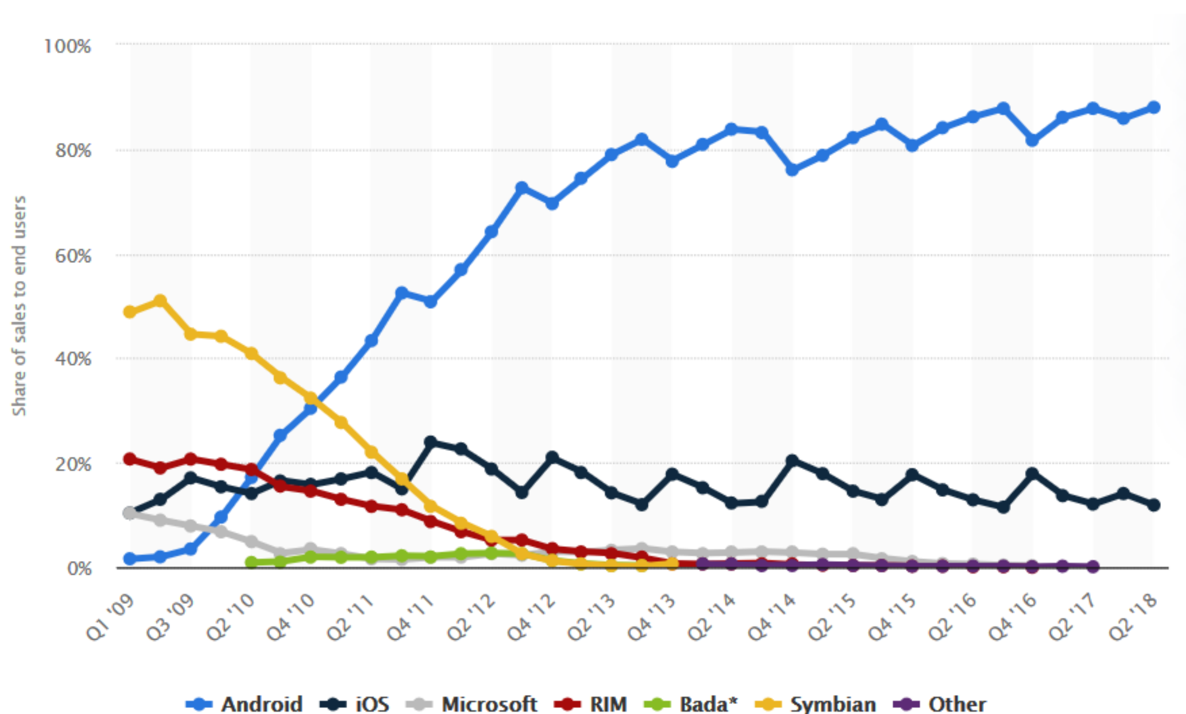

Figure 3: Global market share [28]

The increased use of Android platforms has rendered the platform vulnerable to new and sophisticated malware. The malware are designed to evade detection with devastating effects on victims. Malware authors cause losses that run into billions of dollars annually. [29] As a result, this makes it a lucrative venture as Android system users continue to increment. It is estimated by Amro [30] that a new strain of malware is detected every 10 seconds. This rate is not only alarming but requires immense resources to identify and neutralise. The percentage increase in evasive malware jumped to more than 2000% in 2017, and the trend is expected to continue [31]. Massive attacks are being launched with thousands of users falling victims and experiencing losses. Malware authors have used crafty techniques to overcome authentication requirements put in as security measures. Some malware are attacking all authentication devices hence exposing authentication vulnerabilities. [32]

Alarms have been raised primarily due to increased ransomware cases where malware authors hold Android systems to ransom until they are paid off. The malware authors would mainly use untraceable cryptocurrencies to cover their tracks [32]. Apart from the common malware targets of locking device screens, ransomware have targeted supporting other malicious intentions such as wiping out data, resetting security settings, GPS tracking, and theft of personal information. However, even with increased malware threats, antimalware companies and Google have been investing resources towards developing detection techniques progressively [33]. Critical databases have been developed over time and have played a considerable role in enhancing Android security for millions of users; however, it can be deduced that malware authors seem to be a step ahead all the time [33].

3.2 Malware Family

The threat posed by Android Malware has provided a need for researchers across the globe to collect malware samples, identify malware families, and suggest methods for countering this threat. The need for a collection of Android malware became apparent after the detection of the first Android malware named ‘fake player' in 2010 [34]. Collection of malware samples is deemed essential for systematic examination to create robust detection platforms and

techniques. Collection of samples is carried out whenever new malware is reported and is updated in existing databases such as those held by antivirus companies [4].

There are thousands of Android malware instances which keep on increasing based on their malicious intentions and engineering processes. Based on the capabilities of such malware, they are classified into different families. The number of malicious applications has risen steadily since 2010, and the trend is expected to continue [35]. In 2013, the number was estimated at 500,000, which rose to more than 2.5 million applications in 2015. The number increased to 3.5 million in 2017 and is expected to continue growing with more sophisticated and highly destructive malware expected [36]. Ransomware is one of the most reported forms of malware and accounted for close to 30% of cases in 2017. In 2017, the Rootnik malware family accounted for 42% of all cases, while PornClk family accounted for 14%. Other families such as Axent, Dloadr accounted for less than 10% of reported cases each. [36]

Malware in the same malware family has similar traits and form of action or malicious activity. They may also have different variations which have different detection evasion patterns meant to avoid different detection techniques. For example, the KungFu family is one of the most harmful families of malware with different variations such as KungFu1, KungFuD, and KungFu3, among others [37]. Malware in the family is similar in the sense that they are packaged and downloaded from third-party platforms in the open markets. The malware has privilege root exploits which do not require the input of the user and transmits sensitive information such as IMEI number, phone number and contacts, Android model and versions. Subsequent versions of the malware have more detection evasion capabilities and more harmful features. [38]

Most malware families are already defined and detected. This identification has seen a reduction in the number of new malware families reported annually. However, variations of malware within the families continue to increase. For instance, 2012 reported more than 100 new families of Android malware when compared to only four new families in 2016, even though variations of malware within the families increased. 2014 reported 46 new families

while 2015 reported 18 new families [38]. Some of these malware have found their way to the Google Play store, while more of them are from third-party markets [38]. Malware families have similar malicious behaviour, which makes categorization easy. For instance, some malware families such as Grabos are designed to trick users into downloading other malicious applications which may facilitate malicious activities such as sharing of sensitive and personal information [39]. Such applications may be disguised in the form of other utility applications such as free music downloader, hence attracting huge followings. This disguise exposes users to downloading other harmful applications from unknown sites and also opening them without consent [25]. Other families operating and using this kind of approach include the TrojanDropper.Agent.BKY and AsiaHitGroup. Table 1 below displays a list of malware families and their descriptions.

Table 1: List of malware families and their descriptions [25] [34] [36] [37] [38] [39]

Malware family/Name	**Description/ how it works**
AnserverBot	Infected host application displays a new dialogue to request, and upgrade a new application. The downloaded application contains a hidden payload. The payload communicates with a remote server for receiving commands.
BaseBridge (AdSMS)	When Infected host application is installed, it will trick user into upgrading the application. The downloaded application is installed and would communicate with a remote server for downloading configuration file. The file contains the premium numbers where the malware will call or send SMS messages
BeanBot	The malware sends device data to a remote server. It also sends premium SMS messages.
Pjapps	The malware communicates with a remote server and sends Browse’s history and bookmarks. It also sends premium rated SMS.
BGSERV	The malware sends premium rated SMS. It also communicates with a remote server for sending private information.
CruseWin (CruseWind)	The malware sends premium rated SMS. It can also upgrade itself.
DroidCoupon	It uses a simplified root exploit—" Rage against the Cage" in Android 2.2 and earlier, hide Platform, so it is challenging to detect it. The malware leaks users private information.
DroidDeluxe	Install a password recovery tool, and It will not work on Android 2.3, with a message: "This application has stopped.”

DroidDream (DORDRAE)	It disguises itself as applications such as a battery-monitoring tool, a task-listing tool. The malware sends device's information to a remote server.
DreamLight	Service named "CoreService" is started when infected device receives a phone call. The malware sends device's information to a remote server.
DroidKungFu (LeNa)	The malware roots the device. The malware sends device's information to a remote server.
Smssend(fake player)	The malware sends premium rated SMS.
Gamblersms	The malware asks user to provide a phone number and an email address. This information is then abused.
Geinimi	The malware extract information from the device and sends it to a remote location. A hacker can execute commands remotely, such as to send SMS and make phone calls.
GGTracker	The malware tricks user to download battery saver application via malicious advertisement. Once installed, it will send premium text messages.
GoldDream	The malware has bot capability as it communicates with a remote server for fetching commands or sending private information. It intercepts incoming and outgoing SMS messasges and forwards them to the remote server.
GPSSMSSpy (mobinauten, SmsHowU, SMS spy)	The malware sends premium rate SMS messages.
Jifake	The malware is installed when user opens the link to download apk from a malicious website, but users do not see any file being downloaded. The malware sends premium rate SMS messages.
Plankton	The malware is installed with an infected host application. It harvests user's data and sends it to a central server.

3.3 Malware Attacks and Evasions

Malicious Android applications are developed with the evil intention that can lead to severe losses. In this regard, developers are always looking for evasion techniques to avoid detection and to carry out malicious plans [40]. Evasion techniques used are becoming more complex and sophisticated and spread across multiple families of malware. There are three most common evasion techniques that we have come across in existing literature: *Anti-security techniques* avoid detection by anti-malware software by hiding behind an existing process of some legitimate applications already running on a device, *Anti-sandbox techniques* detect automatic analysis and as a result, avoid executing a malicious behaviour, and *Anti-analyst*

technique uses monitoring tools such as obfuscation to prevent reverse engineering. These evasion tactics have been referred to extensively in existing research: Figure 4 presents break down of each of these techniques against several papers which reference them.

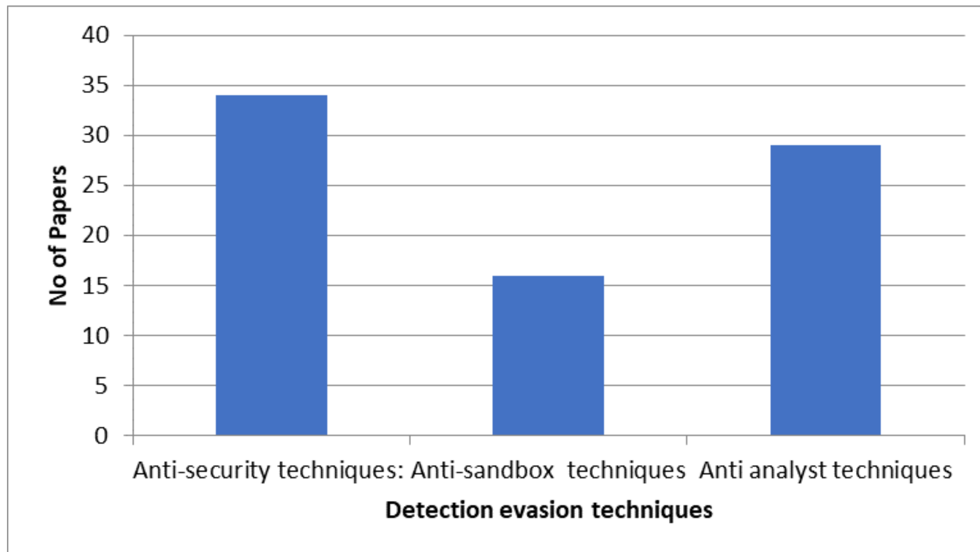

Figure 4: Detection of evasion techniques by several papers

A short description of additional evasion techniques is mentioned below:

Packing: even though packed malicious applications have been present for long, recent trends have shown more sophisticated packing techniques which are challenging to detect. Packing allows applications to compress files, to reduce the size of applications, unpacks and executes these files in memory at run time. It is a legitimate technique, but it also allows malicious applications to compressed malicious code in encrypted classes.dex files and unpacks them at runtime for malicious purposes [37].

Muitidex Applications: standard malware detection focuses on Dalvik Executable (DEX) file in the Android system. Applications typically have one DEX file which contains executable code. Some Android malware are designed in a manner that the payload splits between two DEX files. This technique can be used as a simple way of evasion to avoid detection; this is especially true for static detection methods that analyse one DEX file and misses the second DEX file, which may contain part of the malicious code [41].

Instant run-based malware: Instant-run is an Android feature that enables developers to release updates more quickly through a debug application which pushes a zip file into an application for updating purposes [42]. Malware authors are disguising malware payload into these zip files used by the instant-run feature to evade detection [43]. The approach is one available for Android Lollipop and later versions and can only be used on applications installed through side-loading.

Malformed manifest files: this evasion technique uses strange values in the AndroidManifest.xml file and the resource file. The method can confuse static detectors and avoid detection.

Chattr: Chattr, a Linux command, is being used to lock malware on Android systems, mainly where root privileges are acquired. The chattr utility can be packed into an app in an encrypted format and can be used to lock the app into the system folder. Once an app is locked, any attempts to remove it even with root privileges fails.

Following table displays papers where each of the above evasion techniques has been discussed.

Table 2: List of papers addressing each detection evasion technique

Evasion Techniques	**Papers**	**No of Papers**
Anti-security techniques	[40] [37] [41] [42] [44] [45] [46] [4] [47] [48] [49] [50] [51] [52] [53] [31] [54] [55] [56] [34] [4] [35] [36] [37] [38] [57] [25] [58] [59] [60] [61] [62] [51] [63]	34
Anti-sandbox techniques	[43] [64] [65] [66] [67] [68] [69] [63] [70] [2] [71] [36] [65] [72] [73]	16
Anti analyst techniques	[29] [43] [64] [65] [66] [67] [68] [69] [63] [70] [2] [71] [36] [65] [72] [73] [25] [58] [59] [60] [61] [74] [75] [76] [62] [51] [65] [49]	29
Packing	[36] [65] [72] [73] [25] [58] [59] [60] [61] [74] [75] [76]	12
Muitidex Applications	[38] [57] [25] [58] [59] [60] [61] [62] [77]	9

Instant run-based malware	[55] [56] [34] [4] [35] [36] [37] [38] [57] [25] [58] [59] [60] [61] [62] [51] [49] [73]	20
Malformed manifest files	[49] [73] [52] [53] [31] [63] [54] [55] [56] [34] [4] [35] [36]	13
Chattr	[2] [71] [36] [65] [72] [73] [29] [30] [31] [32] [33]	11

In addition to evasion techniques, there are malware attacks such as evasion attacks, gradient descent attacks, and tree-ensemble attacks, poisoning attacks, classifiers and clustering attacks.

Evasion attacks: this is a form of attack where malware may use benign application behaviour such as injecting API calls relevant to benign applications to manipulate antimalware software algorithms [78]. Evasion attacks are designed to avoid detection while releasing their payload on Android systems. Most malware use evasion techniques in a bid to counter exposure by existing antimalware systems [2]. Successful evasion attacks can be significantly expensive and harmful due to delayed or failed detection.

Poisoning attacks: these are evasion techniques that target machine learning models for malware detection. The models are attacked at the training stage, hence jeopardising their capabilities to detect individual malware families [65]. The attack means that the model is trained on how to evade certain types of malware by injecting additional seeds or requesting additional permissions to misled the algorithm [65].

Classifier attack: This type of attack targets detection techniques where malware are classified incorrectly. The error in the detection means that suspicious application may pass for good ones while genuine ones may be flagged for malicious [72] [73]. An attacker with sufficient information belonging to a training set can temper classifier by choosing intrusion points that may confuse the detection model in a bid to pass malicious code without detection [79].

3.4 Malware Monetisation Strategies

There are a variety of approaches that malware attackers use for monetisation purposes. One of the strategies that such users have found they can exploit the victims is by using malware that performs tasks that provide financial rewards [80]. However, the attackers ensure that they receive the reward instead of the victim, which creates an incentive for engaging in such activities. There are various approaches that attackers use to monetize malware and ensure that their activities produce a financial gain for them. For example, some malware is designed to automatically click on specific advertisements that result in the developer being paid [81]. In this regard, the malware covertly opens specific web pages and begins automatically clicking on advertisements on the website, which results in the developer being paid through the google ads program. A similar approach involves the automatic subscription to targeted YouTube channels. Once subscribed, the malware also proceeds to play some of the videos so that ad revenue is paid to attacker. Some malware also help the developers to perpetrate financial fraud. For example, some applications can automatically pay for orders that will benefit the criminals at the expense of the victim. In addition, the user credentials can be stolen, and the attackers can use them to access financial accounts and steal funds from victims or use them to make personal purchases. Additionally, some malware is used to lock victims out of their devices and the attackers use the incident to demand ransom from the victims in exchange for access [82]. As such, there are a variety of approaches that attackers use to monetize malware on android platforms.

4. Systematic Literature Review

In this chapter, we will describe the criteria for our systematic literature review. we will present related work to show literature surveys that have been submitted by different researchers over the years. We will also define our research method, research questions, search strategy and scope of our inclusion and exclusion criteria.

4.1 Related Work

Previous related surveys may be categorized into two groups: mobile malware studies and Android security studies. This review searched survey papers in the two research domains. Examining, classifying, and identifying mobile malware have emerged as an exciting research field because mobile platforms emerged. Before the emergence of current mobile platforms, for instance, Android and iOS, Dagon, Martin and Starner [83] presented a mobile malware taxonomy. Although threat models for old mobile gadgets were described, such as PDAs, this dissertation draws on some aspects from the study, particularly regarding the security taxonomy of Android.

Felt et al. [20] assessed the malware behaviour spread over Symbian, Android, and iOS platforms. Additionally, they analysed the efficiency of techniques that official application markets, such as Google Play store, and Apple Appstore, apply to identify and prevent such malware. Along similar lines, a thorough survey regarding the malware evolution for smart gadgets is offered by Suarez-Tangil et al. [84] in 2014, indicating a specific rise in malware targeting mobile devices since 2010. Additionally, the paper analyses 20 research attempts that investigate and identify mobile malware. The studies fail to assess features of the techniques for investigating and detecting malware, nor the approaches for exposing the vulnerability of Android. Apart from the general, platform-independent malware surveys, there are several relevant surveys, which describe Android security subareas, mainly linked to certain kinds of security challenges within the Android platform. For example, Chin et al. [16]

investigated security issues within the Android inter-app communication and presented numerous categories of possible attacks on applications.

Another study is a comprehensive survey taken by Sadeghi et al. [85] that offers a review of the existing approaches for analysis of Android security. The study presents an Android security analysis taxonomy, derived from existing literature, against multiple domains. However, the study has omitted a comprehensive analysis of existing custom-built detection frameworks and the evolution of Android malware.

Another example is the survey undertaken by Shabtai et al. [86] that offers a thorough analysis of the security measures that the Android framework provides; however, it does not comprehensively examine other research attempts for detecting and mitigating security challenges within the Android platform. The study undertaken by Zhou et al. [4] discusses and classifies 1,260 instances of Android malware. The set of malware known as Malware Genome is then utilized by researchers over the years to assess their suggested malware identification techniques. Each of the surveys summarizes specific domains; for instance, assessment of Android's inter-app threats or Android malware families. However, they did not offer a thorough overview of current research regarding the analysis of Android security.

4.2 Research Method

The review follows the guidelines of the systematic literature review (SLR) process suggested by Kitchenham [87]. Additionally, the review has considered Brereton's [88] lessons on using SLR within the domain of software engineering. The process consists of three major stages: reporting, conducting, and planning the review. According to the guidelines, the research questions below are formulated to direct the systematic literature review.

- **RQ1:** How do we classify Android application security analysis provided in the literature?

- **RQ2:** What is the current state of analysing Android malware detection techniques and technologies?
- **RQ3:** What challenges, gaps, and patterns might be deduced from the existing research attempts, which will inform further research?

The first research question focuses on research on Android applications and how application analysis can be classified with regards to security. The first question is linked to the second research question, which aims at examining the status of malware detection and Android protection as depicted by different players in the industry, such as developers and software engineers. With continued interest in the Android platform, subsequent versions have been released from time to time with improvements and correction of previous weaknesses. This version increment has been associated with the continuous development of malware detection techniques which include static, dynamic, and hybrid. Detection and security frameworks such as Androguard and DroidOlytics, for example, are also critically analysed with their capabilities and limitations. Finally, the third research question focuses on gaps identified in Android malware detection and security and the several recommendations for moving forward.

4.2.1 Search Strategy

The search strategy was guided by search terms that are relevant and related to the subject topic. The keywords used for the study were picked from the main issue "Android malware detection and Android security". In this regard, the significant search terms were Android Malware detection and Android security. The keywords were used to locate papers which were then subjected to an inclusion and exclusion criteria from the selected databases with relevant articles. In addition to the Keywords used for guiding the primary search strategy, additional phrases emanating from the framework of the literature review as depicted by the research question were also used for the strategy, inclusion and exclusion criteria. For instance, Android malware detection techniques, which yielded terms such as Androguard, MIGDroid, and Dendroid, among other relevant words, were considered in the search strategy. Other phrases

that were used for the search strategy are linked to Android detection techniques which include static, dynamic, hybrid and machine learning. Due to the progressive nature of the subject topic, the search criteria were flexible initially; therefore, the current research being carried out in and most recent publications to that effect. After conducting an iterative search for the main topic, any additional phrases identified from the result and based on the scope of the survey, a definite list of keywords was created. The list contains following terms for conducting the review: “Android dynamic analysis”, “Android static analysis”, “Android vulnerability”, “Android malware prevention”, “Android intruder detection”, “Android intruder prevention”, “Android antimalware”, “Android Malware attacks”, “Android malware detection framework”, and “Android malware prevention framework”. An AND operator was used for each word in the quotation marks. Whereas an OR operator was used for each search term separated by comma e.g.

Android AND Malware AND Detection

OR

Android AND Malware AND Prevention

4.2.2 Digital Libraries

Reputable databases and search engines within the review protocol were used to identify top-quality referenced research articles from dependable places. The following databases were found to have the relevant publication in the area of interest

1. Science Direct
2. IEEE Xplore
3. ACM Digital Library
4. Wiley Library
5. Springer library

6. USENIX Proceedings
7. Google Scholar
8. University of Strathclyde’s Digital Library

4.2.3 Inclusion and Exclusion Criteria

The inclusion and exclusion criteria were meant to identify papers that were mostly in line with the research questions and the objectives of the literature review. If a paper meets inclusion criteria, then it is included in our study.

4.2.3.1 Inclusion Criteria

1. The articles must have been published in the last eleven years, i.e., between 2008 and 2019. Earlier publications from 2019 have summaries of previous millstones and research in Android security, hence providing a solid background. 2008 also marks the period when the Android platform was commercially launched.
2. They must be full versions of journals and articles published from conferences and discuss Android malware detection and Android security.
3. The papers must also identify at least one of the following items: specific problems, gaps, vulnerabilities and propose solutions.
4. Papers must be written in English.

Notably, some retrieved papers from the chosen keywords did not fit our literature review’s scope. As Figure 5 illustrates, the range for the surveyed research in the current study falls at the convergence of three dimensions:

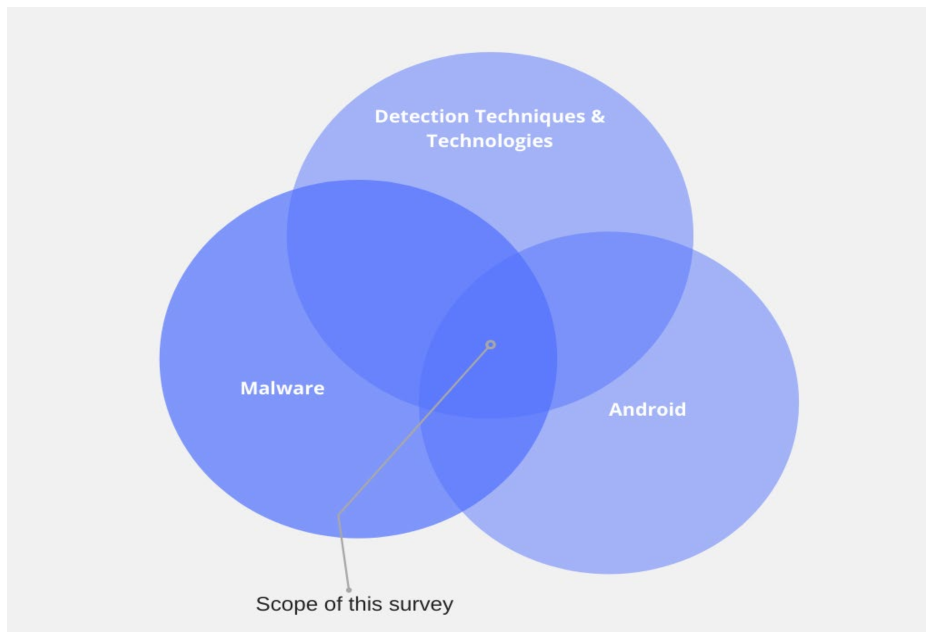

Figure 5: Scope of the Survey

4.2.3.2 Exclusion Criteria

The exclusion criteria aimed at eliminating papers or publications that did not meet the quality or scope threshold of the literature review. With thousands of articles published within the period of interest, coming up with 100 plus relevant ones required robust criteria. The exclusion criteria ensured that multiple publications that did not meet the requirements were eliminated and not used for the review.

1. Summaries of conferences and workshops, editorials, and abstract due to their limited nature regarding information.
2. Articles related to workshops during the early stages and are not yet published.
3. Posters with limited information on the subject topic.
4. Books and academic dissertations due to the scope of the literature review.
5. Paper mentioning Android malware detection and Android security but do not mainly focus on these aspects.

6. Papers lacking accessibility with regards to full-text versions i.e. unable to download the full paper.
7. Appear written in other languages apart from English.
8. Opinionated papers with broad discussions that do not fall within our research scope

The initial search returned 1067 papers; the breakdown for each search term is displayed in Table 3 below.

Table 3: The breakdown of search terms and their results

Search Term	**No of Papers**
"Android Malware Hybrid Analysis"	4
"Android Malware Prevention"	8
"Android Antimalware"	21
"Android Malware Dynamic Analysis"	32
"Android Malware Static Analysis"	74
"Android Malware Attacks"	107
"Android Dynamic Analysis"	169
"Android Malware Detection Framework"	179
"Android Static Analysis"	228
"Android Vulnerability"	245
Total	1067

There were 195 duplicate papers, so 872 were left for further screening. In the screening phase, the abstracts of these papers were read to remove obvious candidates that do not conform to our research scope. The final count of documents which followed our research scope and were included in our literature review was 142.

Figure 6 below illustrates the quantities of selected papers based on the year of publication.

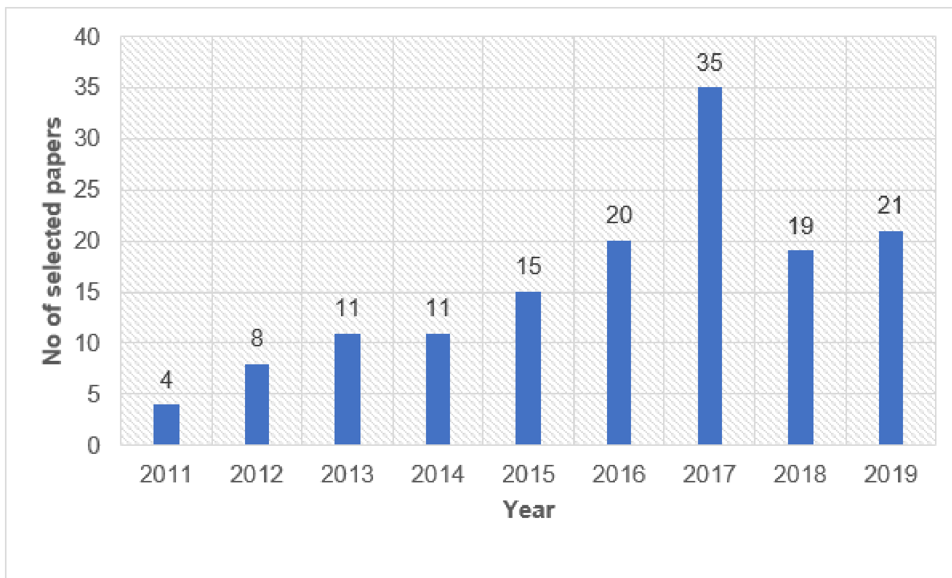

Figure 6: Number of selected papers by publishing year

4.3 Summary

In this chapter, we have presented a summary of existing literature surveys related to our work. Furthermore, we have established the parameters for our systematic literature review – broken down by research method, research questions, search strategy, and scope of our literature review by defining an inclusion and exclusion criteria for existing research papers. We have also shown the search result of each of our search terms.

The next chapter, Chapter 5 – A Classification of Android Malware Security Analysis – will provide answers to our research questions through a suggested taxonomy and the outcome of the literature review. Chapter 5 will endeavour to answer three research question, i.e., How do we classify Android application security analysis provided in the literature? What is the current state of analysing Android malware detection techniques and technologies? And what challenges, gaps, and patterns might be deduced from the existing research attempts which have directed our research.

5. A Classification of Android Malware Security Analysis

This section endeavours to answer the first research questions, i.e., How do we classify Android application security analysis provided in the literature? The Android operating system is a mobile platform which runs on multiple devices ranging from mobile devices, smartphones, set-top boxes, among others. With security being a significant issue of concerns for the platform, the use of third-party applications and cloud-based systems increases the risk of malware and security breaches [60]. Developers are therefore keen on having a system which can counter malware which keeps on getting more complex and difficult to detect.

Malware is used to target different types of applications in the Android system. Applications include entertainment applications, society tools, productivity applications, communication tools and puzzles, among others. Current research on Android application security has evolved significantly with new techniques being used for categorisation. Android application security analysis is categorised depending on the nature of the malware and previous experiences with similar patterns. The current automated Android malware detection and classification techniques are broadly divided into three categories, as shown in Figure 7 below:

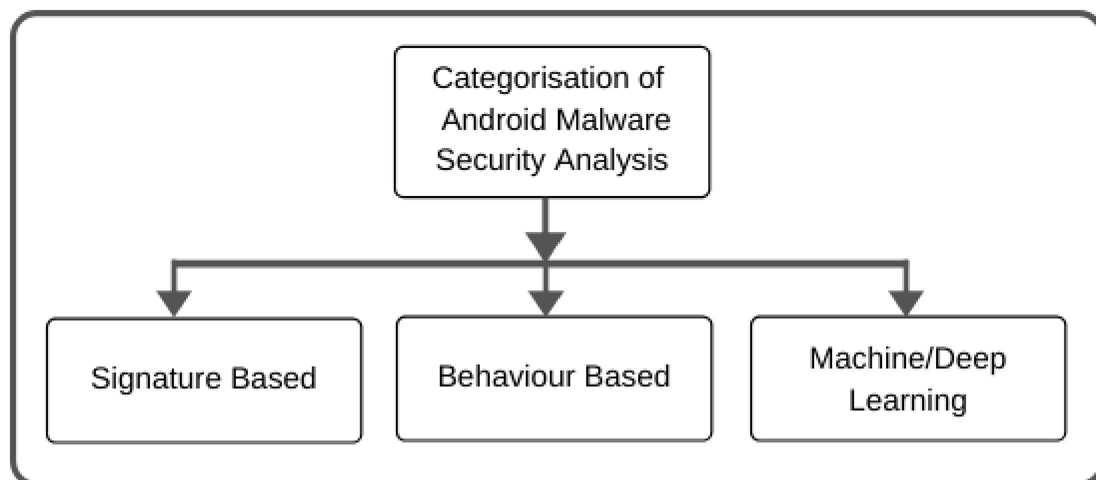

Figure 7: Categorisation of Android malware security analysis

The first category is the signature-based approach that seeks to identify specific patterns in bytecodes and API calls in malware detection [39]. The second category is the behaviour-

based approach that seeks to detect malware by observing the actions of an application. Both detectors are subject to evasion and may require complementary detection techniques to counter, i.e. the third category. The third category is the machine learning-based approaches which perform binary classification based on application behaviours and extracted features. In the following section, we will discuss each of these categories in more detail.

5.1 Signature-Based Detection

One of the primary methods for detecting malware is the signature-based approach which is used by most of the antimalware products sold commercially [89]. It extracts semantic patterns from applications, and a unique signature is created. This approach operates under the simple model that malware exists when an application code matches the malware signatures [90]. Signature-based methods are not very useful in detecting unknown malware because a database may not contain their signature [47]. Therefore, if a database does not have a specific signature for new malware, then it fails to be detected.

Signature malware classification uses multi-label classifiers that are designed to identify malware based on different classes that are known. Similar patterns help in flagging up unknown malware with similar characteristics such as API calls, package names, class names, string variables, and call graphs etc. The technique uses a malware graph database against which new threats are analysed – a graph database stores data in a graphical format as graphs. Vertices and edges create a graph whereby vertices are connected by edges. This representation is useful for finding a relationship by traversing over many edges. Static analysis of malware is conducted with high similarity graphs being eliminated. There is also a challenge in the process of identifying repackaged applications that are mutated from millions of applications that are supposed to be genuine and threat free. Malware developers can quickly engineer malware into legitimate applications through obfuscated program segments that have a structure similar to legitimate applications but with malicious logic in them [58]. Another challenge is the association of malware with existing applications to enable security analysis. Unless malware patterns are already known and existing, it means that existing

databases must wait for the release of malware in the wild or possible attacks for them to be included in the analytics databases. Conventionally a cryptographic hash of malware was found to be ineffective as a hacker could easily change hash values and package names in an application to avoid detection.

Signature-based malware detection has evolved significantly where approaches such as DroidAnalytics are being used for screening [59]. DroidAnalytics [61] is a signature-based static analysis tool to manage and analyse Android malware at the opcode level. The automated system is programmed to collect, analyse, and administer malware and create a mechanism of detection by creating a signature from the collected information such as package names, class numbers, API call number etc. Signatures of known and identified malware, based on this collected information, are created and then used to determine skewed patterns which alert to the possibility of new malware. This technique has been found to be more efficient and robust with high detection rates. The tool compares the signature generated as a result of the analysis with the existing malware database for classification purposes. DroidAnalytics has identified 2475 malware applications from 102 families. When compared to previous permissions-based malware detection techniques, signature-based DroidAnalytics easily tracks application mutations and their derivatives which generate new malware. The building block for signature-based malware analysis is also based on systems such as extensible crawlers in DroidAnalytics [59]. Third-party marketplaces for applications can be identified where the crawler performs a regular download of available applications. This aids in the enhancement of the existing database towards malware analysis and detection [49]. With continued updating of such third-party marketplaces or websites, extensible crawlers have continuously been developed to detect when new applications are available for download. This crawling functionality makes the DroidAnalytics system to be the initial evaluation point for the security of the applications. Detailed security analysis and security reports are given instantaneously with multiple developers and stakeholders on the lookout for signs of evasion of detection techniques.

The dynamic payload detectors are used to detect malicious malware packages that download malicious code via the internet using virtual machines. Attachments are scanned to detect suspicious files which may contain '.elf ' or '.jar' file types [61]. Camouflaged files are generally used by hackers to avoid detection. Signature dynamic payload detectors overcome this by checking magic numbers in the applications as opposed to the file extensions. The dynamic payload detector treats these files as targets if they have characteristics such as internet permissions or show attributes of re-delegating other applications to download the files. The suspicious files, as well as the downloaded files, are then transferred to the signature generator which carries out further analysis based on existing data to determine the nature and threat level of the malware.

Another form of Android application security analysis is the Android App Information (AIS) Parser which is a structure within Droid Analytics used for the reorientation of the 'apk' file structure used for Android applications [61]. The underlying cryptographic signature of an application can be revealed using this approach where its package information, permissions, disassembled codes, among other aspects, can be identified [74]. The AIS parser conducts a decryption process which makes it easy to for analysis and retrieval of information which is checked against databases for malware detection purposes. Sophistication in current malware architecture has made it easy for perpetrators to bypass this through mutation or re-engineering of applications. The signature-based analysis also requires a flexible analysis which is not facilitated by the cryptographic hash for security scrutiny. For DroidAnalytics signature analysis, three-level signature generation schemes have been developed and found to be more efficient. The DroidAnalytics signature analysis depends on the nature of the mobile application, the calls of the applications, methods as well as any dynamic payloads of malware [61]. This technique does not only cater to application obfuscation but classification and analysis for easier detection in the future.

Androsimilar [47] utilises signature-based methods to detect variants of known Android malware families. They report a true positive rate (TPR) of 76.48% in deducting known

malware families using a signature-based method. DroidSift [74] implemented a signature-based process to detect known malware instances, 93% of the time correctly. DroidSift uses an API dependency graph for constructing feature sets for classification purposes on a data set of 2200 malware and 13500 benign applications with an accuracy of 93% using Naïve Bayes classifier.

DroidLegacy [91] extracts signatures that recognise malware developed by piggybacking benign applications with malicious code. DroidLegacy decompiles the piggybacking malware application into loosely coupled modules and then compares their API calls with a known malware family signatures. Their method has achieved 94% accuracy; however, their dataset is imbalanced with 1052 malware and 48 benign applications. DroidLegacy achieved 87% precision on the entire dataset, which may be due to the skewness.

APK Auditor [92] is a permission-based Android malware detection tool; it utilises static analysis to extract permissions information and store it along with the analysis result into a signature database. The tool also contains an Android application that is stored on end-users' mobiles to allow them to request analysis. The application communicates with a central server, which provides the result of the investigation. APK Auditor has been tested on 6909 malware and 1853 benign applications dataset with a reported accuracy of 88%.

According to Howard, Pfeffer, Dalal, and Reposa, [93], most of the intrusions happens without the realization of the defender. It is, therefore, critical to anticipate and prepare for malware attacks in advance. Howard, Pfeffer, Dalal, and Reposa [93] present a technique based on the prediction of future malware variants' signatures. The predicted variants are then injected into the defensive system of the Android device. The study uses machine learning to identify and predict future patterns to create a malware signature that can be detected by the system. The signature-based analysis is added to machine learning so that it can be able to identify future attacks. Essentially, the researchers add the signatures to the predictive malware defence (PMD) to detect new variants of malware and initiate protection. By introducing signature-based analysis and retraining SESAME, 11 more samples were detected with no

false positives. The SESAME was retained on 114,000 malicious and 10,000 benign binaries [93]. This result indicates that it is possible to predict malware evolution and develop signatures for system defence.

Alam, Adharbi and Yildirim [94] utilize signature-based analysis to build a signatures database in a vector space from a dominance tree using common Android application signatures. These signatures are then used to detect malware. The features used in this study include permissions and flow control from a sample of 1294 applications labelled as either benign or malicious. For malware detection, the application samples are processed and reconstructed for new signatures with similar characteristics as the original signatures. The study's findings show an improvement in detection rate and false-positive rates in the range of 1.7%-0.4%. The only limitation highlighted regarding the technique is that it is unable to detect malware that uses obfuscation and encryption techniques [94].

5.2 Behaviour-Based Detection

Behaviour-based malware detection observes an application's behaviour for suspicious activity to classify it as malware or benign. The behaviour of an object, as well as its potential actions, are analysed for threats and suspicious activities. The behaviour-based Detection flags that the object is questionable, and it is suspected to be malicious when it attempts to execute unauthorized or abnormal actions. During run-time, there is a swam of behaviours that indicate the presence of potential danger. Examples of such acts include attempts to register for automatic-start, disable security controls, discover the sandbox environment, or install rootkits.

The Behaviour-based detection technique works through observing and assessing the context of each line of code processed from the malware. The method analyses all access requests to services, specific files, connections, URL, and processes. The analysis includes all instructions invoked and processed at the program and operating system level, including low-level lines of code embedded in rootkits. The technique determines all activities that are

considered suspicious and malicious, which, when aggregated, may distinguish malicious objects before they can be released into the system and network to execute its behaviour. The technique may utilise the following functionality for creating a behaviour profile of an application: monitoring sensitive API, network traffic monitoring, files operations, memory usage, and CPU usage.

Calderon et al. [95] presented a behaviour analysis technique focusing on HTTP and HTTPs network traffic packets. Malware may try to communicate with a command and control server using HTTP or HTTPs protocol before executing malicious behaviour. Their technique used the occurrence of each client header used in HTTP packet and in the case of HTTPs, the encryption algorithm (TLS 1.0, 1.1, or 1.2) used. The features are fed to Machine Learning algorithms, and the experiment result shows that the technique has achieved over 90% of precision and recall. However, malware using deprecated protocols such as SSL cannot be detected by this method [95]. Marin et al. [96] presented a Deep Learning approach for detecting malware by monitoring network traffic. Their approach does not require expert knowledge on using the necessary features, i.e. instead of using shallow models where handpicked features are defined by experts for pre-processing, raw data received from the traffic stream is used as input for Deep Learning models. The results show that using their approach, the Deep Learning method performs better than the Random Forest algorithm.

Lungana-Niculescu et al. [97] proposed a method of reducing the false negatives and false positives of a behavioural analysis technique in malware detection. The author introduces deep learning-based classification as a supplemental decision-making element. Their method monitors the actions performed by the malware process and sets the flag for suspicious activities as 1, otherwise 0. Examples of questionable actions are injecting itself into another process, launching sensitive operations, and establishing itself as a start-up process. These suspicious actions are used as features for the deep learning algorithm. The experiment result showed 118 false negatives and 112 false positives, successfully reducing the false positives by 97%. Overall, they achieved 94.53% accuracy, 92.1% precision, and 87.88% recall on the

validation data set. However, the study was unable to establish the correlation between behaviours and malware files, which was classified as future work by the authors so that the results can be seen at a file level, not just behaviour level [97].

Onwuzurike et al. [98] sought to detect malware using behaviour analysis based on a sequence of abstracted API calls to capture the behavioural model of Android applications. The behavioural analysis technique was used to model the MAMADROID, a malware detection system that depends on transitions between different Android applications API calls. API features were used in this case. The MAMADROID system models a sequence of API calls that are, in turn, used as features for machine learning to classify applications as either malicious or benign. The study evaluates the effectiveness of the MAMADROID using a dataset of 8500 benign 35500 malicious applications [98]. The operation mechanism is based on the abstraction of API calls the Android applications without interfering with its behaviour. They reported F-measure of 0.99, 0.97, and 0.9, depending on different datasets the models were being tested. A limitation of their study was the skewed dataset as the malicious applications were more than four times of the benign applications used in their experiments. [98]

Researchers are in a continuous race to determine effective ways of detecting malware due to the increasing vulnerability of Android applications. Alptekin, Yildizli, Savas and Levi [99] present the TRAPDROID framework that focuses on capturing the unified behaviour of applications. The framework detects malware by analyzing process events such as system calls, broadcast events (such as SMS_RECEIVED, SCREEN_ON, etc.) hardware performance, and binder statistics. The study uses behaviour analysis to evaluate suspicious applications from 355 malware and 281 benign samples [99]. The behavioural analysis helps determine whether a file is trying to open without the necessary permissions, making it easy to detect ransomware. After adjusting for SVM, the success rate of the framework reached 93.2%, implying that the behavioural analysis technique was able to identify most of the malicious applications from the data set.

Olukoya, Mackenzie and Omoronyia [100] proposed the use of behavioural characterization to assess malware risks in new applications. In this case, the behavioural approach is used to describe permission sensitivity and mismatch by evaluating self-triggered permissions against the requested authorizations. Notably, the permissions requested by an application describe its behaviour and its interactions with the device, other applications, and stored data. Sensitive data that may require permissions in Android devices include call logs, text messages, and contact lists; As such, analyzing the behaviour of application permissions is critical to ensuring privacy. From a dataset of 33, 580 malware, and 10,000 benign permissions extracted from 10,000 applications from google play, the study employs behavioural characterization to assess risk ratings of Android permissions based on intent and API call features [100]. According to Olukoya et al. [100] analyzing risk signals, in-app permissions can help protect innocent users from falling victims of malware intrusions. Further behavioural characterization helps in labelling malicious and benign applications. The proposed framework for this study demonstrated 95% accuracy in improving risk signalling. However, less flexible contextual factors (such as actual usage of permissions in the source code) limit the framework from attaining higher accuracy in estimating risk ratings [100].

Malicious activities such as phishing, spoofing, and eavesdropping have become widespread with the recent developments of Android applications, thus the need to develop more accurate and useful malware detection techniques. Sourav, Khulbe, and Kapoor [101] use behavioural analysis to run a test on Android application files to determine if these should be classified as benign or malware. To aid the investigation, the researchers extracted a dataset from the Kaggle database containing 338 applications labelled as malicious or benign. The presence of different permissions and calls indicated using specific codes (0 and 1) for ease of classification. The analysis was based on 330 features expressed in the binary form [101]. These features were categorized into application permissions, a standard operating system command, and API calls. The study's findings illustrate that the incorporation of behavioural analysis into the analysis of neural networks (ANN) model increased the accuracy level to

95%. This result coincides with Olukoya, Mackenzie and Omoronyia's findings [100]. Sourav et al. [101] succeeded in introducing an integrative method based on deep learning to detect malware that performs better than the traditional techniques. However, the analysis is limited in that it contains a smaller data set; future work is proposed to use a more significant data set so that this method can be tested on a complex data set. According to Fan et al., [102], using an automatic behavioural approach (CTDroid) can help informative engineer features from Android malware. However, this technique is challenged by difficulties in recognizing information and semantic gaps between a programming language and extracted behaviours. After evaluating benign and malware applications, the model demonstrates that it can achieve 95.8% with only 1% of false-positive rate [102].

5.3 Machine Learning-Based Detection

The penetration of Android platform and sensitive industries such as banking, healthcare has increased the danger and risk associated with Android malware due to the dynamics of the targets [75]. Machine learning technique of malware detection has been in use and has developed significantly over the years as improvements to the platform are carried out, and more malware developed [103]. Challenges associated with machine-based classifiers include extraction of feature representations from applications and having to choose classifiers that can be trained exclusively on one class. Different approaches have been used to counter these challenges [76]. Significant research and testing have been established for detecting Android malware. Some strategies use power consumption to determine where anomalous patterns are used to detect possible malware. In other methods, system approaches are used to identify unusual patterns which are flagged out for further analysis and possibility of malware.

More conventional approaches use signatures to carry out comparisons with known malware structures. New Android malware may go undetected for prolonged periods even when antivirus infrastructures are set up. The lack of timely updates of newly created malware signatures means that the malware architect has a significant window for carrying out attack

undetected [76]. In some instances, the presence of malware is only detected after the damage or intended malice has already taken place. Google has invested in detection infrastructure when applications are uploaded in the play store. However, malware developers are still able to craft malware that can be downloaded from the store undetected [104]. The aim of having no application infected by malware is also made more complicated by the open policy, which allows users to download applications from third-party platforms. Machine learning detection techniques have been developed to deal with the extended period that it may take to detect new malware [2]. The aim is to narrow the window of opportunity that attackers have before a threat is flagged and efficiently dealt. The machine-learning malware detection techniques are effective and significantly reduce the amount of time required to realise that new forms of malware have been released [2]. Abusnaina et al. [105] point out that this approach to malware detection entails an inbuilt set of methods that give Android devices learning abilities without explicit programming. In simpler terms, machine language use algorithms to identify and formalize the characteristics underlying the data availed rather than adding a programmatic set of rules for classification purposes [106].

Kumaran and Li [107] used Machine Learning algorithms to improve malware detection from a data set using static analysis. The machine learning techniques are trained from a dataset of 500 benign and 500 malicious applications and accuracy determined by a 10-fold accuracy scheme. The features extracted (I83 in total) from the dataset were classified into requested permissions, declared permissions, and intent filters. Kumaran et al. establish that machine learning is an accurate (91.7% accuracy) classifier of complete data sets, especially when combined with permissions [107].

Different machine learning techniques have been developed over time to deal with the ever-growing list of Android malware. For instance, Risk Ranker [108] is an automated risk assessment technique which profiles applications in Android markets for the probable presence of malware. Profile Droid uses a multi-layered profiling system which seeks to characterise the behaviour of individual applications at different levels and aspects, flagging

out suspicious components for further analysis and categorisation [63]. The Android Application Sandbox [109] has been utilised in static and dynamic detection mechanisms where decompiling of Android applications codes is carried out, and risk scores determined and classified concerning risk priorities. Risky samples are then subjected to further analysis and comparison for further understanding of threat levels and capabilities. Most antivirus companies have used machine learning methods to detect computer malware, and the same is being replicated for the Android system. Benign and suspicious classifications have been used in the Bayesian classification as a way of categorising the risk level for applications [110]. The machine learning concept used samples of 49 Android malware families that have been earlier detected and classified as well as 1000 benign Android applications. Applications appearing to be suspicious are flagged for further analysis and categorisation. Other machine learning techniques use graphical representations and training where flow graphs and permissions on SVM models are used to differentiate benign and suspicious applications [89].

Machine learning that is trained using permission features has varying levels of accuracy in malware detection. The higher the number of malware samples and benign applications, the more robust and useful a machine learning detection technique is deemed to be [104]. Permission features are used to train SVM and Bayesian models, where the risks associated with respective applications are evaluated and ranked [63]. However, this is dependent on the framework of each machine learning technique and algorithms utilised therein. For instance, DroidMat uses k-means clustering through the SVD or Singular Value Decomposition approach [56]. The framework was based on 238 malware samples from 34 families as well as 1500 benign Android applications. The effectiveness of machine learning is also dependent on the extent of the database used for the process. The database also needs to be frequently updated either automatically or manually to capture a broader range of probable malware.

An SVM-based approach has been used by Li et al. [111] that combines risky permissions and sensitive API calls and feeds them as features to the SVM algorithm. They have achieved 86% accuracy on a data set of 700 applications, consisting of equal distribution of malware

and benign applications. Sanz et al. [112] acquired an accuracy of 86.41% on a data set of 249 malware and 1811 benign applications. Their method extracts permissions from an Android manifest file and utilises various machine learning algorithms for detecting malware. A Random Forest algorithm trained with fifty trees achieved the highest accuracy. DroidDet [66] employed the Rotation Forest algorithm and trained it with features extracted by conducting a static analysis of 1065 malware and 1065 benign applications. Static analysis extracted permissions, monitoring system events, sensitive APIs, and permission rates as features for constructing the machine learning model. Their method achieves 88.26% accuracy, and they report a 3.33% improvement to the SVM algorithm.

Deepa et al. [113] applied feature extraction and dimensionality reduction methods - namely Information Gain, Correlation Feature Selection (CFS), and Kruskal methods - in their static analysis approach. Their technique utilises the extraction of three types of features: method names, strings, and opcodes for constructing machine learning models. Their dataset consisted of 612 malware and 758 benign applications. Their techniques achieved 88.75% of accuracy by using Adaboost with J48 as the base classifier.

AppContext [114] extracts security-sensitive behaviours by conducting a static analysis. The security-sensitive behaviour is classified as API calls to certain methods that are permission-protected or are sink method, i.e., require access to functions that write data to a file. This information is used to train an SVM classifier. AppContext achieved 93.2% accuracy when applied on 202 malware and 633 benign applications datasets.

Image representations can be used for detecting malware, a study by Darus, Ahman, and Ariffin [115] decompiled 300 malware applications and 300 benign applications into classes.dex files which are the binary files that are run on the Android operating system. They converted these files into 8-bit grayscale image representations and used GIST Descriptor to extract features for feeding to machine learning algorithms and managed to detect 183 malware files. However, they were unable to generate images for 117 malware files due to

corrupt APK files. Their experiment result showed the accuracy of 84.14% with Random Forest algorithm.

Machine learning can also be customised or designed for specific Android systems with particular functionalities to facilitate quicker detection and shorter windows of opportunity for full blow spread and attacks [67]. With more complex malware being developed in the current Android scope, machine learning techniques have also become more complex and sophisticated to facilitate early detection [116]. Machine learning has also been used in conjunction with other detection techniques as a way of complimenting them and having robust and highly capable systems. These techniques check for multiple indicators such as many packets sent, processes running at one given time, touch screen pressing [68]. These indicators are analysed for anomalies, which are then used to determine the risk of malware. Based on the machine learning algorithms and settings, the activities associated with the Android applications are used for data collection and then classified as normal or abnormal [117]. Abnormalities are further analysed and compared with existing malware patterns. For instance, Crowdroid uses behavioural based malware detection techniques where system call patterns are converted to feature vectors which are then subjected to k-mean algorithms which then detect the presence of malware in the applications [65].

Some of the typical features that are used in machine-learning malware detection techniques are the SMS manager's API for receiving, sending and reading short messages, the phone manager's API for accessing device identity, network operations and the identity of subscriptions [66]. Other features include the package manager's API for retrieving information about installed packages as well as downloaded packages that are yet to be installed.

A regular feature on the learning framework depicts safe application, while abnormality may represent the probability of malware [63]. However, this is dependent on the ability of the machine learning platform to utilise various feature such as permissions and bag of words extracted using text mining approach [63]. For most of the machine learning techniques used

for malware detection, robustness and performance are critical aspects. Robustness is the ability to extract various features to develop multiple detection mechanisms in case one or more elements have been designed to avoid detection. Robustness can be achieved through the extraction of different parts of APK application files with commands being detected from the executable files [54]. Having various features allows for faster identification of malware and their specific intentions. For instance, ensemble machine learning algorithms such as Random Forest can utilise these features simultaneously to detect malware [55].

On the other hand, performance is determined by the diversity of the employed features. The performance can be further enhanced by having a large malware repository which acts as a guideline to the features that are most targeted and probable loopholes that malware architects can take advantage. Machine learning is, therefore, dependent on the ability to progressively update feature and malware samples as new and more complex ones are developed.

Fatima et al. [118] present a Machine Learning approach that uses a genetic algorithm for selecting discriminatory features in malware detection. Two sets of applications, including 20,000 malware and 20,000 malware samples, are analysed using reverse engineering. Classification accuracy of 94% was obtained after reducing the number of 99 features to 33 by using Genetic Algorithm. Their main contribution is the reduction of features using a heuristic searching approach based on fitness function for feature selection [118].

With the penetration of Android platforms and system, multiple features are developed progressively and are susceptible to malware as opposed to conventional functions. Even though machine learning techniques are better than signature-based detection techniques, they also have some shortcomings [56]. One of the primary challenges lies in the identification of contributed features. API calls and permission are the most widely used feature for machine learning. The second challenge relates to the scope of machine learning with only a few large-scale databases that can be relied on for efficient malware detection and machine training models [116].

5.4 Taxonomy of Android Malware Detection

In previous three sections, we attempted to answer the first research question: How do we classify Android application security analysis provided in the literature? We have identified and reviewed two primary categories: Signature-based detection, and Behaviour-based detection, as well as complimentary category Machine/Deep learning. Whilst performing the literature review in pursuit of finding answer for our research questions, we have derived a taxonomy for Android malware detection. The proposed taxonomy is displayed in Figure 8 below.

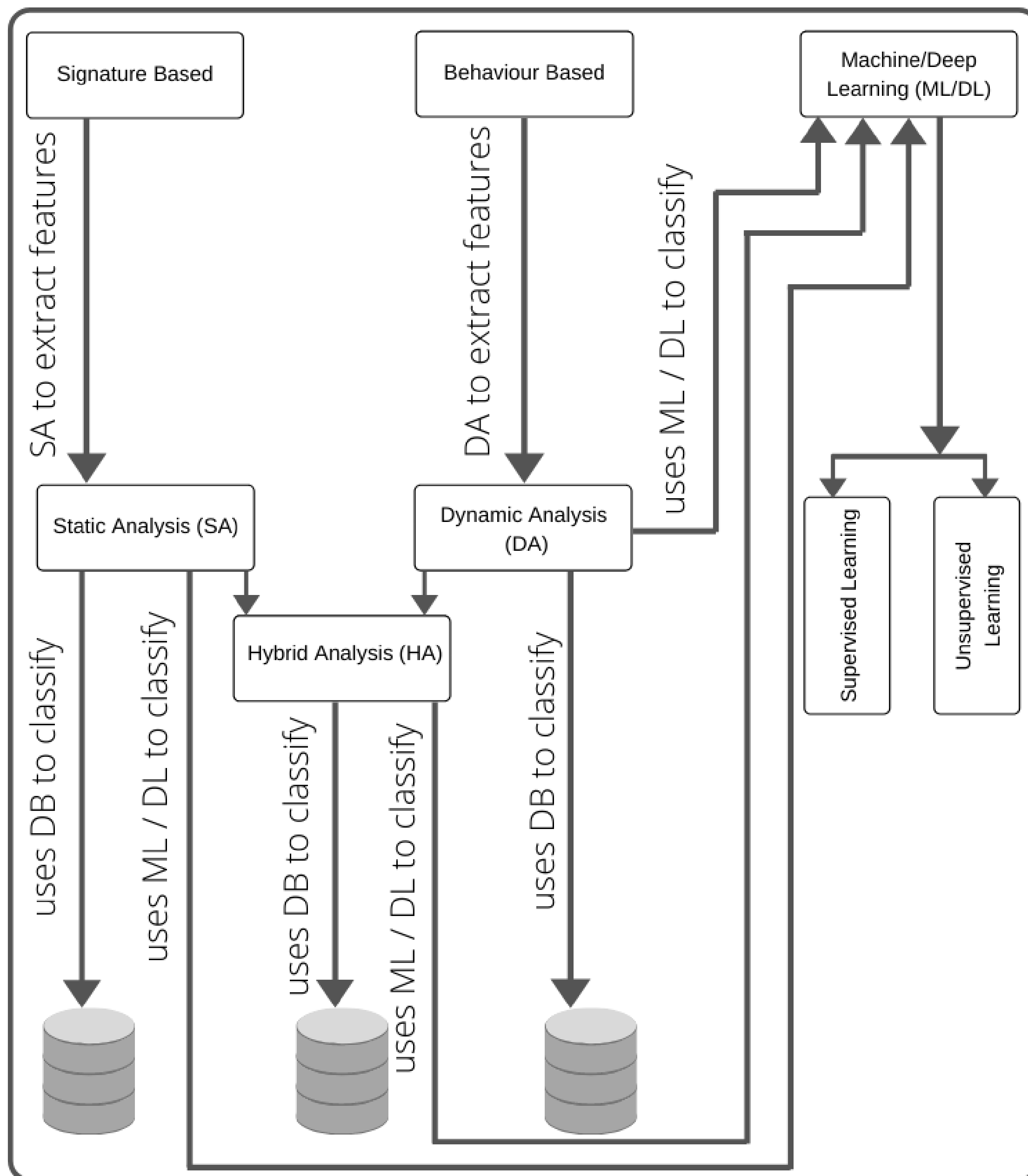

Figure 8: A Taxonomy of Android malware detection

We have discussed three broad categories of Android application analysis with regards to security in the current chapter, these categories are shown in the above figure. The first category Signature based detection utilises Static Analysis technique to extract the required features. Static Analysis uses one of the two methods to classify an application as malware or benign. The first method utilises the extracted features and compares them against malware database features for classification purposes. The second method utilises the third category: Machine Learning or Deep Learning for classification purposes. Similarly, the second category Behaviour based detection utilises Dynamic Analysis technique to extract the required features and uses either existing malware database or Machine/Deep Learning algorithms for classification purposes. The third technique, Hybrid Analysis, utilises features from both Static and Dynamic Analysis against existing malware database or utilises Machine/Deep Learning algorithms for classification purposes.

The next three sections will discuss above-mentioned three analysis techniques in more detail. The next sections will also discuss the custom-build detection technologies use for detecting malware and will also attempt to answer the second and third research questions:

RQ2 What is the current state of analysing Android malware detection techniques and technologies?

RQ3 What challenges, gaps, and patterns might be deduced from the existing research attempts.

5.5 Techniques for Malware Detection

This section attempts to answer the first part (techniques) of the second research question. Broadly speaking, there are three types of techniques for detecting malware. We will now discuss each of these techniques in more detail.

5.5.1 Static Analysis Techniques for Malware Detection

One of the common forms of malware detection is a static analysis which uses features that are extracted without executing code [119]. When compared to dynamic detection techniques, static analysis is considered to be more efficient while dynamic analysis is more useful when dealing with obfuscated code. Static analysis of malware detection uses code analysis to detect malware [120]. Obfuscation allows malware architects to bypass detection unless complementary techniques are used. However, it is useful in detecting well-known malware and has a low-cost implication [121]. It also consumes less time and resources to carry out static malware detection analysis.

Static detection has two main techniques which include misuse detection and anomaly detection. The misuse detection technique is also referred to as a signature-based technique which uses a set of predefined sequences and instructions to detect if malware matches defined patterns. Semantic language-based signature approaches are used where families of malware have specific signatures that are used to identify them [25]. Data flow properties are also determined using static analysis techniques. The challenge, however, is to define signatures that can effectively detect obfuscation as well as dynamic code loading challenges [122]. The misuse technique also uses an approach where specific security features are extracted and checked against data flow to determine the presence or absence of malicious signatures.

In addition to the misuse technique, the static analysis also utilizes anomaly detection, which relies on machine learning algorithms to identify Android malware. Features that are extracted from known and identified malware are used to train a machine learning model which then detects new or unknown malware [66]. For instance, a K-nearest neighbours approach to train a machine learning model based on features such as short message and call patterns as well as other applications. Android events and permissions are also used to prepare models which then classify malware as well as benign applications.

Chavan, Troia and Stamp [123] presents a comparative analysis of benign and malicious applications using the static feature. The dataset consisting of 989 benign and 2657 malware samples were analyzed using various machine learning techniques such as logistic model trees, artificial neural networks, and AdaBoost to extract permission features for Static Analysis. A total of 230 distinct permissions were obtained in experiment 118 of which were from the malware dataset [123]. The study's findings illustrate that even a small number of permissions can serve as a strong feature vector. Although the study faced a challenge in malware classification, it was able to attain 95% detection accuracy with random forest.

Mehtab et al. [124] integrates machine learning techniques to train models based on static analysis for detecting malicious applications. The data set for the rule model involved a sample of 910 malicious applications and 510 benign applications. The model, which is rule-based, was founded on features such as permissions, functions, codes, and call features, which help determine the correlation between the possibility of an application being benign or malicious. By integrating machine learning, the accuracy of the model reached 99.11%, implying easier detection and categorization of malware [124]. However, the sample size used in this study was significantly small, which affects the quality, accuracy, and reliability of insights provided. A more significant data set would be more appropriate for future research. Another deficiency in their approach is that they trained the model on 40% sample and used the remaining 60% sample as the validation data set. However, stratified cross-validation was not used in their method, so extracting more information about the algorithm’s performance is not clear.

Jiang et al. [125] develops model based on static analysis to evaluate application information using an opcode sequence. The study uses opcode sensitive features such as APIs, and STRS to classify malware into their respective families from a sample of 5560 malware applications. The experiment achieved 99.5% accuracy in allocating malware to their correct families [125]. However, they have not provided any result on identifying malware from a set of both benign and malware samples. Gamao [126] uses permission-based static detection to generate application features from Android applications. Fifteen distinct permissions were

utilised for 25 benign and 25 malicious applications and classified using Random Forest and Naive Bayesian machine learning [126]. Their study showed that the Random Forest performed better than the Naïve Bayesian, as the former produced 4.8% error rate, compared to the latter that achieved a 9.5% error rate in classifying malware.

Static analysis applications may also check for the functionality and probability of malware presence through analysis of the source code [127]. Android application source codes are known and can be detected for anomalies or unusual patterns or commands that can be linked to malicious intent. The detection is done without the execution of the application, which is essential in the discovery of malicious patterns that may not operate or function unless a particular set of conditions are met. Hence, the malware may remain inactive throughout multiple stages until its engineered designs identify that all requirements for it to cause malice are met. The static analysis identifies such patterns and raises a red flag over the same. However, if obfuscation such as renaming strings or variable names is incorporated into the application, it may be difficult for static analysis to identify unusual patterns [127].

With multiple malware being regularly developed, using static methods to detect malware may be inadequate hence the need for other techniques such as dynamic methods and machine learning. Static techniques are useful in instances where Android systems are not densely connected to the internet or do not receive new applications regularly [128]. Applications work under permissions granted by users to perform specific actions; hence permissions provide a critical role in controlling applications' access rights. Application installation prompts users to allow access to Android resources in the system. The permissions-based approaches analyse the permissions sought by an application and determine whether unusual patterns are detected. For example, if a calculator application seeks permissions to read SMS or user's location, then this is an abnormal pattern. Where unusual patterns of effects than usual are detected, applications can be flagged for further analysis with possible malware presence. This analysis is carried out on the manifest file leaving other types of data out [66]. Even

though most malware families target the manifest file, different targets cannot be detected using this approach, hence limiting its detection capabilities.

A machine learning-based framework (AndMFC) developed by Turker and Can [129] succeeded in extracting 42 permission and 958 API call features from a 24467 malware samples and fed them to different machine learning algorithms. The framework had achieved higher than 96% accuracy; however, the recall (89.06%) and F1 scores (91.94%) were low [129]. Furthermore, the accuracy (93.63%) was lower than the overall accuracy in the detection and classification of unknown malware.

The static analysis technique analyzes executable files on a structural basis. Such files have many static features that are extracted by Portable Executable (PE) file. Ijaz, Durand and Ismael [130] use a static analysis technique to extract features from a sample of 39,000 malicious and 10,000 benign files. In total, 92 features were extracted statically, including PE files such as sections and headers, which contain other feature groups. Important features used the analysis include time-date-stamp, number-of-symbols, major-link-version, and subsystem, among others. The study succeeded in establishing that static analysis is more accurate in detecting malware in internet-based devices than dynamic analysis due to intelligent behaviour of malware. For example, malware may not execute its malicious code when it detects a controlled environment.

Some of the techniques for signature and permission-based static approaches include entry point analysis which determines where a program may commence using call-backs and activities that are user-initiated. Malware detection using this technique analyses the nature of permissions to determine whether there are signs of program starts which are not user-initiated. Another method used in the static analysis is the reachability analysis which tracks the possibility of following paths related to two locations [90]. Other static analysis methods include failed initialisation analysis which tracks whether objects have been initiated in the correct sequence and flags out abnormal sequences [131]. Cyclicity analysis, on the other hand, analyses whether running applications create a specific cycle of execution that may

have anomalous or malicious outcomes. If the cycles created matches the signature cycles of a known malware family, then they can be flagged for further analysis. Path length analysis reviews pointer differences from an application element or variable based on the maximum number recorded [119]. Safe applications lie within normal parameters, while wicked ones show signs of abnormal quantities. Data flow analysis on the static method focuses on the application's aspects such as context-sensitivity, path sensitivity, flow sensitivity, interprocedural and intraprocedural analysis [119].

5.5.2 Dynamic Analysis Techniques for Malware Detection

Dynamic analysis malware detection technique utilises data execution on a real-time basis. The method aims to detect anomalies in malware when they are operational or running as opposed to analysing the applications offline [62]. By loading target data, analysis of applications is enabled through the evaluation of behaviour, which can be used to detect anomalies or malicious patterns [132]. This technique is more resource-oriented than the static technique, which uses signature and permission-based analysis. The operating environment for the method uses a sandbox, virtual machines, among other ways to simulate the execution of an application to identify specific behavioural models [121]. The execution of the applications is done analogous to the real-time scenario.

Dynamic analysis allows researchers to run the chosen malware in a controlled environment to collect its capabilities and purpose. A study by Uppin and George [133] provides a step by step process in the identification of a MasteryBot (the chosen malware) through dynamic analysis. The process entails setting up a safe malware environment that involves the host, guests, and Android emulator machines' configuration to facilitate the extraction of the required features for analysis. The findings show that the Trojan can hide as a fake adobe flash player and tries to download key logger from an external source [133]. The malware encrypts all files in folders and subfolders and automatically deletes the original files after encryption is completed. The malware displays a dialogue box to force users to watch

pornography and directs them to send an email to 'googleprotect@mail.ru' to retrieve their encrypted files [133].

Based on the complexity of malware, dynamic techniques can also aid in the identification of next targets based on the detected codes. For instance, IP addresses can be used to identify spatial addresses used to determine the uniformity and geographical distribution if a host is affected by the malware. In addition to detection of the malware, the dynamic technique also enables identification of malware patterns and intended targets for contingency purposes. Multiple methods have been tested using this approach with high detection rates of more than 90%.

The experiment results by Kumara et al. [134] illustrate the use of a semi-automated machine learning framework based on dynamic malware detection. The experiment, which includes 1400 malware applications and 1600 benign applications, utilizes a machine-learning algorithm to classify behavioural features under a supervised model. In his case, the accuracy of detection depended on the monitoring time for the malware execution and applying k means cluster feature on the data collected. The results show that the framework can provide a reliable malware detection technique for Android applications [134]. Sang et al. [135] developed a three-step (feature extraction, model training, and assembling the models) for malware detection. This framework is based on deep learning. The study uses handcrafted features, including N-gram (a continuous sequence of n items from a given sequence), API calls, and image representation of binary file to analyse a sample of 14,000 malware and 14,000 benign applications. The resulting framework attained 96.24% accuracy in malware detection from real-life datasets [135].

The dynamic analysis technique is robust against obfuscation if malicious codes are executable as it does not rely on source code only [33]. Bacci et al. [136] illustrates how the application of Android morphing techniques affects the effectiveness of dynamic analysis. Specifically, the study investigates the degree at which obfuscation techniques jeopardize the efficacy of dynamic analysis through experimental analysis consisting of 3500 trusted (Google

Play Applications) and 3500 malware applications [136]. Each of the applications is executed on an Android device, and the effect is measured in terms of accuracy. From the experiment outcome, obfuscation techniques affect the effectiveness of the method moderately regarding false negative and positive rate. Therefore, the process performs reasonably well in analysing non-obfuscated and obfuscated malware. The technique is efficient in the sense that it induces minimal alterations to the application execution trace. The main limitation highlighted is the inability of the technique to evade intelligent malware behaviours [136].

Dynamic analysis can detect dynamic code loading, which happens during runtime where it is recorded [25]. However, it fails to determine the degree of code that is executed within an application because one path is identified for each execution, hence cannot achieve 100% malware detection rate. The overheating of hardware, when running applications, also makes it challenging to implement the dynamic analysis in comparison with the static technique. Overheating may occur due to high CPU usage due to parallel processing or other CPU intensive work when performing dynamic analysis directly on the device. However, this limitation is becoming a lesser issue as the newer mobiles has efficient cooling mechanism. Different algorithms are utilised when designing dynamic detection techniques [137]. The methods may focus on different Android functionalities that are mostly targeted by malware architects. These include network, API call, SMS and location, among others. Some techniques record the frequency of the system API calls where anomalies are flagged for possible malware presence. However, malware can only be detected when an application meets a specific API threshold.

Another dynamic analysis technique is TaintDroid which captures data over networks for application analysis [138]. Abnormal patterns over the network can be identified and analysed further for malware capabilities. Other techniques used in the dynamic approach include monitoring and tracking of system calls where classification is later done based on algorithms developed using machine learning techniques. Using typical applications and samples of known malware such as GoldDream, Rio Unlocker and KungFu, allocated resources are

monitored once applications are started. The behavioural patterns are recorded and extracted, and the resource data converted into feature vectors; these vectors may be based on functional categories such as CPU usage, SMS, Network, Virtual memory, etc. Using Naïve Bayes and Random Forest cross-validation elements, correct classification of data types that were considered normal yield high accuracy. However, some malware applications were detected as benign. Random forest was, however, superior in detecting benign and malicious applications when compared to other algorithms.

Other dynamic techniques include the Bayesian method and the Chi-square methods which can be used separately or in a combined form. Both are machine learning algorithms which are being used for feature selection [139]. The Bayesian algorithm usage increases accuracy levels; accuracy levels are approximately 80% but rise to 89% when combined with Chi-square [140]. Benign applications and their repackaged versions can also be examined by dynamic analysis through self-written applications that are installed on Android devices and analysed for behavioural patterns.

Sihwail et al. [141] takes an integrated approach by combining memory forensics and features of dynamic analysis in malware detection to extract malicious artefacts from memory using a data set of 1200 malware and 400 benign applications. The study aimed to determine whether using an integrated approach increased the accuracy of malware detection and reduced the creation of falsely alarmed files. The findings incorporate the relevant features from memory, and dynamic process reduced false positive rate to 1.7% [141]. In essence, the approach outperforms other analysis methods. In isolation, Dynamic analysis is limited in that it consumes a lot of resources; however, the approach of Sihwail et al. overcomes the resources issue a little bit by incorporating features that are only available in the memory. However, dynamic analysis can be easily outsmarted by some malware that can detect the controlled environment and provides a single way of file execution [141].

Network Traffic analysis is another dynamic analysis technique, performed to extract features from network traffic generated by an application to create traffic patterns. Traffic patterns may

be generated by extracting information from packets, e.g. source IP, destination IP from outgoing packets only, incoming packets only, or both incoming and outgoing packets. Other features may include, the total number of packets in the inbound direction or outgoing direction or both, the maximum and minimum size of packets [142]. Traffic patterns are different for benign applications which aid in the identification of malicious ones. The extracted features can be fed into machine learning algorithms for classification purposes. Rather than detecting malware when applications are offline, dynamic analysis enables the detection of malware when in action. This way, the target data can be identified, as well as the mode of operation [67]. Through dynamic analysis, the pattern of malware, e.g. API calls, system resource access, URL accessed, memory usage, and CPU usage etc. can be entered into databases to aid in future detection using other techniques as well.

A study by Fallah and Bidgoly [143] benchmarks machine learning techniques by analysing features such as packet size, duration of the network flow etc. extracted from the network traffic to detect malware. Their experiment utilised 600 benign samples and 400 malware samples. Although the technique attains 90% F1-measure of malware detection, it does not show acceptable results in identifying new malware families [143].

There are several dynamic approaches which yield varying results and accuracy levels. For instance, anomaly-based detection techniques use machine learning methodologies to identify and detect malicious patterns in applications. The anomaly-based approach carries out an in-depth analysis because it utilises a statistically higher number of features (e.g., network packets or system calls). Each feature is an anomaly in comparison with a baseline, hence requiring a lot of resources to execute. If the approach invokes excessive system calls, there is a likelihood of identifying benign applications as malware [69]. It may also fail to detect malware that are utilising dynamic code loading to avoid detection. Because the malicious behaviour may not be executed straight away, and malware may wait for a specific input, e.g. input of credit card information. Classification of benign applications as malware is one of the main drawbacks, hence leading to false alarms and unnecessary analysis.

The taint analysis method evaluates variables in applications that can be modified by the users. User inputs can be maliciously harmful if they are not well checked. The dynamic taint analysis process uses the TaintDroid system to check for possible malware [144]. Aspects or elements of importance are marked with a 'taint' identifier which sticks with the information when it is used. The tainted information is then tracked with regards to movements and characteristics. Sensitive elements of targets such as camera, GPS and calls are monitored for possible malicious interferences. The tracking system tries to identify aspects such as information leakages from third-party developer applications. TaintDroid records all this information and any information leaving the system in a manner that it should not. TaintDroid can identify the application carrying out the data sending process for any suspicious patterns or behaviours [145]. One of the drawbacks of the Taint analysis is the inability to track information that leaves the target channel and has a corresponding return through a network reply.

5.5.3 Hybrid Analysis Techniques for Malware Detection

It is obvious from our literature review that neither static nor hybrid methods are 100% accurate in Android malware detection. The accuracy rates vary based on the approaches as well as the targeted areas of interest [44]. One technique may be suitable for one situation or Android system when compared to the others. Therefore, more than one approach may be required in some instances to achieve high accuracy and detection levels. The hypothesis is that using more than one approach increases the chances and detection accuracy. Combining static and dynamic methods should increase the robustness of the detection system and enable monitoring of edited applications more effectively and more accurately. Due to the unknown nature of malicious applications, static analysis can be used initially from an online position and analysed accordingly [45]. After that, the same application can be dynamically analysed through kernel-level sandboxing with prior knowledge of the probability of malware as analysed through static techniques. Android application sandbox is a security technique which is used to distinguish running applications and is typically used for the execution of

applications that are not tested and trusted. For instance, third-party applications, unverified application sources, untrusted suppliers, and user websites applications qualify for testing using the sandboxing technique [70]. It is deemed useful for running applications before risking running them into host machines or Android operating system, which may be possible targets [52].

Static analysis technologies such as EvoDroid and Smartdroid provide higher code coverage for static analysis where possible activity paths are identified before applications can be logged into dynamic analysis systems [46]. The information from the static analysis can then be incorporated into machine learning databases to facilitate further dynamic analysis at a higher level of detection and accuracy. Execution paths can be analysed for the presence of malware where the two approaches are used for higher levels of accuracy and classification [4].

Other approaches that have been utilised include testing where static and dynamic methods have been used to uncover malicious codes. An Android application sandbox is used as the first step to the hybrid analysis where applications are disassembled [109]. A search in the disassembled code is carried out to identify any suspicious patterns before the application is run through an Android emulator which carries out a real runtime analysis. However, caution should be taken while carrying out a hybrid analysis to avoid confusion and cumbersomeness. For instance, the long-time taken by static analysis combined with multiple resources required for dynamic analysis may work to the disadvantage of a hybrid system. It is a complicated and time-consuming operation to extract features from both static and dynamic analysis and may hinder scalability [48]. Based on the intention or the scope of the hybrid system, the disadvantages should be avoided by choosing approaches that do not contradict or complicate the detection process.

Du et al. [146] proposes a combination of static and dynamic with machine learning to create a model for malware detection. Static features are extracted and vectored for testing and training support vector machines (SVM). The SVM model is then used to detect malware. In the study, the researcher uses static analysis to extract resources and information code

segments. This way, all the activities of data in Android software were extracted for use in the model. The advantage of using this static analysis technique is that it is relatively simple, and all the matching segments can be extracted directly. For static analysis, data obfuscation occurs because pattern matching is virtually impossible when it comes to matching maliciousness and application behaviour. API and Permission calls are the most common static features used in Android malware detection [146]. Their process can take either the signature-based or permission-based approaches. Combining static and dynamic analysis to create the SVM model that achieves malware detection accuracy of 94.38% [146].

XU, Zhang, Jayasena and Cavazos [147] proposes a hybrid analysis for the detection of malware (HADM) for identification and grouping of malware. Notably, hybrid analysis utilizes features extracted from the dynamic model execution and uses them in a static analysis algorithm. For this study, a sample of 4002 benign and 1886 malicious applications was analysed. Some of the analysed features include permissions, intent filters, advertising networks, and API calls. The study succeeded in improving application classification accuracy by training deep neural networks (DNN) for each feature set. The accuracy ranged from 87.3%-94.7% for all the tests conducted [147]. However, the Android hybrid analysis is cumbersome as it relies on multiple methods (static and dynamic analysis) to detect malware [147].

Due to difficulties in the construction of malware detection models, a study by Liu et al. [148] proposes the use of a hybrid malware detecting scheme to make improvements to the traditional methods. Some of the proposed techniques to support the scheme included new analysing features, clustering methods, and framework (Androguard) to improve the efficiency of malware detection. Liu et al. use different unique features such as com+ features (a combination of permissions and API calls) and function call graph generated by using Androguard. These features were extracted from a data set of 1000 benign and malicious applications [148]. These applications were categorized into 46 families of malware, and both dynamic and static calls and permission features were used for the study. The study

succeeded in developing a hybrid scheme with functional detection capabilities. All algorithms achieved higher than 86% accuracy while SVM achieved the highest accuracy of 96.92%, but it has the worst false positive rate as it inaccurately classified 24.54% of benign applications as malware. Similar to Xu et al. [147] the research points out that the complexity of using multiple methods limits the applicability of Hybrid analysis.

In another study by Martın, Lara-Cabrerab and Camacho [149], hybrid analysis is used to generate behaviour information for developing a new malware detection approach by fusing static and dynamic features. The information generated during the analysis is critical in building a classifier that integrates dynamic and static feature vectors. A comprehensive dataset of 22 000 malicious and benign samples was analysed to extract static features such as permissions, intents, packages etc. and dynamic features such as files accessed, phone calls made, SMS sent etc. [149]. The resulting classifier is slightly more accurate (89.7%) compared to individual contributors, that is, dynamic (78.6%) and static analysis (89.2%) used in their experiments. One of the strengths of hybrid analysis is that it combines multiple analysis methods to detect suspicious activities within system applications. Besides, it generates malware information in real-time while conducting a dynamic analysis. Also, it consists of a security mechanism that runs untrusted and untested codes and programs mostly from unverified third parties, websites, suppliers, and users, without jeopardizing the well-being of the operating system and host device. In isolation, dynamic and static analysis is subjective to a high cost of development and manual analysis. However, combining the two techniques yields a hybrid model, such as ONAMD, with the capacity to achieve 87.83% accuracy [150] when using the SVM algorithm. ONAMD used permissions, package information, content providers, broadcasters etc. as static features and control flow graphs as dynamic features extracted from 600 applications, 248 of these applications were benign while 288 were malware.

One of the main tools used for hybrid analysis approach is the sandbox method which is a combination of dynamic and static elements [49]. The sandbox technique utilises APK file

static analysis where aspects such as user permissions, and manifest.xml files are reviewed and analysed for suspicious codes. In the dynamic phase, an Android emulator is used where the application is run in a controlled environment, and analyses are carried out as the application is executed. The emulator is designed to detect anomalies in application behaviour. Some of the defects may have been identified at the static stage, depending on the family of malware. In cases where the static step fails to detect any malware due to code obfuscation etc., it can then be detected in the dynamic phase of the process when the malware executes the malicious obfuscated code or sends unauthorised network traffic [50]. To better understand the behaviour and functionality of an application, the dynamic phase may check elements such as network traffic and native calls which are possible malware targets [51].

The Andrubis framework, on the other hand, uses the result of static analysis to guide the dynamic analysis. The presence or absence of malicious patterns after a static analysis informs the decision to carry out extensive dynamic analysis [53]. For example, the result of the static analysis may provide all possible entry points for an application, and this information may be used during dynamic analysis to execute the simulation events. Once the analysis of the bytecode and manifest.xml file is completed, the data generated is used for the dynamic analysis where simulations, taint analysis, or system-level analysis is carried out. Based on the complexity and amount of data generated by the static analysis, the dynamic analysis phase may even be longer. Different algorithms can be formulated where data from the two stages are complementarily used to identify malicious patterns and behaviours.

With the current complexity and sophistication of Android malware, multiple level hybrid approaches have been developed to aid in the detection and neutralisation of such threats. Static, dynamic and hybrid methods are either performed on-device or off-device. On-device approaches yield quick results after analysis, although supportive hardware resources may be limited [31]. However, mobile devices which are the primary users of Android platforms are limited with regards to resources; hence hybrid analysis is limited. The limited resources make

off-device more effective where analysis is remotely done with adequate resources. Multilevel hybrid techniques such as SAMADroid have been proposed to overcome these challenges [64]. The method utilises different levels of detection, which increases the probability and accuracy of detection. For instance, the first level may comprise of static and dynamic analysis while the second entails local and remote host analysis which identifies the behaviour of applications in different platforms. The third stage of the multi-level hybrid analysis requires a machine learning intelligence approach where data generated is incorporated in the learning model that further complements the analysis process.

5.5.4 Summary

In this section, we attempted to answer the first part (techniques) of the second research question: what is the current state of analysing Android malware detection techniques? We have identified and reviewed three primary techniques: Static Analysis, Dynamic Analysis, and Hybrid Analysis.

Static Analysis technique applies a signature-based and permission-based method to extract features from a set of benign and malware samples without running the applications. The extracted features are, but not limited to, permissions, intents, method names, string variables, package information, and text mining etc. These extracted features are passed as input to machine learning or deep learning algorithms to classify applications as benign or malware. The static analysis technique consumes fewer resources and efficient in detecting malware. Our survey indicates that static analysis has achieved up to 96% accuracy. However, the static analysis uses signature-based method so are prone to the limitation of the signature-based method, e.g. difficulty in detecting obfuscated malware. This limitation may be overcome if the analysis uses non-code-based features such as permissions, intents, and package information which are resilient to obfuscation due to Android's development model. Another limitation is that the technique is unable to extract features from dynamic code.

Dynamic Analysis technique extracts features from an application by executing them in a controlled environment. The method extracts features such as API calls, CPU usage, memory

usage, battery power. It may extract network traffic features such as IP address, length of packet lengths etc. The technique is also useful in detecting obfuscated malware as it does not depend on code analysis. The feature may be fed to various machine learning and deep learning algorithms to obtain a classification score against different metrics. Our survey indicates that the dynamic analysis has achieved up to 96.24% accuracy, which is slightly better than the static analysis (96%). A limitation of the dynamic analysis technique is that it requires a lot of resources to employ as applications need to be run in a controlled environment. Another limitation is that it fails to detect intelligent malware which detects a sandbox environment and does not execute malicious code.

Hybrid Analysis technique, as the name implies, consist of static and dynamic analysis. There are mainly three steps for hybrid analysis, employing static analysis, next executing applications in a controlled environment for conducting dynamic analysis, and finally feeding features extracted from both analysis to machine learning and deep learning algorithms. Our survey indicates that the hybrid analysis has achieved up to 96.92% accuracy, which is marginally better than static analysis (96%) and dynamic analysis (96.24%). Theoretically, combining both techniques should increase detection accuracy significantly; however, our survey shows that this is not the case. The main limitation of the hybrid technique is the complexity of combining both static and dynamic analysis, while marginally increasing overall accuracy. This complexity could be the reason for the decline in the usage of the hybrid analysis from 2017 onwards, as shown in Figure 10 in the Trends in Analysis Techniques Usage section below.

5.6 Technologies for Android Malware Detection

This section attempts to answer the second part (technologies) of the second research questions by discussing the state of Android malware custom-built detection technologies. Whether static, dynamic, or hybrid, multiple custom-built frameworks of Android malware detection have been developed and proposed. They are based on various algorithms which keep on changing and improving to handle new forms of malware [151]. These technologies

also attempt to counter the limitations associated with previous versions or have a unique element that can efficiently detect malware from multiple or specific families. These technologies may also have different versions based on continuous improvements and regular updates. Based on the approaches used, detection and security frameworks have various complexities which are also determined by processes, resources used and the extent of databases. In this section, we shall discuss these technologies:

CrowDroid – This technology utilizes dynamic analysis in a real-time situation through a machine learning model. It recognizes Trojan-like malware in Android systems such as Android-based mobile devices. The machine learning model in CrowDroid analyses the number of times individual system calls within an Android system are issued by an application when executing an action that requires user authorisation [6]. It is designed to detect an anomaly when user authorisation patterns are suspicious. The K-means model is used to classify observations for malware analysis. One of the primary differences, when compared with other frameworks, is that authorised features are monitored in the cloud.

Burguera, Zurutuza and Nadjm-Tehrani [6] describes CrowDroid as a machine learning-based framework used to detect Trojan-like malware from Android applications. In an experiment to demonstrate the functionality of CrowDroid model, a sample size of 10 malware and 50 benign applications was obtained from 20 clients. The framework utilised modified files and system call features. The model successfully differentiated the benign and malware applications based on the presence of Trojan at a 100% success rate. However, the experiment was done on only 60 applications, so it is not known how it would behave when a bigger sample size is used. The other limitation of CrowDroid identified was that it was not easy to convince the clients to adopt it due to privacy issues, i.e. private data needed to be accessed and stored for analysis [6].

TaintDroid – This framework utilises a dynamic approach to malware detection, whereby it aims to monitor malware that may facilitate the leakage of sensitive information. It is implemented on the Dalvik Virtual Machine, which identifies signs of information leakage

based on a machine learning model which categorises data based on the sensitivity [152]. Private information which is tainted is tracked based on anything that receives or reads the data, hence facilitating the detection process. Users of Android systems are notified every time private data leaves their devices.

According to Enck et al. [152], Taintdroid is a malware detection technology used to analyse network traffic to detect anomaly behaviours from Android applications. The framework enables real-time analysis by leveraging on a virtual execution environment. To demonstrate the used of TaintDroid, Enck et al. [152] studied 30 popular third-party applications with access to sensitive and confidential data from the internet. Out of the 30 evaluated applications, the study established that 20 of them misused user's private information at a confidence level of 95%. However, the sample size used in this study was relatively small (30 third party applications), making it difficult to generalize the study results. Although TaintDroid effectively tracks sensitive application data, it causes false positives if the tracked information contains configuration identifiers. Besides, it is incapable of tracking information once it leaves the phone. The experiment also shows that tracking incurs 14% performance overhead.

DroidOlytic – This technology is one of the premier Android detection frameworks that use the signature-based static technique. The technology creates malware signature database which is used to detect hidden malware in applications from third-party sources and suspicious suppliers. The statistical signature used in DriodOlytics is considered to be among the most robust in the detection of obfuscated and repackaged applications. Using the signature of known malware, DroidOlytics uses the statistical data to flag up applications with similar patterns, which are mostly used by malware architects. The concept of operation entails finding statistically similar regions with known malware to detect any variants from a sample.

In demonstrating its operation, Faruki et al. [153] tested a sample of 6151 benign against 1260 malware sample from 57 distinct families. From the stated sample, 251 obfuscated malware signature features were extracted. The results of the experiment demonstrated accuracy between 96.87% (third party dataset) and 99.4% (Google play dataset), depending on the

dataset used for the benign applications [153]. In testing obfuscated repackaged applications, the proposed approach attained 73% accuracy rate with a limited dataset; most of the obfuscated samples were missed since they had not been updated with new signatures.

Androguard – This framework uses a Python tool to compare applications with known malware in a database using clustering and similarity distance approaches. Malware architects may design and reverse engineer applications in a repackaged manner that resembles benign or genuine applications [154]. Androguard is one of the leading frameworks that detect this with high levels of accuracy. It is one of the primary applications used for similarity analysis where malware and benign applications can be distinguished despite apparent similarities as engineered by malware architects [127].

Chavan et al. [123] uses AndroGuard to filter malicious and broken application files from a dataset of 989 malware and 2657 benign samples using 230 distinct permission features. Specifically, the Androguard framework is used to reverse engineer application files based on the requested permissions. The identified applications are then classified using a binary architecture, that indicates whether the application asked each of the corresponding permission or not. From 230 permissions used, 118 were classified as malware. In total, 1260 applications were classified as malware using the framework with an accuracy level of 96% [123]. However, the technology is limited in that it only extracts permission features from the malware sample, implying that some malware from a benign sample may go undetected.

MigDroid – This technology uses an invocation graph technique to detect repackaged applications with Android malware. The technology reflects the interaction contention present in different approaches or detection methods [155]. The invocation graphs are exploited based on weaknesses that emerge from the difference between injected malware and genuine applications. The framework starts by constructing a method invocation graph on the smali code, which is then divided into graphs that are connected. The smali code is the human-readable format of the binary Dalvik bytecode of .dex file of Android APK. Malicious codes are determined after calculation of threat scores associated with each subgroup. It is considered

to be a complementary framework to other approaches such as Androguard which are not based on application repackaging.

The application of MigDroid framework for malware detection is illustrated in a study by Hu et al. [156]. MigDroid leverages on method invocation graph (MIG) based on statistical analysis to detect malware in repackaged applications. MIG detects malware by analysing the connection between injected malicious and benign applications. The experiment results from a sample of 1260 malware samples show that the framework attained 95.94% detection rate [156]. Some of the features used in the study include instruction code sequences, call graphs, API calls and application strings.

Dendroid – This framework utilises modern approaches such as text mining to detect malware as malware threats got sophisticated and increased in number. This Intelligent malware detection framework relies on text mining the code as well as methods of retrieving information [157]. The technology is based on a statistical analysis of data found in Android operating systems malware families. The modelling processes used in text mining are analysed and modelled when measuring similarity levels with selected and known malware samples.

In this technology, code structures (mined text) are used to compare samples and train the clustering algorithm and classification. The underlying Dendroid technology entails extraction of codes, features, and classification of the Android applications. The study by Suarez-Tangila et al. [157] applied the Dendroid technology to extract statistical features of code structures from a sample size of 1231 malware samples grouped into 33 families. Although the study does not indicate the actual accuracy level of the model, the results show that the model was accurate, scalable and fast in detection and classification of, malware [157]. The main setback of the study is that there are limited studies that use text mining to detect malicious codes, limiting comparative analysis on accuracy and performance [157].

EvoDroid – Mahmood, Mirzaei and Malek [158] presented EvoDroid technology. EvoDroid provides an evolutionary framework for evaluating Android applications capable of passing

genetic makeup of benign applications. The model operates by combining the Android-specific program analysis technique and the evolutionary algorithms noel technique [158]. The former finds the code segments which are amenable to be searched independently and the latter utilises code segments found in specific program analysis to create a stepwise search for achieving higher code coverage. Mahmood et al. [158] selected a dataset of 100 applications and performed a comparison with other technology such as DynoDroid and Monkey. Throughout the benchmarks, EvoDroid showed the code coverage of 98%, implying that it can be used effectively for cod analysis by other technologies for extracting features during static analysis. However, EvoDroid fails to generate models for applications that use native codes or third-party libraries.

SmartDroid – Zheng et al. [159] proposes SmartDroid framework which integrates dynamic and static analysis to reveal UI-based trigger conditions automatically in Android applications. Static analysis, in this case, is used to extract switch paths for call graph and activity features. In contrast, dynamic analysis is used to transverse UI elements based on the information acquired, such as Activities during the static analysis. From the experiment consisting of six applications, the method demonstrated automatically triggering UI events identified (72 intents and 54 expected paths) during the static analysis [159]. It is, therefore, more effective compared to other models such as TaintDroid for triggering UI based events during dynamic analysis. However, SmartDroid does not cover sensitive behaviours such as rooting which reduces its effectiveness in malware detection. Furthermore, it does not include UI triggers that depend on data received from the server or some configuration files [159].

MalDroid – This technology is regarded as a Machine Learning based detector for Android malware. This framework has been utilised in Android malware detection with high success rates, see details below. The model collects data on different families of malware and extract the features that are most likely to be affected by these families of malware [160]. However, due to its static approach, it may be unable to detect repackaged and obfuscated applications.

Torregrosa [160] presents MalDroid; a machine learning-based framework. The model detects malware by performing static analysis to gather as many features as possible from the Android applications. From a dataset of 2295 malware and 426 benign samples, 299 permissions including intents, permissions, API calls, risks and system calls, were generated. The extracted features were then used to classify the applications as either benign r malware. The experiment results illustrate a 99.8% accuracy rate [160] however, the dataset is highly skewed as the number of malware applications is just over 84% of the total samples, and no steps were taken to compensate for this skewness. Regarding limitations, MalDroid suffers from the setbacks of static analysis and fails to detect malware that evolves with time [160]. As a result, it must be supplemented by other technologies such as Androguard to improve its performance.

DroidRanger – This technology utilises imported packages, permissions, and API calls from applications as a way of detecting and classifying malware. The extracted features are compared with known malware where similar patterns and behaviours are identified [161]. The framework also focuses on system calls which are made from the native code in the Android system. It then looks for any patterns or attempts to hide this code, which indicates the presence or absence of malware.

Zhou et al. [162] presents DroidRanger to detect malicious applications on Android markets such as Google Play and four alternative markets. A total of 204,040 applications were downloaded from five Android markets; their results indicated that 32 applications downloaded from the official Android market were infected, corresponding to 0.02% infection rate. The applications downloaded from the alternative markets showed infecting rate ranging from 0.20% to 0.475%, as 179 applications were found to be malicious. Droid ranger uses permissions, packages, and API calls to create a signature of malicious applications. It also used behaviour-based detection by finding applications that did not have native code in the correct directory at phase one. In phase two, these applications were executed to classify them as benign or malicious. One limitation reported by authors was that they had focused

their research on free applications, and further study should be conducted to investigate paid applications as they offered unique differences in comparison with the free applications. However, they failed to explain the differences. Another limitation reported was the inability of DroidRanger to detect background SMS messages sent to unauthorised premium numbers.

Andromaly – This technology is a host-based detection system which progressively monitors Android devices feature and events. It relies on machine learning techniques and models which review user behaviours by watching 88 characteristics such as scheduler, CPU load, system calls, messaging, and battery power etc. to determine the patterns of these behaviours [163]. The monitoring results in the creation of a database with standard parameters that depict the absence of malware. Using several classifiers such as Bayesian Networks, K-means, and Logistic regression [164]. Users can be alerted once behaviours go beyond standard parameters, hence allowing further analysis which may indicate the presence of malicious applications.

Shabtai et al. [165] presented Andromaly model for malware detection in Android applications. The framework operates through a malware detection system which continuously monitors specific features and then uses machine learning to classify them as either benign or malicious. During the experiment, a total of 44 applications were used, including 40 benign and four malicious samples from which 300 API call and permission features were generated from each; based on their availability. The framework yielded 4% misclassification rate. However, the model was constrained in that it had limited ability to govern feature access rights based on the installation time [165]. Another limitation is that the dataset used was small and needed to be subjected to a bigger dataset for detecting its efficacy.

5.7 Discussion, Trends, and Research Gap

In this section, an analysis of trends for the surveyed study is provided, followed by a discussion of the identified gaps within the surveyed literature, which may assist in directing future research attempts within the area.

5.7.1 Trends in Analysis Techniques Usage

Based on the literature review result, it is evident that Android malware security has been accorded significant attention within recently published literature because of the extensive use of Android platforms. Moreover, critical trends since 2014 are observed as indicated by the literature review results.

Figure 9 below shows the popularity of each of these three techniques. It is evident from the graph below that the Hybrid approach is the most used technique, followed closely by the Static method, in the papers that we included in our survey. A significant number of articles have used machine/deep learning technique as evident in Figure 9 below, further breakdown of this complimentary technique is shown in Figure 11 below,

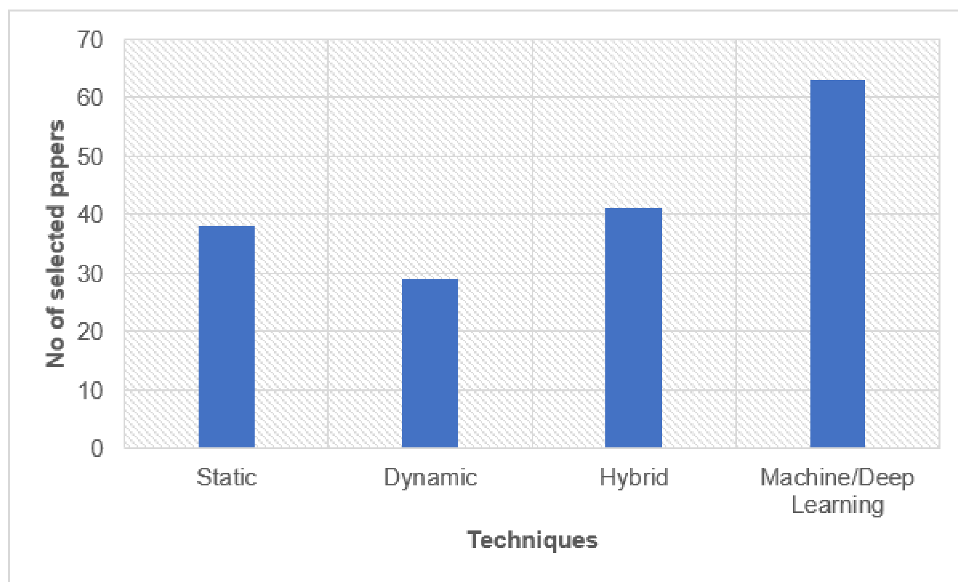

Figure 9: Malware Detection techniques by the number of surveyed papers

Figure 10 below provides a further breakdown of Android malware techniques for the last few years.

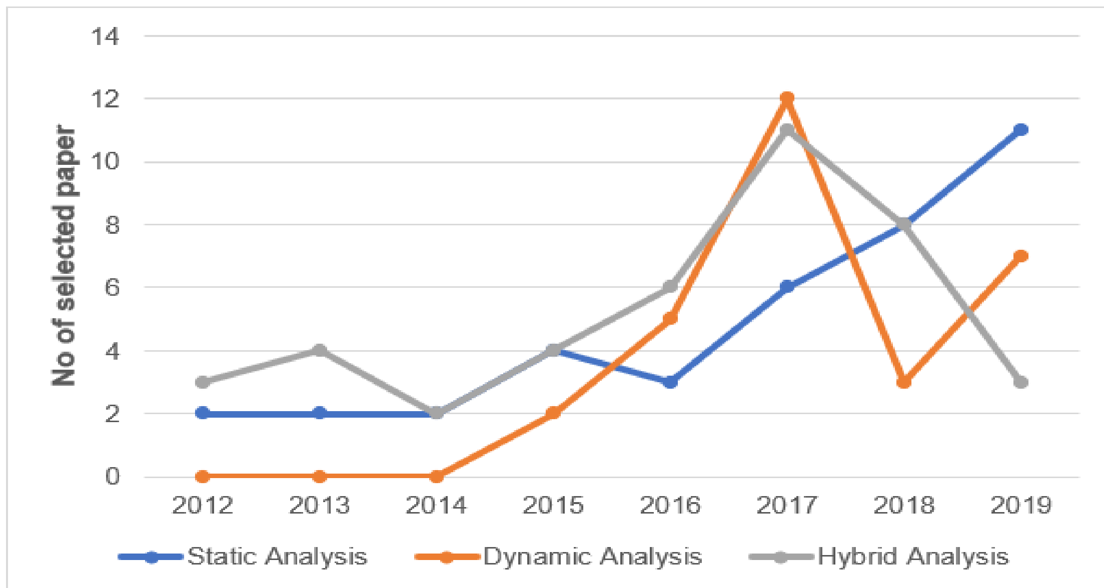

Figure 10: Trends in Analysis techniques referenced in surveyed papers

As indicated in Figure 10 above, the hybrid mechanism was dominant in security analysis within the Android malware. Additionally, the dynamic analysis mechanism has shown significant growth and is a widespread technique along with the static method. The primary assumption for the decline in the usage of hybrid technique could be the complexity involved in executing it, while not achieving significantly higher accuracy, as discussed in the Summary of Techniques for Malware Detection section above.

Figure 11 below shows trends in the usage of machine learning and deep learning as complementary approaches over the years. It can be observed in the chart that their usage has increased significantly over the last three years. One of the reasons for this increment could be the advancement in machine learning and deep learning algorithms, as the world embraces Artificial Intelligence.

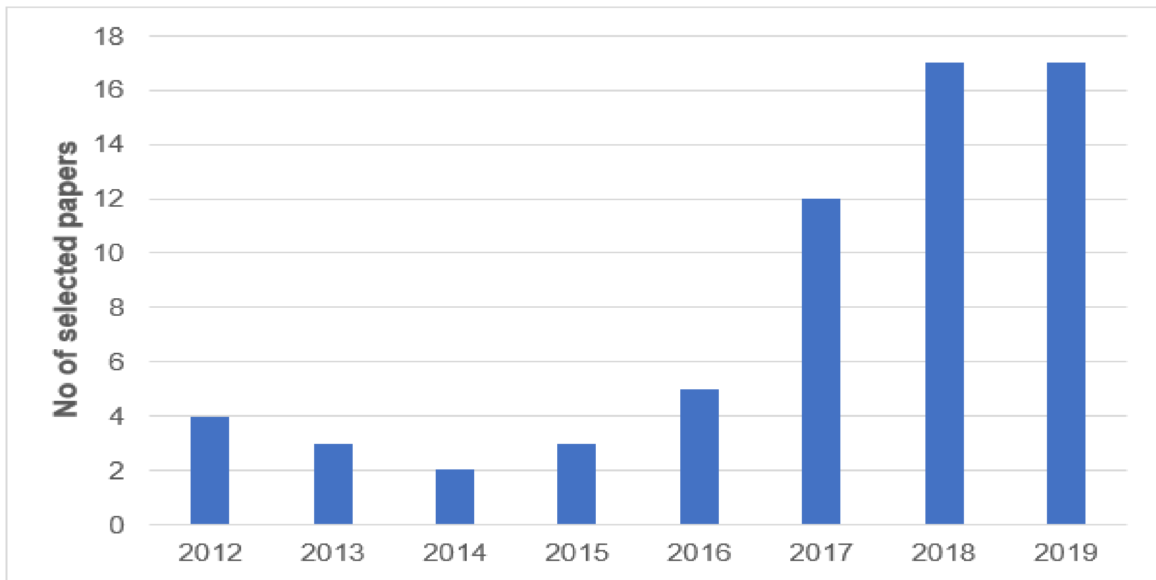

Figure 11: Trends in Machine/Deep learning techniques referenced in surveyed papers

Despite significant research attempts directed toward mitigating security risks within mobile platforms, a considerable rise in the quantities of security threats targeting the platforms is being witnessed [165]. Given this, the first recommendation involves increasing collaboration and convergence among investigators in the area of mobility, security, and software engineering to attain the common objective of tackling such mobile attacks and threats.

5.7.2 Research Gap in Android Malware Detection

Antimalware architects are in the process of developing new ways of detecting and preventing Android malware attacks which are different from those found in operating systems such as windows [165]. One of the most apparent gaps in research is reliable detection techniques that have the capability of detecting Android malware quickly and reliably. Sometimes, it takes long before an Android malware is detected, categorized, and neutralized. At times, it is only discovered after their malicious intentions have already been met [166]. There is no single detecting technique, either static, dynamic or hybrid that has the capability of guaranteeing reliable detection with 100% accuracy. Malware architects are well versed in malware detection techniques and are increasingly developing ways of avoiding exposure by utilising existing behaviour gaps. Even though some detection techniques are superior to others, e.g.

hybrid methods are more robust than dynamic and static, they all have their advantages and shortcomings [167].

The scope of Android malware has also become sophisticated and created gaps which have allowed the development and penetration of advanced malware as well. There are instances where more than one application or point of attack is used. The multiple application or point of entry attack is also orchestrated from different devices to complete a malicious process [21]. Applications which look benign and pass the detection stage are designed to work function with other similar applications, where they have severe impacts on affected devices. Once both seemingly benign applications are installed on the same device, they may communicate in the background via services for malicious purposes [16]. Therefore, the usage of services in any detection technology's feature set should minimise this threat. There is, therefore, a gap in detecting such applications and flagging them up.

Another research area that requires more investigation is where the seemingly harmless application gets installed as there is no malicious code detected during static analysis, these seemingly benign applications then download and install malicious code. Although dynamic analysis may protect against this technique, the static analysis does not offer any detection capability [43]. Therefore, the usage of detecting if dynamic code is used during the static analysis may provide some resilience to the static analysis against this type of threat. With the relatively new techniques of malware architecture, antimalware companies are always carrying out research and staying vigilant to develop new ways of dealing with the new generation of malware [139].

With an increased number of Android users [12], which continues to grow progressively, the vulnerability of users is, therefore significantly high. The gap in research with regards to this aspect is the vulnerability levels and the probability of users falling victims to malware attacks. Even though users may be aware that there are some forms of malware present in applications, they have no way of discerning unless they suspect malicious activities in their devices. The nature of malware attacks is also likely to be complicated in a manner that users

do not understand or notice. A further complexity is added by the fact that anyone with the ability can develop a sophisticated malware by collaborating on hacking forums on the dark web. The open Android market has, therefore, been associated with a further complication in the fight against Android malware [57].

From the vast amount of literature reviewed in this thesis, several research gaps have been identified, which may serve as a source of hypothesis for further research. Most of the studies discussed provide malware analysis techniques but do not provide step by step process on how to analyze the latest malware threats with the suggested tools [133]. Conducting extensive research to ensure a step by step malware analysis process would be valuable. Developing workflows (process taxonomy for identifying malware) to determine the root cause of malware behaviours and classify them into their respective families also provides an essential avenue for further research [146], [106], [105], [97], [98], [123].

Further, limited research has been done on the cost of developing malware systems which are less interactive with malicious artefacts, that is, systems that cannot be compromised by malware [134], [115]. Kumara et al. [134] highlight that most of the research studies focus on the development of new effective models but rarely provide the cost structures of creating such models. Conducting further research in this area can help Android software companies to identify the most effective and efficient malware detection systems. In Mehtab [124] and Fan [102], the findings illustrate that more research is required to determine how obfuscated malware can be analyzed by integrating different techniques, such as static analysis and machine learning, to enhance malware sequence analysis capacity. Such integration is also paramount in analyzing the behaviour of unpacked malware samples exposed to similar attacks as packed samples which often give the attacker a higher success rate [130], [105]. In a packed sample, malicious code is hidden through compression or encryption, making analysis difficult. Besides system integration, Sihwail et al. [141] and Fatima et al. [118] suggests further research to evaluate the effects of combining genetic and machine learning algorithms.

Sang et al. [135], Alpletkin et al. [99] and Martin et al. [149] suggest that future research works can consider evaluating advanced network architectures such as DenseNetor, AndroPyTool, and NasNet. Network architectures can design components of Android applications configuration, communication protocols and principles of operation and can help in devising more effective malware detection systems. The principles of configuration and operation define the underlying guidelines and rules for deployment in advanced network architecture for malware detection, e.g. active monitoring of network assets. Another gap identified concerns the evolution of malware and classification of unbalanced datasets [129], [168], [107], [95], [93]. Understanding the evolution of malware to create signature features to be used as vaccines for future malware detection is an important area that requires further research. Notably, most of the malicious artefacts evolve with time necessitating for efficient techniques facilitate detection. As they evolve, the capacity of the current detection systems is constrained. Further research in this area will help to create more flexible systems whose effectiveness is not affected in the long term.

Given the cumbersomeness, such as manual steps required from acquiring benign and malware samples to extracting features and storing results, more research is needed in the creation of automatic detection tools. Jiang et al. [125] suggest further research in the development of unsupervised and transfer machine learning in malware detection such as storing acquired knowledge for solving related problems and sharing it with the research community through automated APIs. This access to automated API will help to replace the current techniques, which are mostly manual and limited in real-time malware detection. In analyzing the most important features for use in malware detection [101], [148], it is also important to identify features that create 'noise' during the analysis process [143], [150]. This 'noise' prevents the achievement of optimal outcome in the detection and affects the accuracy levels of the models used. It is, therefore, a research area that requires further examination. Another critical area of research identified entails the determination of whether dynamic and static analysis technique can accurately classify applications when subjected to new code

morphing techniques [136], [169], [100]. The two models may be bypassed if the code is altered.

Table 4 below provides a breakdown of features used in publications. The table suggests that the most common features utilised in the literature are Permissions, Application components and System/API calls extracted during static analysis.

Table 4: Publications utilising the required features

Publication	Features
[2] [17] [19] [21] [36] [45] [46] [48] [52] [55] [58] [59] [61] [63] [64] [65] [66] [69] [70] [71] [75] [77] [82] [89] [92] [100] [101] [103] [107] [110] [111] [112] [114] [116] [118] [120] [121]	Permissions
[36] [37] [45] [48] [49] [60] [61] [64] [65] [77] [89] [107] [118]	Application Components
[6] [18] [19] [31] [34] [37] [39] [43] [44] [46] [48] [50] [51] [52] [53] [54] [55] [56] [59] [60] [61] [63] [64] [65] [66] [67] [70] [71] [74] [77] [78] [80] [91] [93] [94] [98] [99] [101] [103] [105] [106] [110] [111] [116] [121]	System/API Calls
[32] [56]	Power/Battery Consumption
[35] [48] [54] [56] [61] [71] [95] [96] [104]	Network Traffic
[37] [39] [49] [60] [68] [75] [99] [119]	Flow/Communication Analysis
[48] [58] [59] [93] [113]	Strings
[77]	Native Code
[115]	Image Patterns

The literature review suggests that different studies have used features such as permissions, API calls, system events, strings, and method names etc. in their static analysis. However, features such as services and the presence of useful functionality such as detection of code for cryptography, dynamic code loading, native code, HTTPs, database, and reflection have not been utilised in detail especially in same experiments. The cumbersome in extracting these features could be one of the reasons for less usage of these features. These features may be useful in detecting malware due to the following reasons:

1) Services allow an application to run a long-running background process and communicate with other applications, so it will be useful for malware to use them for malicious purpose.

2) Malware may use cryptography, dynamic code loading, native code, and reflection to avoid static analysis detection.

3) Malware may use HTTPs (secure communication) to send private information to a central server.

4) Malware may use local database temporarily on a local device before sending data to a central server.

The above gap in research can be fulfilled by creating an approach that amalgamates the above features for detecting Android Malware. The literature review also suggests that the process for extracting features from multiple malware and benign applications is not straightforward as it requires various steps. These steps are decompiling APK, obtaining the needed features, storing those features into some storage medium and feeding them to machine learning algorithms. There is a requirement to automate this process using a frontend interface. Taking all this into consideration, we developed Droid Fence which combines these neglected features (usage of services, cryptography, dynamic code loading, native code, HTTPs, database, and reflection) with forty permissions, and twenty services to prepare a matrix of sixty-six features. Nine machine learning and deep learning algorithms utilise these features for classification purposes.

Considering all the above, the following research questions can be identified which can help devise our research path. The results of our experiments in detecting Android malware must achieve over 90% accuracy whilst keeping FPR less than 3%. These results will conform to the machine learning methodology proposed by Soviani, Scheianu and Suciu [168] to aid in the detection and recognition of malware. The primary hypothesis of the study was that the technique used should be able to detect malicious applications before they cause damage. As such, the accuracy target for malware detection should be at least 90% with false alarm (FPR) not exceeding 3% for a classifier [168].

- **RQ4:** Is it possible to devise an efficient process for running experiments, decompiling the APK, obtaining the required features, and viewing and comparing the results?
- **RQ5:** Will the usage of neglected features identified in literature review along with Android permissions and services allow the achievement of over 90% accuracy whilst keeping FPR less than 3% in detecting Android malware?
- **RQ6:** How does the performance of the proposed deep learning algorithm (in terms of accuracy, F1 score, precision, and recall) compare to that of existing machine learning algorithms?
- **RQ7:** Does the approach developed as part of RQ5 performs better (in terms of Accuracy) than comparative methods?

The next two chapters will attempt to answer the above four questions. Chapter 6 – Droid fence Overview and Background on Features – will attempt to answer fourth research question and will provide an overview of our custom-built technology Droid Fence and how it works. The chapter will also offer a summary of the features that we are using in our experiment. Chapter 8 – Droid Fence Methodology & Performance Evaluation – will attempt to answer research question five, six, and seven by providing detail on methodology, our experimental settings, data set collections, our results, and the comparison with related methods.

6. Droid Fence – Overview and Background on Features

Droid Fence is a custom-built web-based framework for managing experiments. Droid Fence is developed to automate the extraction of the required features from malware and benign applications directory by conducting static analysis via a frontend. Once features are extracted, Droid Fence completes the automated process by storing the extracted features against each application record in a relational database, feeding them to the required machine learning and deep learning algorithms, storing the result into the database, and finally displaying the outcome of each experiment.

Droid Fence has contributed towards answering RQ4, RQ5, RQ6, and RQ7. In this chapter, whilst answering RQ4, an overview of Droid Fence and background on the selected features for experiments is presented. The remaining research questions RQ5, RQ6, and RQ7 will be addressed in the subsequent chapter Droid Fence – Methodology & Performance Evaluation

6.1 Overview

In this section, we attempt to answer the fourth research question.

- **RQ4:** Is it possible to devise an efficient process for running experiments, decompiling the APK, obtaining the required features, and viewing and comparing the results?

To answer the RQ4, we have developed Droid Fence, a framework for simplifying the steps involved in running experiments. There are different components that we built to achieve this goal. We will now provide an overview of Droid Fence architecture and outline the functionality of each of the components that are part of Droid Fence. Figure 12 displays an overall high-level design of the Droid Fence.

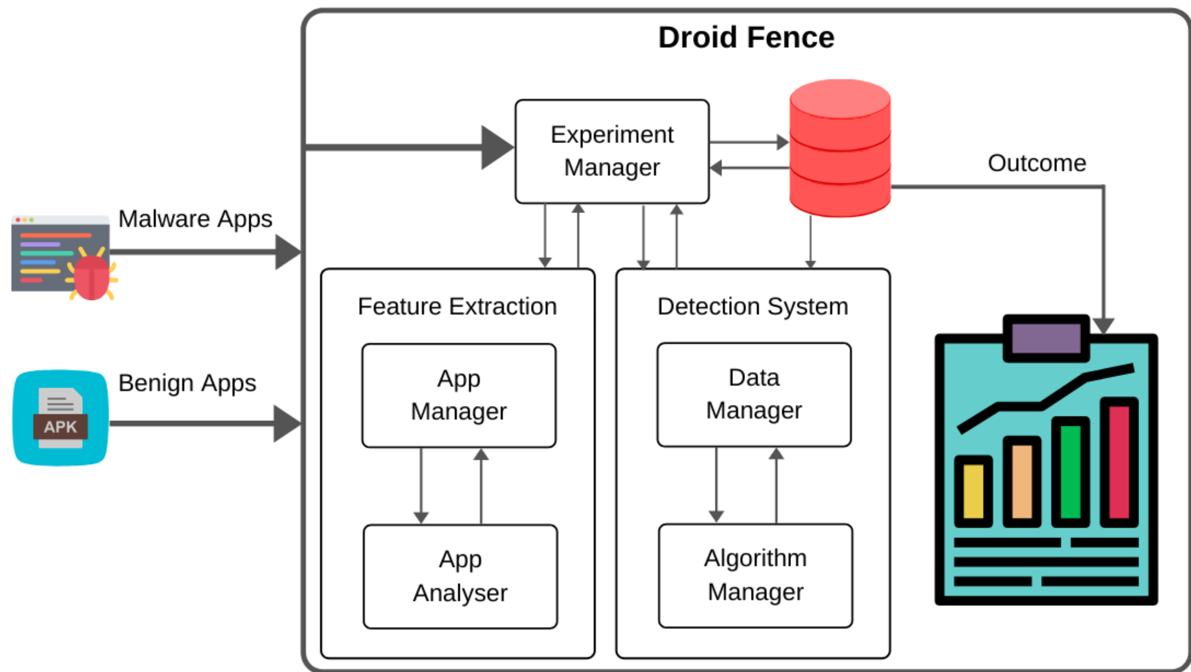

Figure 12: Droid Fence - Overview

There are four major components of Droid Fence: Experiment Manager, Feature Extraction, Detection System, and Database. The Experiment Manager component is responsible for managing each experiment, receiving malware and benign applications dataset, controlling flow between all components, and committing to the database. The Feature Extraction component analyses each application and extracts the required features and sends this information back to the Experiment Manager for storing in the database. The Detection System retrieves relevant features from the database for each application, trains machine learning and deep learning algorithms, evaluates them on our dataset against different metrics, and finally sends this information back to Experiment Manager for serialising into the database. The database is used for storing and retrieving features, dataset information, and classification results. The resultant outcome is viewed for each experiment via the Droid Fence interface.

6.1.1 Experiment Manager

The Experiment Manager (EM) is the first point of contact for commencing an experiment. An experiment can consist of one of three types: Feature extraction, Detection, and both. Feature extraction experiment, as the name suggests, is conducted to extract required features from

both malware and benign datasets and storing them into the database for utilising it in future experiments. EM uses two components for this purpose: Feature Extraction (FE) and Database. EM sends the dataset and required features to FE, which in turn analyses each application in a dataset and returns the required information to EM. EM then commits the relevant information into the database. The outcome interface does not show classification or detection rate for this type of experiment; however, it shows experiment details, applications and features extracted for each application via frontend.

Detection experiments are executed to employ dataset features which have already been stored in the database as part of Feature extraction experiments. EM employs the Detection System (DS) and a database for this type of experiment. EM sends the required information, such as features and the required algorithm names, to be employed in an experiment to DS, which in turn retrieves the relevant features from the database. DS converts the data into a suitable two-dimensional tabular data structure and feeds them to required algorithms for classification purposes and sends the result back to EM. Finally, EM commits the result into the database. The outcome interface shows detection rates and comparison for each algorithm as well as features employed in the experiment.

Droid Fence allows both of the above experiments to be conducted at the same time. For this type of operation, EM employs all four components; it first utilises FE component in the same way as it does for the Feature extraction experiment. Once Feature extraction returns control to EM, it serialises data into the database and executes the DS component for retrieving features and running classification algorithms. Finally, EM commits the data returned by DS into the database. The outcome interface displays features utilised in the experiment, detection rates, and comparison for each algorithm employed in the experiment.

6.1.2 Feature Extraction

The Feature Extraction (FE) component is utilised for traversing through a dataset and extracting the required features for each application. FE receives the dataset path and the requested features to be obtained from EM. FE employs two subcomponents to achieve the

desired functionality: App Manager (AM) and App Analyser (AA). The primary purpose of AM is to traverse through dataset path, pass each application to AA, temporarily store the analysis results, i.e., required features into a Python set, and finally return the analysis results to EM for committing into the database. AA receives an application as an input, and it utilises the Androguard tool [170] to disassemble and decompile the application in DEX files. Once an application is decompiled, AA uses Androguard API to extract the required features and returns this information to the AA subcomponent, which adds it into a python set. Once all applications have been analysed, AA returns the control to EM, which commits the features information into the database. Features supported by Droid Fence are permissions, services, presence of reflection, HTTPS, dynamic code, native code, cryptographic code, and database. The flexible design of the Droid Fence allows developers to extend its functionality for supporting additional features. Figure 13 shows feature information extracted by AA and displayed in the Droid Fence user interface.

XDA Assistant.apk

SHA256	5e6420c8c7713f96d5e502155379e299b1fe42e2f88c4804011716e9d729a46c
Package	com.lextel.ALovePhone
Malware	False
Uses Native Code	False
Uses Cryptic Code	False
Uses Reflection	True
Uses Https	False
Uses Dynamic Code	False
Uses Database	True
Permissions	android.permission.RESTART_PACKAGES, android.permission.READ_SMS, android.permission.READ_CONTACTS, android.permission.WRITE_SMS, android.permission.WRITE_CONTACTS, com.android.launcher.permission.INSTALL_SHORTCUT, android.permission.VIBRATE, android.permission.MOUNT_UNMOUNT_FILESYSTEMS, android.permission.CHANGE_WIFI_STATE, android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.SET_WALLPAPER, android.permission.WRITE_EXTERNAL_STORAGE, android.permission.ACCESS_WIFI_STATE, android.permission.READ_PHONE_STATE, android.permission.WAKE_LOCK, android.permission.CAMERA, android.permission.DISABLE_KEYGUARD, android.permission.CHANGE_NETWORK_STATE, android.permission.BLUETOOTH, android.permission.GET_PACKAGE_SIZE, android.permission.WRITE_SETTINGS, android.permission.CLEAR_APP_CACHE, android.permission.READ_LOGS, android.permission.BLUETOOTH_ADMIN, android.permission.KILL_BACKGROUND_PROCESSES, android.permission.BROADCAST_STICKY,
Services	com.lextel.widget.Widget_Service, com.lextel.ALovePhone.taskExplorer.widget.Taskexplorer_Widget_Service, com.lextel.ALovePhone.taskExplorer.widget.Taskexplorer_Widget_Service_2x1, com.lextel.ALovePhone.topApps.download.DownloadService,

Figure 13: Droid Fence Feature Extraction - Application info

6.1.3 Detection System

The Detection System (DS) is employed by EM to retrieve dataset and features information from the database, prepare the data into a suitable format, execute machine learning and deep learning algorithms to detect malware for four metrics and return the result to EM. DS

accomplishes the above tasks with the help of two subcomponents: Data Manager, and Algorithm Manager. Data Manager (DM) ensures that the feature data retrieved is converted into a suitable format of zeros and ones, as shown in Figure 14. DM uses the 'DataFrame' class of pandas [171] to transform data into a two-dimensional tabular data structure. If a feature is present for an application, then it is represented by 1 else 0. DM drops the application name column before preparing data for the Algorithm Manager (AM) subcomponent.

android.permission.ACCESS_COARSE_LOCATION	android.permission.ACCESS_NETWORK_STATE	....	use_https	use_dynamic_code	use_database	class
0	1	....	0	1	1	0
1	1	....	0	1	1	0
1	0	....	0	0	0	1
1	1	....	0	0	1	1
1	1	....	0	0	1	0
0	1	....	0	0	1	0
0	1	....	1	0	0	0
0	0	....	0	0	0	0
0	1	....	1	1	1	0
0	1	....	0	0	1	0
1	1	....	0	0	1	0

Figure 14: Droid Fence - Data Preparation

AM subcomponent receives a dataset from DM and feeds it into eight machine learning and one Sequential (Deep Learning) algorithms. The flexible design of the Droid Fence ensures that developers can extend its functionality to support additional algorithms. AM uses a stratified k-fold method to train and validate data for classification purposes (more details can be found in section 7.1 Experimental Settings below). AM stores the classification result in Python dictionaries as temporary storage until all algorithms have been evaluated. The results are stored for four metrics: Accuracy, F1-Score, Precision, and recall.

Furthermore, a confusion matrix is generated and stored. Once all results are acquired, the control is returned to EM, which commits results into the database. The outcomes of the experiment can be viewed in Droid Fence (examples of graphs and confusion matrix are shown in section 7.8 Experiment 2 - Performance Comparison below).

6.1.4 Database

Droid Fence uses relational database MySQL [172] for storing experiment details, applications information, features, and experiment results. Figure 15 depicts an Entity-Relationship (ER) diagram for thirteen tables and their relationships.

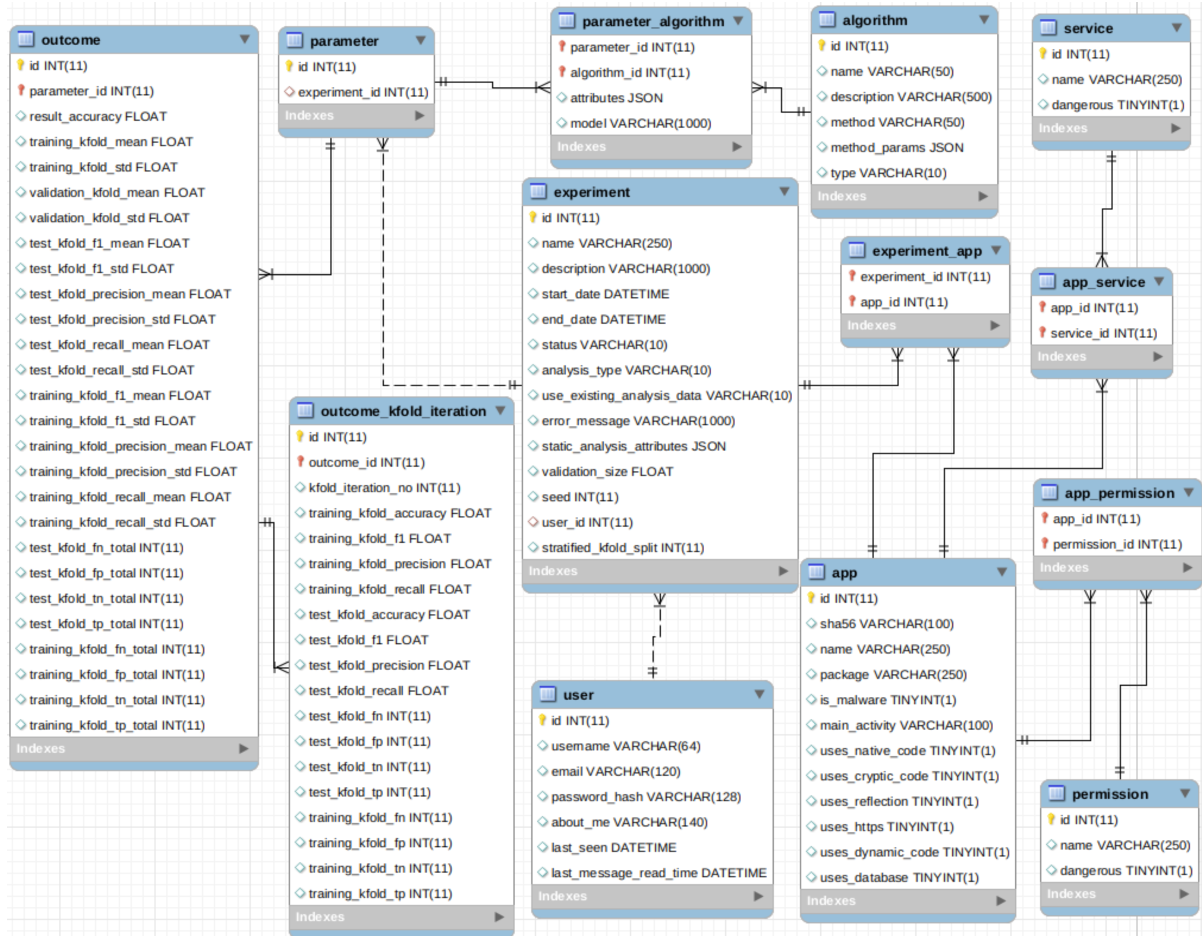

Figure 15:Droid Fence - Database ER Diagram

We are displaying only those tables which store the required information for each experiment and its associated data. Some tables which do not store experiment related data have not been shown for brevity. Table 5 below describes the purpose of each table used in the Droid Fence database. The flexible database design ensures that Droid Fence allows developers to extend its functionality, such as increasing the number of features or algorithms used for evaluating malware and benign dataset while providing a comparison of each algorithm in a

relatively easy manner. This flexibility ensures that if the requirement changes in future, Droid Fence can be amended to incorporate the new requirements.

Table 5: Droid Fence - Database Tables Descriptions

Database Table	Description
algorithm	A lookup table to store all algorithms supported by Droid Fence. It saves six machine learning and one Sequential (Deep Learning) algorithms.
app	A lookup table to store all applications across datasets. The table is linked to app_service, app_permission, and experiment_app tables.
app_permission	A linked table to store all permissions used by each application. The table is related to the app and permission tables.
app_service	A linked table to store all services used by each application. The table is related to the app and service tables.
experiment	A table to store information related to the experiment, e.g., analysis type, parameters used, etc. The table is linked to the user, experiment_app, and parameter tables.
experiment_app	A linked table to store applications that are used in each experiment. The table is related to the experiment and app tables.
outcome	A table to store the outcome for each algorithm in an experiment. The table is linked to parameter and outcome_kfold_iteration tables.
outcome_kfold_iteration	A table to store the outcome for each kfold iteration for each algorithm in an experiment. The table is linked to the outcome table.
parameter	A linked table to help combine the outcome for each algorithm in an experiment. The table is linked to parameter_algorithm, experiment, and outcome.
parameter_algorithm	A linked table to store algorithms and their parameters used in the experiment. The table is related to an algorithm, experiment, and outcome tables.
permission	A lookup table to store all permissions extracted from all applications across datasets.
service	A lookup table to store all services extracted from all applications across datasets.
user	A table to store user's information, including credentials, who may be running using Droid Fence after logging in.

6.2 Droid Fence - Background on selected features

In this section, we give a background on features that we selected for utilising in Droid Fence. We will also clarify the distinction whether specified features were extracted by Droid Fence using its code or Androguard API. We will also discuss how Droid Fence feature extraction can be deceived by malware developers. In later sections, we describe in more detail how these features are extracted. We have used a total of 66 features consisting of the usage of forty top permissions, twenty top services in our dataset, presence of reflection, dynamic code loading, native code, database, HTTPS, and cryptographic code.

6.2.1 Permissions

The privacy of an Android user is secured by using the permission system built into the Android operating system. The permission system consists of four different protection levels: Normal, Signature, Dangerous, and Special Permissions. Normal permissions are granted at install time and must be requested in the application's manifest file if an application requires to access resources or data outside the application's sandbox. These permissions do not post a significant risk to the operation of other applications or user's privacy, e.g., permission to setup up a time zone [173]. The signature permissions are granted at install time if both applications – the one which defines permissions and the one that uses those permissions – are signed by the same certificate. Dangerous permissions are requested to access users' private information or operations that may potentially affect other applications, e.g., the permission to access a user's call logs is dangerous, and users must explicitly approve the permission. Applications must prompt users to grant dangerous permissions at run time. Special permissions consist of sensitive permissions that do not behave like dangerous and normal permissions. The two primary examples of sensitive permissions are SYSTEM_ALERT_WINDOW and WRITE_SETTINGS. If an application needs to use these permissions, it must declare that in a manifest file and sends an intent at runtime, prompting the user's approval. [173]

An application declares the permission it requires by using permission tags in the application manifest, e.g., if an application needs to read users' contacts, then it must declare it, an example declaration is shown in Figure 16. Droid Fence has utilised Androguard API to extract the permissions from manifest files for each application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.myapp">
    <uses-permission android:name="android.permission.READ_CONTACT"/>
    <application ....>
        ....
    </application>
</manifest>
```

Figure 16: Android Manifest - Permission Declaration

Permissions are classified into different groups depending on the device's features or capabilities. Android handles permissions at a group level; therefore, if an application declares single permission belonging to a group, all permissions of that group will be declared in the application's manifest. For example, if any permission in an 'android.permission-group.CONTACTS' is declared and subsequently approved by a user then all permissions belonging to the permission group are granted. In this scenario, if 'READ_CONTACT' is granted, then the remaining two permissions of 'android.permission-group.CONTACT' group – 'WRITE_CONTACTS' and 'GET_ACCOUNTS' – are also sanctioned. [173]

More information on the Permission features used by the Droid Fence in our experiment is discussed in section 7.5 Features below.

6.2.2 Services

Services provide a facility for an application to execute a long-running process without constant interaction with a user. Services use the application's main thread, so it is imperative to start service in a secondary thread to avoid 'application not responding' errors if service is to perform any CPU intensive or blocking operations. Service is started by calling

'Context.startService()' function, which allocates resources to keep service running until it stops itself or someone else stops it. [174]

Services also provide a means for interacting with other applications. An application can expose some of its functionality to other applications by allowing connectivity to its service. A long-standing connection can be approved by calling 'Context.bindService()' method to utilise the exposed functionality of an application. [174]

An application must declare its intention of using services in a manifest file by using the 'service' declaration tag. An example declaration of 'MyService' is shown in Figure 17. Droid Fence has utilised Androguard API to extract services from manifest files for each application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.myapp">
    <application ....>
        <service android:name=".MyService" />
        ....
    </application>
</manifest>
```

Figure 17: Android Manifest - Service Declaration

More information on the Services features used by the Droid Fence in our experiment is discussed in section 7.5 Features below.

6.2.3 Reflection

Reflection is a useful concept of object-oriented programming as it allows access and inspection of interfaces, classes, fields, and functions at run time. Furthermore, it permits dynamic invocation of functions and instantiation of new objects, which allow greater flexibility at run time, if types of objects are not known at compile time. Another useful application of reflection is to allow developers to decide at runtime, if a required functionality (functions or classes) is available before using it, thus providing a solution for backward compatibility for older devices. In Android, 'java.lang.reflect' and 'java.lang.class' packages provide classes

which can be used to utilise reflection in an application, within the security constraints of Android.

Figure 18 below shows an example code in Android that can be used to access a class and its methods at run time. Exception handling is omitted for brevity.

```
Class myClass = Class.forName("android.text.Html");
Method[] allClassMethods = myClass.getDeclaredMethods();
for(Method aMethod : allClassMethods )
{
    // Do something
}
```

Figure 18: Android - Reflection Example Code

Droid Fence utilises Androguard API to extract strings, classes, and all methods from each file. Next, Droid Fence uses its App Analyser (AA) module to traverse through the extracted data to detect if any reflection packages or methods are being used by the application. The Droid Fence feature extraction component sets ‘use_reflection_code’ value to true if it detects the usage of reflection in an application.

6.2.4 Dynamic code loading

Android provides a dynamic code loading technique to allow developers to load code that exists outside of their application code. This facility will enable developers to use dynamic updating functionality, code reuse, reduce start-up time, and acquire extensibility, which ultimately helps them to build modular and leaner applications.

An abstract class ‘ClassLoader’ in the ‘java.lang’ package is responsible for loading classes dynamically. The class loader tries to find the class’s definition data or generates it by converting the name of the class into a file name and then reading it from a file system. [175]

Figure 19 shows example code for dynamically loading class files by a network class loader from a server. Two methods ‘findClass’ and ‘loadClassData’ are defined by a subclass

‘NetworkClassLoader’ that are used to download the binary data that constitute the class and then create a class instance using the ‘defineClass’ method [175].

```
ClassLoader loader = new NetworkClassLoader(host, port);
Object main = loader.loadClass("Main", true).newInstance();
     . . .

class NetworkClassLoader extends ClassLoader {
    String host;
    int port;

    public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassData(String name) {
        // load the class data from the connection
         . . .
    }
}
```

Figure 19: Android - Dynamic Class Loading [175]

Droid Fence utilises Androguard API to extract classes, strings, and all methods from each file. Next, Droid Fence uses its App Analyser (AA) module to traverse through the extracted data to detect if any dynamic class packages or methods are being used by the application. Droid Fence feature extraction component sets ‘use_dynamic_code’ value to true if it detects the usage of dynamic code in an application.

6.2.5 Native Code

Applications for Android devices can be developed using different ways. Android SDK is used widely for application developments, which are classified as high-level developments. However, Android also allows developers to write native code application development specific to different processors using the Native Development Toolkit (NDK). Native code applications are less portable, more complex, and more likely to have memory corruption errors [176]. Despite these challenges, native code applications are created by developers for processor-bound applications as native code can increase performance for processor-intensive tasks. The enhanced performance is achieved due to three main factors: [177]

a. Native code runs directly on the operating system because code is translated to a binary code instead of being compiled into Java byte code, which needs to be interpreted by Dalvik Virtual Machine.
b. Native code allows accessibility to Single Instruction Multiple Data (SIMD) technology, which allows multiple data to be executed in parallel; this utilisation of processor features is not available at Android SDK.
c. Native code allows developers to access code at the assembly level and optimise it.

Droid Fence utilises Androguard API to extract classes, strings, and all methods from each file. Next, Droid Fence uses its App Analyser (AA) module to traverse through the extracted data to detect if any native class packages or methods are being used by the application. The Droid Fence feature extraction component sets 'use_native_code' value to true if it detects the usage of native code, otherwise false.

6.2.6 Database

The database is an ideal medium for storing structured data. Android provides 'java.sql' and 'android.database' packages, which can be used for interaction with different databases.

Figure 20 displays a sample code for creating and querying an in-memory database.

```
String query = "select sqlite_version() AS sqlite_version";
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(":memory:", null);
Cursor cursor = db.rawQuery(query, null);
String sqliteVersion = "";
if (cursor.moveToNext()) {
    sqliteVersion = cursor.getString(0);
}
```

Figure 20: Android - Database Access [178]

Droid Fence utilises Androguard API to extract classes, strings, and all methods from each file. Next, Droid Fence uses its App Analyser (AA) module to traverse through the extracted data to detect if any database class packages or queries are being used by the application.

The Droid Fence feature extraction component sets 'use_database' boolean to true if an application uses a database and false otherwise.

6.2.7 HTTPs

HyperText Transfer Protocol Secure (HTTPS) is a protocol for communicating securely over the internet or any other network. Most applications require communication to be conducted over the internet. Applications may use encryption technology, i.e., Transport Layer Security (TLS) or formerly known as Secure Sockets Layer (SSL) to encrypt data transfer between the client application and the server. Android provides classes in 'java.net' and 'android.net' packages, which can be used for secure communication.

Figure 21 displays an example code for using secure connectivity if a certificate is issued by a widely known Certificate Authority (CA).

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Figure 21: Android - HTTPS Connectivity [179]

Droid Fence utilises Androguard API to extract classes, strings, and all methods from each file. Next, Droid Fence uses its App Analyser (AA) module to traverse through the extracted data to detect if any class packages or methods linked to URLConnection are being used by the application. The Droid Fence feature extraction component sets the 'use_https' field to true if it detects the usage of TLS protocol.

6.2.8 Cryptographic code

Mobile applications may obscure part of code or resources by using cryptographic features of Android SDK. Android provides access to classes in 'java.security' and 'javax.crypto' packages. These packages can be used to encrypt sensitive information. Figure 22 displays a sample code for encrypting a message.

```
byte[] plaintext = ...;
KeyGenerator keygen = KeyGenerator.getInstance("AES");
keygen.init(256);
SecretKey key = keygen.generateKey();
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] ciphertext = cipher.doFinal(plaintext);
byte[] iv = cipher.getIV();
```

Figure 22: Android - Cryptography Code [180]

Benign applications may protect sensitive information such as user credentials, messages, or documents by using cryptographic facilities provided by Android. Malware may use cryptography to obscure SMS short codes, methods' signatures, or variables names. Droid Fence utilises Androguard API to extract classes, strings, and all methods from each file. Next, Droid Fence uses its App Analyser (AA) module to traverse through the extracted data to detect if any cryptography or encryption class packages or methods are being used by the application. The Droid Fence feature extraction component sets the 'use_cryptic_code' field to true if it detects the usage of cryptographic APIs.

6.2.9 Feature Extraction Deception

Droid Fence extracts the required feature without executing an application, therefore it is prone to the limitations of static analysis. One of the limitations of static analysis is that it cannot scrutinize dynamically loaded code because the code is downloaded at runtime. Malware developers can use this information and use a combination of Reflection, Native Code, Database, HTTPs, or Cryptographic code in dynamically loaded code. Although Droid Fence can detect whether a dynamic code is used in an application or not, it will be unable to establish if the five features specified above are used or not. This is identified as one of the future directions in Android Malware Detection area in the last section 8.2 Future Work.

Permissions and Services features, that formed a bulk of our features, cannot be deceived in the same way as the five features identified above because these features must be declared in the manifest file of an application.

7. Droid Fence – Methodology & Performance Evaluation

In this chapter, we attempt to evaluation Droid Fence by presenting results of four experiments. We describe software and hardware components utilised in our experiments, the methodology, features, data collections, and ethical challenges in data collection. We briefly provide detail of the algorithms that we utilise in our experiments. Furthermore, we discuss and assess the effectiveness of the Droid Fence framework for detecting malware. Finally, we also attempt to answer our fifth, sixth, and seventh research questions.

- **RQ5:** Will the usage of neglected features identified in literature review along with Android permissions and services allow the achievement of over 90% accuracy whilst keeping FPR less than 3% in detecting Android malware?
- **RQ6:** How does the performance of the proposed deep learning algorithm (in terms of accuracy, F1 score, precision, and recall) compare to that of existing machine learning algorithms?
- **RQ7:** Does the approach developed as part of RQ5 performs better (in terms of Accuracy) than comparative methods?

7.1 Experimental Settings

The experiments were executed in a virtual environment to sandbox our environment. The underlying hardware consisted of Intel Core i7-6700HQ CPU @2.60GHZ and 32 GB RAM. The virtual environment was setup using Oracle VirtualBox version 5.2.16 r123759. The virtual machine was assigned four virtual CPU, 16GB RAM, and 32MB video memory. The operating system of the virtual machine was 64-bit Ubuntu 18.04.1 LTS.

Droid Fence is implemented using Python v3.6, Flask v1.02, jinj2 v2.10, and HTML 5. It utilises pygal v2.4 for displaying charts and MySQL database for storing experiment results. The other required modules are Androguard v3.2.1, Scikit-learn v0.20, Tensorflow v1.13.1, and Keras v2.2.4.

7.2 Methodology

Droid Fence uses eight machine learning and one deep learning algorithms. Each of the algorithms is evaluated by using a stratified k-fold cross-validation method, and each fold is divided into 80% training and 20% validation set. Stratified k-fold is a variation of the k-fold method, as it shuffles data initially and then splits into n_splits (5 folds in our case), this ensures that classes are correctly shuffled, and there is no overlap among each fold. For each fold, 20% of data is kept as a validation set, while the model is trained on the remaining 80% data set. The trained model is evaluated on the validation set, the result is retained, and finally, the model is discarded. The process is repeated five times for each iteration; this approach ensures that each fold is incorporated at least once in a validation set and four times in a training set. The concluding model measures the result by averaging the outcomes derived at each fold.

7.3 Machine Learning Algorithms

In this section a short description of the algorithms used in our experiments is provided. Figure 23 below displays an image from the Droid Fence presenting parameters used for our experiment. The figure also provides the settings and configuration details of each algorithm.

Experiment ID	221
Experiment Description	Top 40 permissions, 20 Services, Native code, Cryptography, Reflection, Https, Dynamic code, and database. Five Kfold iteration with cross validation
Date & time	Start: 2019-12-15 05:38:45 End: 2019-12-15 05:44:46
Status	SUCCESS
No of Apps	12904
Analysis Type	Static Analysis
Use Existing Analysis Data	Yes
User	admin
Static Analysis Features for ML and Deep Learning Algorithms	Permission, Service, Uses Native Code, Uses Cryptic Code, Uses Reflection, Uses Https, Uses Dynamic Code, Uses Database
Machine Learning and Deep Learning Models	**Model**
	LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100, multi_class='warn', n_jobs=None, penalty='l2', random_state=None, solver='lbfgs', tol=0.0001, verbose=0, warm_start=False)
	LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None, solver='svd', store_covariance=False, tol=0.0001)
	KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=None, n_neighbors=13, p=2, weights='uniform')
	DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best')
	GaussianNB(priors=None, var_smoothing=1e-09)
	SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)
	sequential_1 (Layers=2 Denses1(200, input_dim=66, activation=relu) Dense2(10, activation=relu) Output_Layer(1, activation=sigmoid))
	RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini', max_depth=None, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None, oob_score=False, random_state=None, verbose=0, warm_start=False)
	XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=3, min_child_weight=1, missing=None, n_estimators=100, n_jobs=1, nthread=None, objective='binary:logistic', random_state=0, reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None, silent=None, subsample=1, verbosity=1)

Figure 23: Experiment Details

7.3.1 Logistic Regression

Logistic Regression may be used to predict a binary outcome. A binary outcome, as the term suggests, correspond to an event which has either taken place or have not taken place. The dependent variable in logistic regression is usually binary and can be used to classify various objects into groups based on the available data [181]. In this scenario, the algorithm can be called Binary Logistic Regression [166]. LR aids in establishing a probability that an input belongs to one of the two classes [182].

A logistic regression model works in five key steps. First, the independent variables or the inputs are provided. The inputs can be binary or continuous variables. If they are binary variables, they need to be coded usually in 0's and 1's to label them appropriately for data analysis. The independent variables, which are also the predictors are combined linearly in order to produce the logarithm of the odds. The log odds are then converted to probabilities using a logistic function. The resulting probabilities are usually between 0 and 1 with 0 and 1 being an indication of certainty on one side or the other.

While logistic regression is mainly applied in situations with a binary outcome, it also supports multinomial and ordinal results. Multinomial Logistic Regression is also used to predict more than two outcomes. It is akin to Logistic Regression; however, you can have more than two possible outcomes in MLR. An example of MLR is predicting the most used transport system in a given year. The possible outcomes could be bus, planes, car, and train. Another simple example of such compound events is the rolling of a die. It is different from a coin toss, which only has two possible outcomes [80]. In the case of ordinal results, the logistic regression classifier can be extended in order to handle more variables, but the result is usually an ordered set of categories. Interested readers can find more detail about the LR algorithm in an excellent book by Bonaccorso [183].

In our case, we need to predict if an app is a malware or benign, so it is a binary classification problem. We have used 'sklearn.linear_model.LogisticRegression' module from Scikit-learn v0.20 to implement Logistic Regression. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.2 Linear Discriminant Analysis

Linear Discriminant Analysis is a dimensionality reduction technique used for classification problems. The algorithm classifies outcome by projecting the spaces in high dimension onto lower dimension in such a way that maximises the separability of different categories. The algorithm seeks to project the large space of features in a small subspace. [166] For example, consider a dataset that must be separated into two classes. In this case, the categories are binary such as yes or no, male or female, success and fail among others. However, the classes are multidimensional in that they have different features that characterize them and only selecting a single feature to make categorical decisions might result in some elements overlapping [185]. Consequently, LDA facilitates the classification of a dataset with multiple attributes into two classes without overlapping problems. In this regard, by increasing the number of features to be considered during classification, the likelihood of overlaps reduces.

LDA mainly relies on two primary criteria in creating categories. One criterion is to maximize the distance between the means of the two groups and the other is to minimize the variation within the group. The technique achieves the two goals using three key steps. First, the separability of the classes is calculated by computing the difference between the classes. Second, the distance between the mean and the samples of each class is established. Last, LDA is then used to determine a lower dimension whereby the distance between classes is the highest and the one between elements of the same class is reduced.

However, while LDA is useful in cases where the means of the two classes differ, it is usually not effective when they are the same. In this regard, the technique cannot be used to establish a category that will make the classes separable. Therefore. LDA is useful for data classification

and dimension reduction, but it has its limits, when working with datasets that have equal means. Interested readers can find more detail about the LR algorithm in an excellent book by Bonaccorso [183].

In our case, we have used 'sklearn.discriminant_analysis.LinearDiscriminantAnalysis' module from Scikit-learn v0.20 to implement Linear Discriminant Analysis to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.3 K Nearest Neighbor Classifier

K Nearest Neighbor is used for both classification and regression problems. KNN classifier seeks to predict class for the validation data by calculating the Euclidean distance between the training data and the validation data. It uses the distance between the test data and the training data to identify the correct category for the test data [181]. In regression, the algorithm uses the mean value of the selected training data points.

KNN classifies data using 6 main steps. The first step is to provide training data with known classifications. The classifications are represented in clusters based on their individual characteristics. The next step is to select the number K of neighbors for the test data. The next step involves providing the test data and calculating the distance between the test data and the K number of neighbors. The next step involves selecting the nearest K neighbors based on the calculated distances. The data points for the nearest K neighbors are then counted. The test data is assigned to the neighbor with the highest data points closer to the test data. For example, given a data point with two categories of data, one can find the best fit for the data point using KNN. The value K represents the data points from the training data that should be counted from the nearest neighbors based on their distance from the training data.

Selecting K is a subjective process that is based on observation of the available training data. Heuristic techniques may be applied for large sets of data but general observation is suitable

when small sets of data are involved [181]. However, there is a key consideration that should be observed when selecting the value of k. For instance, choosing a small value for K can cause unstable decision boundaries [186]. Therefore, K should be large for classification decisions to reduce noise while also small enough to maintain distinct boundaries between the classes. Further, an even number as a K value when classifying test data with an even number of categories might result in a similar number of data points, thereby making the decision difficult in case of tied votes [181]. Interested readers can find more detail about the KNN algorithm in an excellent book by Bonaccorso [183].

In our case, KNN is used for classification purposes. We have used 'sklearn.neighbors.KNeighborsClassifer' module from Scikit-learn v0.20 to implement KNN to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.4 Decision Tree Classifier

A decision tree classifier is a predictive modelling strategy that is used in machine learning and data mining. DT can be used for both regression and classification problems. It relies on the use of a decision tree to make conclusions about the elements in a dataset [187]. DT is a binary tree that recursively splits the dataset until we are left with pure leaf or terminal nodes i.e. the data with only one type of class. The topmost node in DT is called Root node. The splitting is a process of dividing a node into two or more sub nodes. When a sub nodes splits into further sub nodes, it is called decision node. The training dataset is used to create a decision tree [182].

The decision tree algorithm makes binary decisions based on the attributes of the validation data and the training data. The algorithm traverses the tree, created as part of the training, for validation dataset to classify the sample. It starts from the root node and compares the attributes of the dataset and training data. Based on the identified path the algorithm follows

a branch from the root node and continues selecting specific branches until it reaches a leaf node.

For example, consider a hiring manager who wants a job candidate based on certain attributes. The criteria include attributes such as the level of education, work experience, candidate's performance during the interview, and salary requirements among others. Using a decision tree algorithm, the hiring manager can select the best candidate who fulfills all the required characteristics by categorising the applicants into different categories. First, the manager can examine the salary requirements and select those who fit within the category. Next, the manager can then select the candidates based on their work experience. Those with the necessary experience can then be categorised based on their level of education. The manager can then examine the level of education for the candidates and select those who have attained the required skills. Finally, those with the relevant education can then be categorised according to their performance in the interview and the best candidate throughout the selection process is likely to become employed. Thus, the decision tree algorithm involves classifying data based on criteria that enable the algorithm to produce a root, decision, and leaf nodes. Interested readers can find more detail about the DT algorithm in an excellent book by Bonaccorso [183].

In our case, DT classifier is used for classification purposes. We have used 'sklearn.tree. DecisionTreeClassifier' module from Scikit-learn v0.20 to implement DT to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.5 Gaussian NB

Gaussian NB is a classification and clustering algorithm derived from Bayes' theorem. Gaussian NB classifiers are usually characterized by high scalability. The algorithm assumes that each feature present in a sample is irrelevant to the presence of any other feature. The algorithm utilises prior knowledge of the sample data and classes to predict classification of

an event. The algorithm calculates the probability of each feature and makes the classification decision based on Bayes theorem. [182]

Guassian NB usually respond well to data that does not fit in memory [181]. Guassian NB classifiers are also characterized by the assumption of a feature's conditional independence. However, such independence is usually not reflected in the real world, which makes the algorithm naïve. For example, the height and weight of a person, which is used to calculate their BMI, are a determinant of the development of lifestyle diseases such as diabetes, and cardiovascular conditions among others. In this case, the higher the weight of a person and the lower their height, the larger the BMI and the likelihood of becoming diabetic. However, when using a Gaussian NB to predict the probability of developing such conditions, the algorithm will consider the three variables independently. The algorithm uses calculus to predict the probability of a certain category and the chosen classification is the one with the highest. Naïve Bayes algorithms mostly deal with categorical variables, as such, they are easier to classify since they do not have experience fluctuations and in terms of their amount [181]. The Gaussian NB specializes in dealing with continuous variables. The algorithm also assumes that the variables are normally distributed. Thus, the Gaussian NB is a classifier that categorizes continuous variables using the principles of the Bayes theorem. Interested readers can find more detail about the Gaussian NB algorithm in an excellent book by Bonaccorso [183].

In our case, we have used 'sklearn.naive_bayes.GaussianNB' module from Scikit-learn v0.20 to implement Gaussian NB to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.6 Support Vector Classifier

Support Vector Classifier is used for both regression and classification problems. SVM algorithm seeks to take data points and create a hyperplane that separates the classes in the

best possible way. The objective of the hyperplane is to minimise the classification error and maximise the marginal distance between classes. The classification is performed by identifying the hyperplane whereby two classes can be differentiated by maximum margin [182]. SVC selects the extreme vectors or data points, which are called support vectors, hence the name of the algorithm. The support vectors are then used to determine the hyperplane, which is then used to classify the data accordingly. The hyperplane relies on the support vectors to determine its position and angle of inclination when the model is plotted on a graph. For example, consider a dataset containing the characteristics of a cat and dog. The SVC algorithm will pick an extreme case of a cat and an extreme case of a dog and then develop a hyperplane during training [181]. When a new dataset is presented that needs to be classified, the algorithm will then use the support vectors, or the extreme cases to determine the category of each dataset as either a cat or dog. Such problems are encountered in image processing and malware detection among others and SVC can assist in categorising large amounts of data. Interested readers can find more detail about the SVC algorithm in an excellent book by Bonaccorso [183].

In our case, SVC classifier is used for classification purposes. We have used 'sklearn.svm.SVC' module from Scikit-learn v0.20 to DT to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.7 XGB Classifier

Extreme gradient boosting (XGB) classifier is a machine learning algorithm that relies on regularizing gradient boosting. XGB is derived from gradient boosting, which is a machine learning method that is applied in classification among other tasks [167]. Gradient boosting produces a prediction model that is presented as a collection of weak learners that are typically presented as decision trees.

In machine learning, decision trees are usually built stepwise based on the prediction of the training data. Boosting involves adjusting the next prediction based on the results of the current step. However, there is usually an error that occurs between the predicted and the actual values. Gradient boosting attempts to minimise the error, also known as the loss function, by gradient descent algorithm, which involves the use of a first order partial derivatives of the error. XGB classifier improves upon gradient boosting by adding various software and hardware enhancements.

First, XGB classifier works similar to a gradient boosting but instead uses the second order partial derivatives of the error. In addition, XGB also incorporates advanced regularisation through, lasso and ridge regression (L1, L2), which enhances the generalisability of the results of the prediction model. XGB also demonstrates sparsity awareness and allows sparse inputs by learning the best missing value. Furthermore, among weighted datasets, XGB uses the weighted quantile sketch to identify split points [167]. The XGB algorithm further supports parallelization and efficient use of hardware resources such as caching awareness and out-of-core computing that enhance disk usage during processing.

In our case, XGB classifier is used for classification purposes. We have used 'xgboost.XGBClassifier' module from Scikit-learn v0.20 to implement DT to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.8 Random Forest Classifier

Random forest classifiers are a group of machine learning techniques that are mainly used for classification among other tasks including but not limited to regression. The random forest algorithm utilises ensemble learning, which involves the use of several classifiers to solve a complex problem [188]. The algorithm uses the decision tree algorithm to predict the outcome of the model. A key advantage of the random forest over the decision tree is its ability to eliminate the problem of overfitting. Overfitting in machine learning refers to the predicted

statistical model fitting the training data exactly, which increases the errors. Reducing overfitting helps to increase the precision of the statistical model.

In order to understand the random forest classifier, it is important to examine the decision tree algorithm. A decision tree consists of branch/decision nodes, leaf nodes, and a root node. The algorithm divides a dataset into branches iteratively until a leaf node is formed, which cannot be divided any further [188]. The decision nodes are usually divided by a simple decision based on the attributes of the dataset that the algorithm is examining.

The difference between the decision tree algorithm and random forest classifier is the randomisation that occurs in the latter. In random forest, the root node and segregating nodes are established randomly. Further, the random forest classifier utilises the bagging method, which involves using multiple sets of training data. The different datasets produce different statistical models, which are ranked and the category with the highest number of occurrences is selected. During classification, the decisions made by the algorithm are categorised on a decision tree. The leaf node represents the final output of a certain decision. In order to select the final output for the classification model, the algorithm uses a multi-voting approach to select the decision chosen by most decision trees. Interested readers can find more detail about the SVC algorithm in an excellent book by Bonaccorso [183].

In our case, Random Forest classifier is used for classification purposes. We have used 'sklearn.ensemble.RandomForestClassifier' module from Scikit-learn v0.20 to implement DT to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Scikit documentation [184].

7.3.9 Sequential (Deep Learning)

Deep Learning is a subset of Machine Learning. It operates like Machine Learning, technically speaking, however it uses different approaches and capabilities. Deep Learning mimics the human brain when solving a problem whereas traditional Machine Learning algorithms use

binary logic to resolve a problem. Neural networks are a key aspect of deep learning. They are based on the current understanding of the human brain and how it works. Neural networks are able to learn complex relationships among various elements in both linear and non-linear outputs. A neural network is usually composed of an input, hidden, and output layer. Each layer contains several nodes also known as neurons. However, some neural networks have several layers and are known as deep learning networks.

Keras is a python library for developing as well as evaluating deep learning models [189]. It is free and open source, which makes it widely available to most people. It provides two strategies for building the neural networks of a model, sequential and functional. In a sequential model, the Keras layers are in a linear composition. The model is advantageous because it is minimal and represents nearly all neural networks. It also facilitates the creation of custom and complex models. Thus, sequential deep learning Keras is a python library that facilitates the creation of complex neural networks.

In our case, we have used 'keras.models.Sequential' module from Keras v2.2.4 to implement Sequential algorithm to predict if an app is a malware or not. The parameters and configuration settings that we used are specified in Figure 23: Experiment Details. Interested readers can find more information including definition and purpose of each parameter in Keras documentation [190].

7.4 Evaluation Metrics

Following metrics are used to evaluate the performance of our experiment.

- **Accuracy:** The Accuracy is defined as the ratio of the total number of correct predictions and the total number of predictions i.e. the proportion of all apps that are correctly classified as malware. The formula for calculating accuracy is:

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

- **F1 Score:** The F1 Score is the harmonic mean of the precision and recall. A perfect model has F-Score of 1. The formula for calculating F1 Score is provided below:

$$F1\,Score = 2 \times \frac{Precision\, \times\, Recall}{Precision + Recall}$$

- **Precision:** The Precision is defined as the proportion of correct true positive predictions of all positive cases. In our case, the proportion of malware detection to the total number of actual malware. The formula for calculating Precision is given below:

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

- **Recall:** The recall is the measure of how accurate the model is in identifying true positives. In our case, how many actual malware were recalled (found) by the model. The formula for calculating Recall is given below:

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

7.5 Features

We used 66 features in three separate experiments, consisting of 40 permissions, 20 services, the presence of specific functionality such as reflection, dynamic code, native code, HTTPS, database, and cryptography. We have selected the top 40 permissions used by all applications (both malware and benign) in our dataset. Table 6 lists permissions that we have used in our experiments. The malware and benign columns show the number of applications in our

dataset using the permission specified in the Permission column. The total column shows the number of applications (malware and benign) using the specified permission. If an application is using any of these permissions, then 1 else 0 is used in a tabular data structure, as shown in Figure 14 above, presented to AM subcomponent for classification purposes.

Table 6: Top Forty Permissions used in Experiment 2, 3, and 4

Permission	**malware**	**benign**	**total**
android.permission.INTERNET	5434	6201	11635
android.permission.ACCESS_NETWORK_STATE	3748	5871	9619
android.permission.WRITE_EXTERNAL_STORAGE	3805	4699	8504
android.permission.READ_PHONE_STATE	5021	2849	7870
android.permission.ACCESS_WIFI_STATE	2471	2759	5230
android.permission.WAKE_LOCK	2160	3047	5207
android.permission.RECEIVE_BOOT_COMPLETED	2715	2064	4779
android.permission.VIBRATE	1664	2542	4206
android.permission.SEND_SMS	3058	453	3511
android.permission.ACCESS_COARSE_LOCATION	1810	1561	3371
android.permission.ACCESS_FINE_LOCATION	1688	1543	3231
android.permission.READ_SMS	2126	625	2751
android.permission.RECEIVE_SMS	2206	540	2746
android.permission.READ_CONTACTS	1338	1298	2636
com.android.launcher.permission.INSTALL_SHORTCUT	1425	785	2210
android.permission.GET_ACCOUNTS	450	1630	2080
android.permission.CHANGE_WIFI_STATE	1016	916	1932
android.permission.GET_TASKS	762	1122	1884
com.google.android.c2dm.permission.RECEIVE	380	1499	1879
android.permission.WRITE_SETTINGS	688	1036	1724
com.android.vending.BILLING	44	1644	1688
android.permission.WRITE_SMS	1270	350	1620
android.permission.READ_EXTERNAL_STORAGE	355	1174	1529
android.permission.CALL_PHONE	764	658	1422
com.android.browser.permission.READ_HISTORY_BOOKMARKS	1025	313	1338
android.permission.CAMERA	253	1010	1263
android.permission.SYSTEM_ALERT_WINDOW	365	829	1194
com.android.browser.permission.WRITE_HISTORY_BOOKMARKS	930	201	1131
android.permission.RESTART_PACKAGES	764	323	1087
android.permission.WRITE_CONTACTS	551	492	1043
com.android.launcher.permission.UNINSTALL_SHORTCUT	809	216	1025
android.permission.CHANGE_NETWORK_STATE	414	588	1002
android.permission.READ_LOGS	525	438	963
android.permission.DISABLE_KEYGUARD	475	479	954
android.permission.SET_WALLPAPER	533	416	949

android.permission.BLUETOOTH	212	688	900
android.permission.INSTALL_PACKAGES	846	36	882
com.android.launcher.permission.READ_SETTINGS	644	233	877
android.permission.RECORD_AUDIO	132	635	767
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS	605	124	729

The second set of features that are used in all four experiments is the top twenty services in our dataset. Table 7 lists the services that we have used in our experiment. The malware and benign columns show the number of applications in our dataset using the service specified in the Service column. The total column shows the number of applications (malware and benign) using the specified service. If an application is using any of these services, then 1 else 0 is used in a tabular data structure, as shown in Figure 14 above, presented to AM subcomponent for classification purposes.

Table 7: Top Twenty Services used in Experiment 1, 2, 3, and 4

Service	malware	benign	total
com.apperhand.device.android.AndroidSDKProvider	608	5	613
com.airpush.android.PushService	552	6	558
com.google.android.gms.measurement.AppMeasurementService	0	555	555
com.google.firebase.iid.FirebaseInstanceIdService	0	519	519
com.google.update.UpdateService	399	0	399
com.google.android.gms.measurement.AppMeasurementJobService	0	229	229
com.google.android.gms.auth.api.signin.RevocationBoundService	0	220	220
com.google.analytics.tracking.android.CampaignTrackingService	0	220	220
com.software.application.C2DMReceiver	213	0	213
com.google.android.gms.analytics.CampaignTrackingService	0	212	212
com.android.view.custom.FirstAService	188	0	188
com.android.view.custom.SecondAService	188	0	188
com.android.view.custom.ThirdAService	188	0	188
com.google.firebase.messaging.FirebaseMessagingService	0	187	187
com.android.view.custom.FourthAService	186	0	186
com.google.android.gms.analytics.AnalyticsService	0	182	182
com.appbrain.AppBrainService	13	168	181
com.soft.android.appinstaller.services.SMSSenderService	153	0	153
com.senddroid.AdService	115	3	118
com.zanalytics.sms.SmsReceiverService	113	0	113

The final set of six features, utilised in three experiments, used by both malware and benign applications is shown in Figure 24 below. If an application is using any of these functionalities, then 1 else 0 is used in a tabular data structure presented to AM subcomponent for classification purposes. An illustration of tabular data is shown in Figure 14 above.

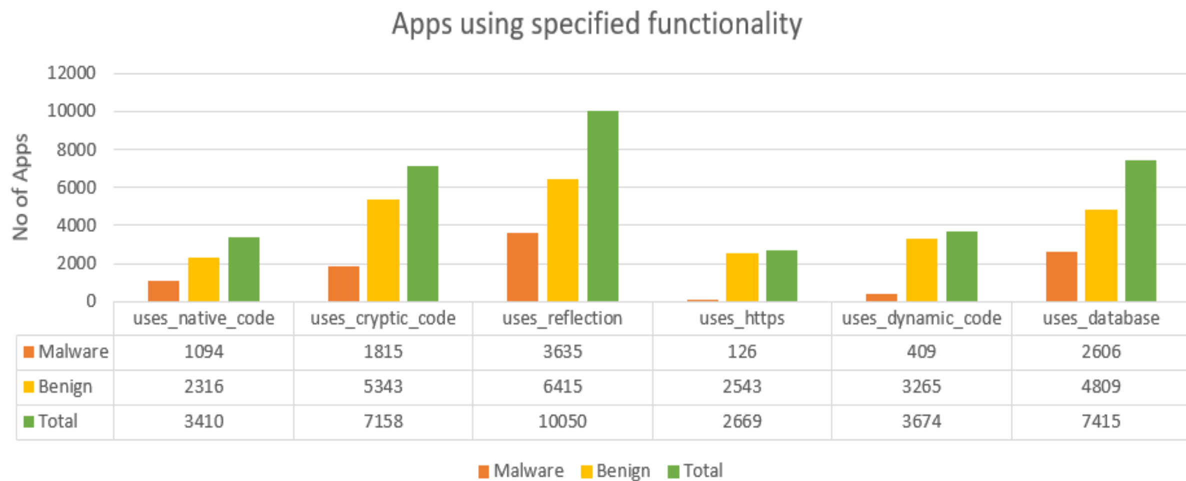

Figure 24: Six specified features used in Experiment 2, 3, and 4

The permissions that allow apps to access restricted data are classified as dangerous permissions [173]. As specified in section 6.2.1 Permissions above, certain operations that may potentially affect other applications, e.g., the permission to access a user's call logs is dangerous, and users must explicitly approve the permission. We utilised dangerous permissions and list of services specified in Table 7 above in Experiment 1 - Performance Comparison to establish if the usage of our neglected figures performs better or not. Table 8 lists dangerous permissions that we have used in Experiment 1 - Performance Comparison. The malware and benign columns show the number of applications in our dataset using the permission specified in the Permission column. The total column shows the number of applications (malware and benign) using the specified permission. If an application is using any of these permissions, then 1 else 0 is used in a tabular data structure, as shown in Figure 14 above, presented to AM subcomponent for classification purposes.

Table 8: Dangerous permissions used in Experiment 1

Permission	Malware	Benign	Total
android.permission.WRITE_EXTERNAL_STORAGE	3806	4698	8504
android.permission.SEND_SMS	3058	453	3511
android.permission.READ_SMS	2126	625	2751
android.permission.RECEIVE_SMS	2206	540	2746
android.permission.READ_CONTACTS	1338	1298	2636
android.permission.WRITE_SMS	1270	350	1620
android.permission.READ_EXTERNAL_STORAGE	355	1174	1529
android.permission.CAMERA	253	1010	1263
android.permission.WRITE_CONTACTS	551	492	1043
android.permission.RECORD_AUDIO	132	635	767

7.6 Dataset Collection

Automated data collection from app stores present ethical, legal, and technical issues. First, the Google Play store contains a lot of data, and an efficient approach to collect it is to use robots commonly known as bots. A web crawler is an appropriate bot for such an activity, as it can help scan as many applications as possible. However, the Google Play store was not designed for data collection using crawlers. Instead, it is meant as a web service where Android users can share mobile applications either for free or at a premium. While crawling for data is usually not explicitly mentioned on Google Play store's terms of service (TOS), it involves undertaking various activities that are prohibited on the platform. According to a study that attempted to implement a web crawler on Google store, the researchers performed various activities that were against the Google Play store terms of use [191]. For example, a user can have an unlimited number of email accounts that they can use on the Google Play store. However, they must belong to the owner. In addition, to ensure that a single user does not create an unlimited number of accounts in a short period Google restricts the creation of email accounts to five without requiring a phone number. Thus, to use a web crawler, such limitations must be overcome. The researchers circumvented the restrictions by crowdsourcing for Gmail accounts, which violates the terms of use [191]. Additionally, the researchers were also able to collect security keys from various applications, which exposed the data security of the users of such programs.

The ethical challenges in the situation are presented by the violation of Google Play store's TOS. The technical issue from data collection on Google Play store is the lack of a fool proof restrictions by Google Play to guarantee security from such activities as developers are able to circumvent the restrictions. As such, data collection on the Google Play store is against its TOS, however, we believe that data collection for research purposes is ethically right.

To evaluate our model, our data collection consisted of 13191 mobile applications divided between 5787 malware and 7404 benign samples. Malware samples were acquired from the Malgenome project [192], Drebin [193], and GitHub [194]. Benign samples were downloaded from the Google Play App Store [195] and Apps APK website [196] between November 2018 and August 2019. The Malgenome project and Drebin dataset contained malware samples collected between 2010 and 2012, whereas the GitHub sample contained malware collected between 2015 and 2019. There is a risk of benign apps containing malware app which could affect the performance of our algorithms. In order to mitigate that, we utilised API service of Virus Total to confirm that the benign samples did not include malware.

We performed four separate experiments on data samples. The purpose of conducting the first two experiments was to demonstrate whether the usage of our features is useful when using the same algorithms but different features. The purpose of the third experiment was to train our algorithms on Malgenome and Drebin datasets and test them on the GitHub samples which contained the samples from different time period. The fourth and the last experiment was conducted on all three datasets to measure the performance of our algorithms against a bigger dataset and compare it to the related methods published by researchers. The details of all experiments are provided in the next four sections.

7.7 Experiment 1 - Performance Comparison

In the first experiment, we used features specified in Table 7 and Table 8. These features are not the ones that we proposed for our final experiment. Droid Fence utilised eight machine learning algorithms and a deep learning Sequential algorithm in this experiment. The

performance of each algorithm is compared by observing accuracy, F1 score, precision, and recall metrics. Moreover, a confusion matrix for each algorithm is utilised for evaluation purposes. Figure 25 below shows an accuracy metric for training and validation data sets. The result shows that all algorithms have achieved accuracy score of 0.82 or above. Sequential (Deep Learning), Random Forest Classifier, and Decision Tree algorithms have performed very well against the training data set measuring accuracy score of 0.88. The same three algorithms also performed well on the validation data set, achieving an accuracy score of 0.87.

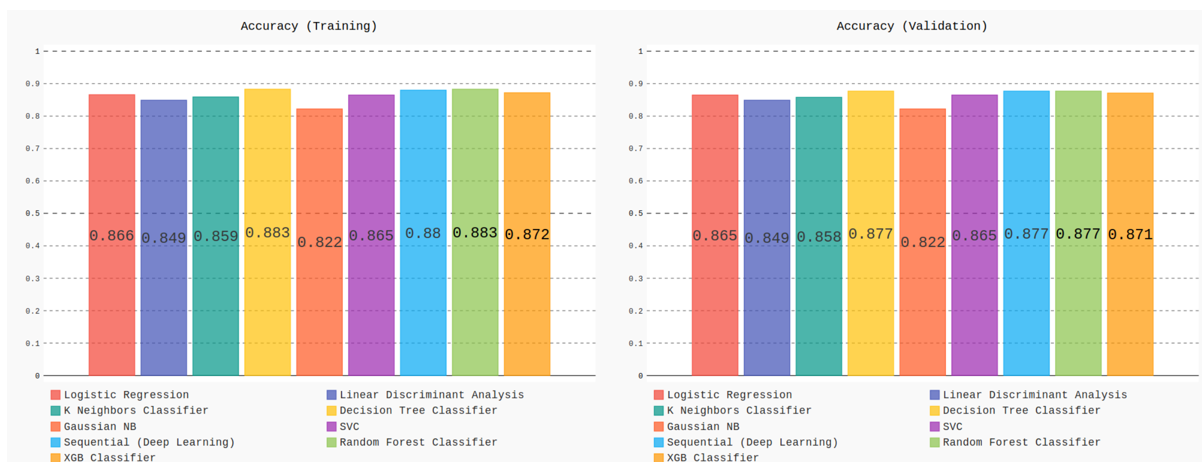

Figure 25: Accuracy Metric for Experiment 1 for Training and Validation Data Set

The XGB Classifier is marginally the second-best performing algorithm on the validation set, just ahead of the Logistic Regression. Gaussian NB has acquired a comparatively lowest score of 0.82. All algorithms have performed almost in the same fashion on both training and validation data sets.

Figure 26 below reflects each algorithm's performance when measured against the F1 Score metric. All algorithms except Guassian NB have achieved 0.80 scores for all but one algorithm. Random Forest algorithm is marginally the best performing classifier on the training data set, producing score of 0.853, followed closely by Decision Tree Classifier (0.852) and Sequential Deep Learning (0.846). The same three algorithms are the winners on the validation data set again, as they obtained F1 Score of 0.843 to 0.845. Gaussian NB appears last when measured against the F1 Score metric. Logistic Regression, Linear Discriminant Analysis, SVC, and

Gaussian NB have performed almost in the same fashion on both training and validation data sets also.

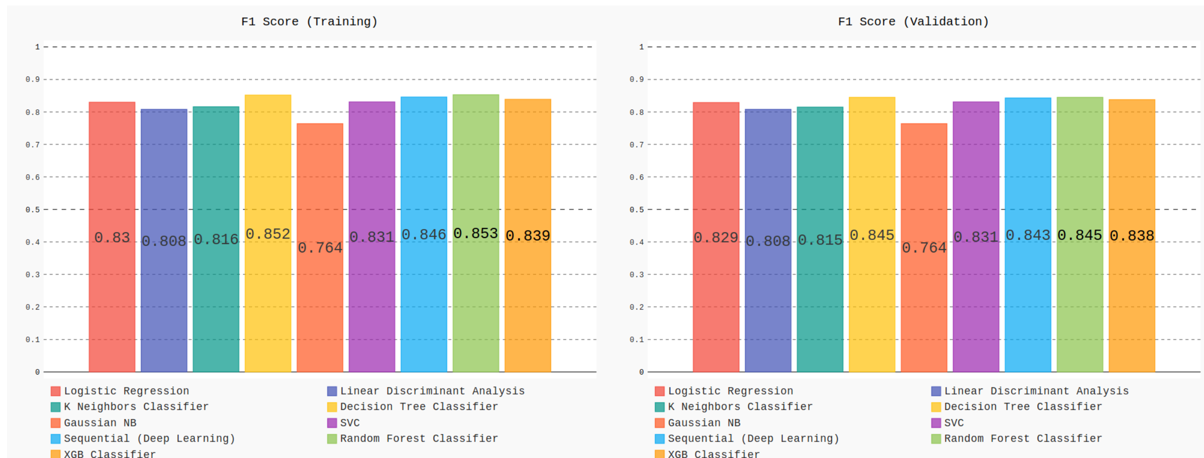

Figure 26: F1 Score Metric for Experiment 1 for Training and Validation Data Set

Figure 27 below illustrates the Precision metric results for all nine algorithms. All algorithms have acquired over 0.91 scores against the Precision metric. Both Sequential (Deep Learning) and Decision Tree Classifier algorithms have attained 0.959 and 0.957 scores respectively on the training data set, closely followed by Random Forest Classifier (0.95) and K Neighbours Classifier (0.954). The Sequential (Deep Learning) algorithm has attained the top position on the validation data set. K Neighbours Classifier is the second-best performing algorithm on the validation data set, marginally surpassing the Decision Tree Classifier. Gaussian NB is the worst performing algorithm in comparison to the remaining classifiers.

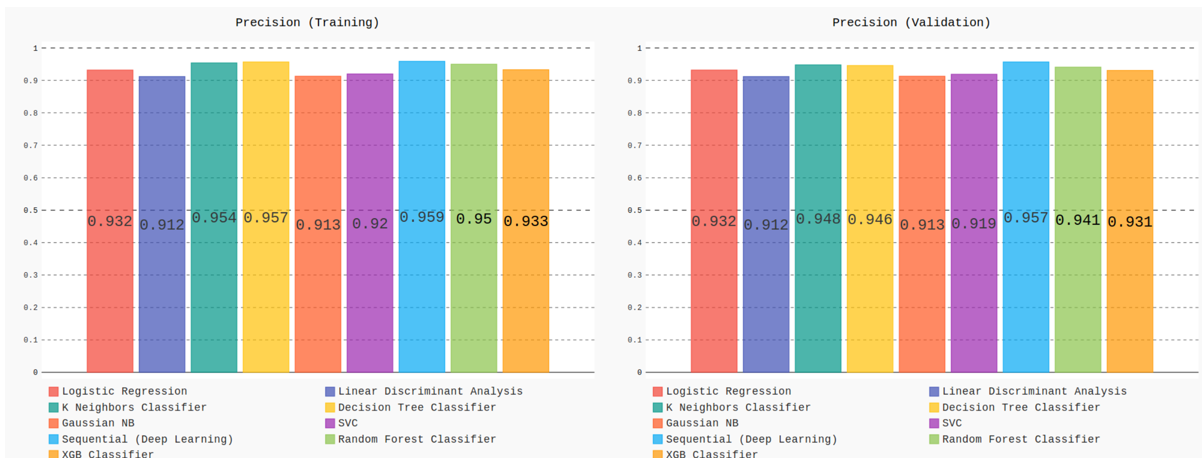

Figure 27: Precision Metric for Experiment 1 for Training and Validation Data Set

Figure 28 below reveals outcomes for all algorithms measured against the Recall metric. Eight out of nine algorithms have gained a score of higher than 0.71. The Random Forest Classifier has obtained the highest recall score on the training data set, followed closely by a Decision Tree Classifier. K Neighbors and Gaussian NB are the lowest-performing algorithms on both training – acquiring recall scores of 0.748 and 0.656, respectively – and validation data set – achieving recall scores of 0.747 and 0.656, respectively. The best performing algorithm on the validation data set is Random Forest Classifier, obtaining a recall score of 0.767.

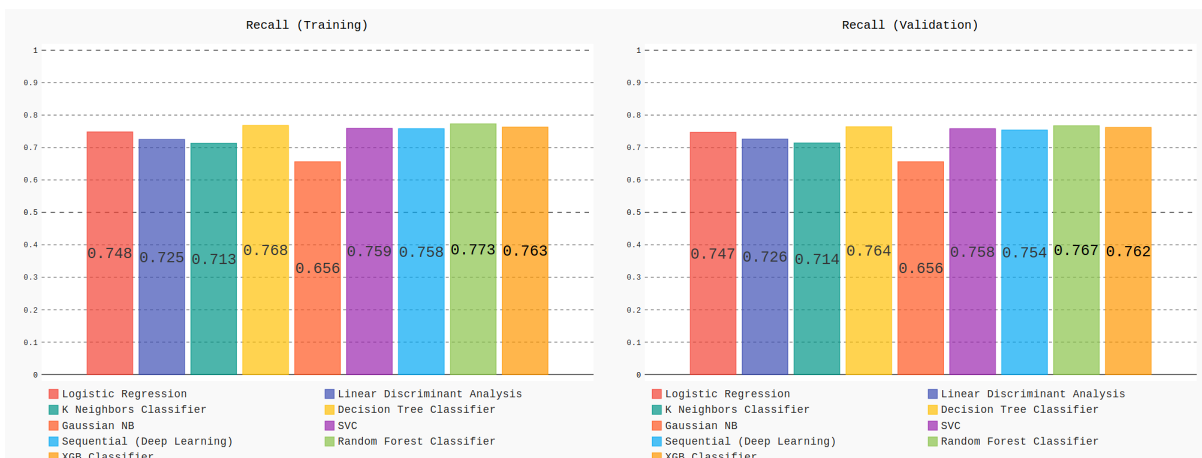

Figure 28: Recall Metric for Experiment 1 for Training and Validation Data Set

Table 9 below displays algorithms in the order of best performing to the worst. It summarises the result measured against accuracy, F1 score, precision, and recall metrics for each

algorithm applied to the validation data set. All algorithms have not performed well, achieving a score of less than 0.80 on twenty-six out of twenty-eight possible outcomes.

Table 9: Detection Result Comparison of Nine Algorithms on Validation Data Set

Algorithm	Accuracy	F1 Score	Precision	Recall
Sequential (Deep Learning)	0.877	0.843	0.957	0.754
Decision Tree Classifier	0.877	0.845	0.946	0.764
Random Forest Classifier	0.877	0.845	0.941	0.767
XGB Classifier	0.871	0.838	0.931	0.762
SVC	0.865	0.831	0.919	0.758
Logistic Regression	0.865	0.829	0.932	0.747
K Neighbours	0.858	0.815	0.948	0.714
Linear Discriminant Analysis	0.849	0.808	0.912	0.926
Gaussian NB	0.822	0.764	0.913	0.656

Sequential (Deep Learning) is the best performing algorithm as it obtained 0.957 against one metric and 0.843 or more against the two metrics. The recall rate is similar to most of the algorithms apart from Linear Discriminant Analysis which has the best recall rate. The top machine learning algorithm is the Decision Tree Classifier attaining 0.84 or more score on three of the four metrics when applied on the validation data set. The close second algorithm on machine learning family is Random Forest Classifier achieving 0.84 or more score on three of the four metrics. There is a marginal difference between the top three algorithms. Gaussian NB remains in the last position

We are using a confusion matrix for summarising the performance of all classification algorithms that we used in our experiment. The confusion matrix provides helpful insight into inaccuracies made by our algorithms as well as the types of mistakes that are made by the classifiers. This insight is beneficial in overcoming constraints of using accuracy metric alone. Figure 29 below provides a confusion matrix for each algorithm obtained on the validation data set.

Model			Actual	
			Malware	Benign
Logistic Regression	Predicted	Malware	TP = 4225 (74.70%)	FP = 309 (4.26%)
		Benign	FN = 1431 (25.30%)	TN = 6939 (95.74%)
Linear Discriminant Analysis	Predicted	Malware	TP = 4106 (72.60%)	FP = 397 (5.48%)
		Benign	FN = 1550 (27.40%)	TN = 6851 (94.52%)
K Neighbors Classifier	Predicted	Malware	TP = 4041 (71.45%)	FP = 222 (3.06%)
		Benign	FN = 1615 (28.55%)	TN = 7026 (96.94%)
Decision Tree Classifier	Predicted	Malware	TP = 4321 (76.40%)	FP = 246 (3.39%)
		Benign	FN = 1335 (23.60%)	TN = 7002 (96.61%)
Gaussian NB	Predicted	Malware	TP = 3713 (65.65%)	FP = 355 (4.90%)
		Benign	FN = 1943 (34.35%)	TN = 6893 (95.10%)
SVC	Predicted	Malware	TP = 4290 (75.85%)	FP = 379 (5.23%)
		Benign	FN = 1366 (24.15%)	TN = 6869 (94.77%)
Sequential (Deep Learning)	Predicted	Malware	TP = 4263 (75.37%)	FP = 194 (2.68%)
		Benign	FN = 1393 (24.63%)	TN = 7054 (97.32%)
Random Forest Classifier	Predicted	Malware	TP = 4336 (76.66%)	FP = 270 (3.73%)
		Benign	FN = 1320 (23.34%)	TN = 6978 (96.27%)
XGB Classifier	Predicted	Malware	TP = 4310 (76.20%)	FP = 319 (4.40%)
		Benign	FN = 1346 (23.80%)	TN = 6929 (95.60%)

Figure 29: Confusion Matrix for Validation Data Set - Experiment 1

Figure 29 above shows a slightly altered version of the confusion matrix, as we are showing additional information for each class. The additional information consists of True Positive Rate (TPR), False Positive Rate (FPR), False Negative Rate (FNR), and True Negative Rate (TNR). It is being shown in terms of percentage points in parenthesis for each class.

Our confusion matrix illustrates that the top-performing algorithm, Sequential (Deep Learning), performs comparatively better at each of the four classes True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) in comparison to the other eight machine learning algorithms. However, the performance is still below par, when compared with the results of experiment 2 in the next section.

7.8 Experiment 2 - Performance Comparison

In the second experiment, Droid Fence was amended to use the features specified in Table 6, Table 7, and Figure 24. These features are the ones that we identified as producing the

best result. The purpose of this experiment was to demonstrate that the usage of our features performs better than the ones utilised in Experiment 1 - Performance Comparison.

The performance of each algorithm is compared by observing accuracy, F1 score, precision, and recall metrics. Moreover, a confusion matrix for each algorithm is utilised for evaluation purposes. Figure 30 below shows an accuracy metric for training and validation data sets. The result shows that all algorithms have achieved a high accuracy score; eight out of nine algorithms have scored 0.91 or above. The Decision Tree Classifier has performed very well against the training data set measuring accuracy score of 0.991; however, the Sequential (Deep Learning) algorithm is the best performing algorithm on the validation data set, achieving an accuracy score of 0.968. The Random Forest Classifier is not far behind achieving an accuracy score of 0.962.

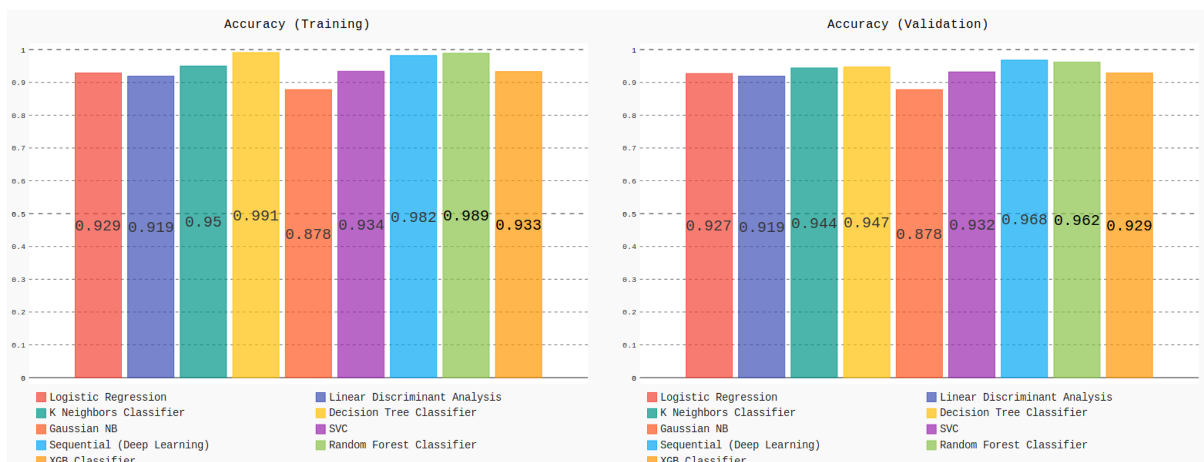

Figure 30: Accuracy Metric for Nine Algorithms for Training and Validation Data Set

The Decision Tree Classifier is marginally the third-best performing algorithm on the validation set, just ahead of the K-Neighbours Classifier. Gaussian NB has acquired a comparatively lowest score of 0.878. Logistic Regression, Linear Discriminant Analysis, SVC, XGB Classifier, and Gaussian NB have performed almost in the same fashion on both training and validation data sets.

Figure 31 below reflects each algorithm’s performance when measured against the F1 Score metric. All algorithms have performed very well, achieving over 0.90 scores for all but one

algorithm. The Decision Tree Classifier is again the best performing classifier on the training data set, producing an almost perfect score; however, the Sequential (Deep Learning) algorithm is the clear winner on the validation data set again, as it obtained F1 Score of 0.964. The Random Forest is again the second-best performing algorithm on the validation dataset acquiring an F1 Score of 0.956. Gaussian NB appears last when measured against the F1 Score metric. Logistic Regression, Linear Discriminant Analysis, SVC, XGB Classifier, and Gaussian NB have performed almost in the same fashion on both training and validation data sets.

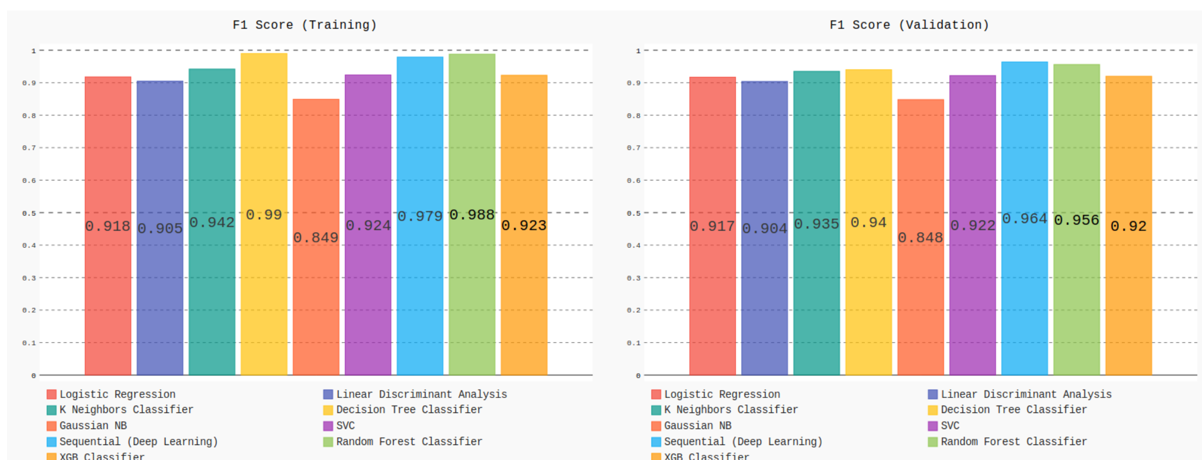

Figure 31: F1 Score Metric for Nine Algorithms for Training and Validation Data Set

Figure 32 below illustrates the Precision metric results for all nine algorithms. All algorithms have acquired over 0.91 scores against the Precision metric. The Random Forest Classifier, Decision Tree Classifier, and Sequential (Deep Learning) algorithms have attained almost perfect scores on the training data set. The Sequential (Deep Learning) algorithm has attained the top position on the validation data set. The Random Forest Classifier obtained the second position on the validation data set. The K Neighbours Classifier is the third-best performing algorithm on the validation data set, surpassing the Decision Tree Classifier by almost 0.2. The XGB Classifier is the worst performing algorithm in comparison to the remaining classifiers, even though it achieved an impressive score of 0.919.

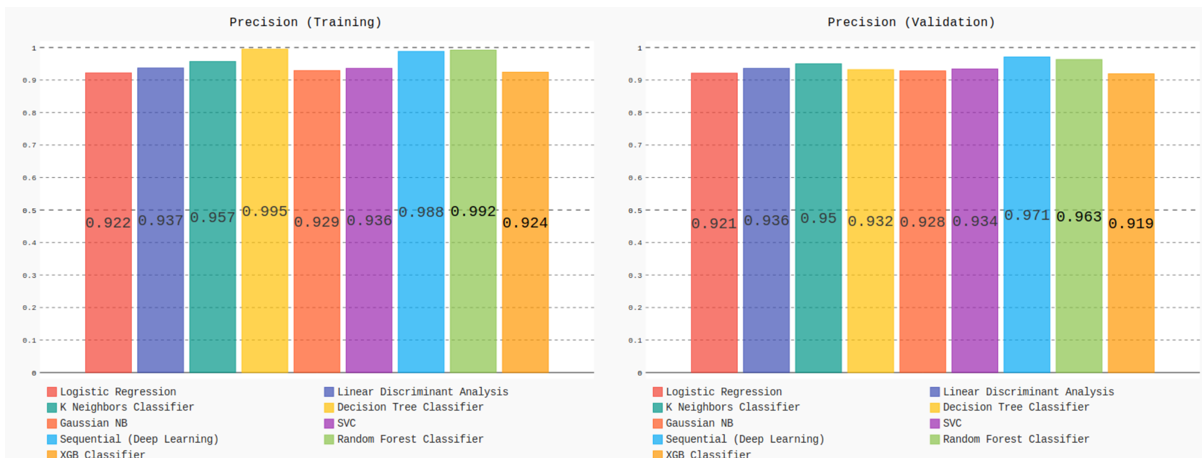

Figure 32: Precision Metric for Nine Algorithms for Training and Validation Data Set

Figure 33 below reveals outcomes for all algorithms measured against the Recall metric. Seven out of nine algorithms have gained a score of 0.91 or higher. The Decision Tree Classifier has obtained the highest recall score on the training data set, followed closely by the Random Forest Classifier and the Sequential (Deep Learning) algorithm, thus keeping the result consistent with the other three metrics. Linear Discriminant Analysis and Gaussian NB are the lowest-performing algorithms on both training – acquiring recall scores of 0.876 and 0.782, respectively – and validation data set – achieving recall scores of 0.875 and 0.782, respectively. Again, the best performing algorithm on the validation data set is Sequential (Deep learning), obtaining a recall score of 0.956, followed closely by Random Forest and Decision Tree Classifiers.

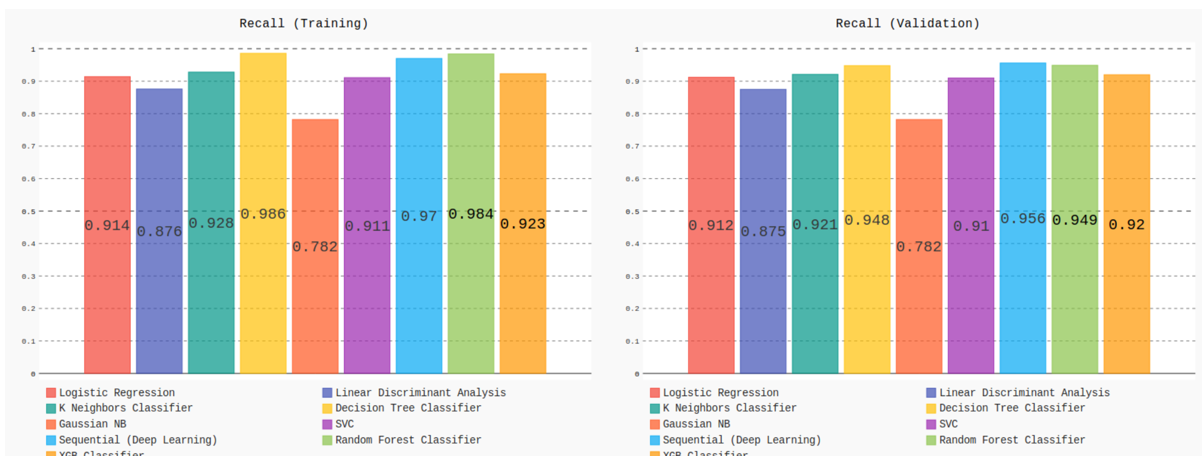

Figure 33: Recall Metric for Nine Algorithms for Training and Validation Data Set

Table 10 below displays algorithms in the order of best performing to the worst. It summarises the result measured against accuracy, F1 score, precision, and recall metrics for each algorithm applied to the validation data set. All algorithms have performed exceptionally well, achieving a score of 0.90 or more on thirty-two out of thirty-six possible outcomes.

Table 10: Detection Result Comparison of Nine Algorithms on Validation Data Set

Algorithm	**Accuracy**	**F1 Score**	**Precision**	**Recall**
Sequential (Deep Learning)	0.969	0.964	0.973	0.956
Random Forest Classifier	0.962	0.956	0.963	0.949
Decision Tree Classifier	0.947	0.94	0.932	0.948
K Neighbours Classifier	0.944	0.935	0.95	0.921
SVC	0.932	0.922	0.934	0.909
XGB Classifier	0.929	0.92	0.919	0.92
Logistic Regression	0.927	0.917	0.921	0.912
Linear Discriminant Analysis	0.919	0.904	0.936	0.875
Gaussian NB	0.878	0.848	0.928	0.782

Sequential (Deep Learning) is the best performing algorithm as it obtained 0.956 or more against all four metrics. The top machine learning algorithm is the ensemble method Random Forest Classifier acquiring 0.949 or more score on all four metrics. The second-best machine learning algorithm is the Decision Tree Classifier attaining 0.93 or more score on all four metrics when applied on the validation data set. The close third algorithm on machine learning family is the K Nearest Neighbours Classifier achieving 0.92 or more score on all four metrics. Gaussian NB remains in the last position; however, it is worth mentioning that using our approach, even the last positioned algorithm has achieved a score of 0.928 and mid to late eighties in three of the four metrics.

We are using a confusion matrix for summarising the performance of all classification algorithms that we used in our experiment. The confusion matrix provides helpful insight into inaccuracies made by our algorithms as well as the types of mistakes that are made by the classifiers. This insight is beneficial in overcoming constraints of using accuracy metric alone.

Figure 34 below provides a confusion matrix for each algorithm obtained on the validation data set.

Model			Actual	
			Malware	Benign
Logistic Regression	Predicted	Malware	TP = 5158 (91.20%)	FP = 441 (6.08%)
		Benign	FN = 498 (8.80%)	TN = 6807 (93.92%)
Linear Discriminant Analysis	Predicted	Malware	TP = 4950 (87.52%)	FP = 340 (4.69%)
		Benign	FN = 706 (12.48%)	TN = 6908 (95.31%)
K Neighbors Classifier	Predicted	Malware	TP = 5209 (92.10%)	FP = 272 (3.75%)
		Benign	FN = 447 (7.90%)	TN = 6976 (96.25%)
Decision Tree Classifier	Predicted	Malware	TP = 5364 (94.84%)	FP = 389 (5.37%)
		Benign	FN = 292 (5.16%)	TN = 6859 (94.63%)
Gaussian NB	Predicted	Malware	TP = 4421 (78.16%)	FP = 344 (4.75%)
		Benign	FN = 1235 (21.84%)	TN = 6904 (95.25%)
SVC	Predicted	Malware	TP = 5146 (90.98%)	FP = 364 (5.02%)
		Benign	FN = 510 (9.02%)	TN = 6884 (94.98%)
Sequential (Deep Learning)	Predicted	Malware	TP = 5408 (95.62%)	FP = 160 (2.21%)
		Benign	FN = 248 (4.38%)	TN = 7088 (97.79%)
Random Forest Classifier	Predicted	Malware	TP = 5365 (94.86%)	FP = 204 (2.81%)
		Benign	FN = 291 (5.14%)	TN = 7044 (97.19%)
XGB Classifier	Predicted	Malware	TP = 5202 (91.97%)	FP = 457 (6.31%)
		Benign	FN = 454 (8.03%)	TN = 6791 (93.69%)

Figure 34: Confusion Matrix for Validation Data Set

Figure 34 shows a slightly altered version of the confusion matrix, as we are showing additional information for each class. The additional information consists of True Positive Rate (TPR), False Positive Rate (FPR), False Negative Rate (FNR), and True Negative Rate (TNR). It is being shown in terms of percentage points in parenthesis for each class.

Our confusion matrix illustrates that the top-performing algorithm, Sequential (Deep Learning), excels at each of the four classes True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) in comparison to the other eight machine learning algorithms.

All but one algorithm (Gaussian NB) have achieved over 90% accuracy, as shown in Table 10 above, and none of the algorithms has FPR exceeding 3%, as shown in Figure 34 above. The FN of 248 for the Sequential algorithm means that it has wrongly identified 248 malware apps as benign (4.38%), similarly FP of 160 means that the algorithm has wrongly classified 160 benign apps as malware (2.21%). The FPR is not exceeding 3% and therefore these results

conform to the machine learning methodology proposed by Soviani, Scheianu and Suciu [168] to aid in the detection and recognition of malware. The primary hypothesis of the study was that the technique used should be able to detect malicious applications before they cause damage. As such, the accuracy target for malware detection should be at least 90% with false alarm (FPR) not exceeding 3% for a classifier [168].

The experiment results show that deep learning algorithm performed slightly better than the machine learning algorithm. Sequential (Deep Learning) is the best performing algorithm in our experiments as it obtained 0.956 or more against all four metrics. The top machine learning algorithm is the ensemble method Random Forest Classifier, achieving a score of 0.949 or more on all four metrics.

7.9 Result Comparison between Experiment 1 and 2

The performance of Experiment 1 and Experiment 2 has demonstrated that the usage of the neglected features identified by us has performed better. Table 11 below lists the best performing algorithms in both experiments. Experiment 2 has performed better than Experiment 1 against all four matrices. Experiment 1 is slightly behind when we compare Precision matric of both experiments, however there is a big difference in performance for the remaining three metrics.

Table 11: Best Algorithms in Experiment 1 and 2.

Best Algorithm / Experiment No	**Accuracy**	**F1 Score**	**Precision**	**Recall**
Experiment 1 - Sequential (Deep Learning)	0.877	0.843	0.957	0.754
Experiment 2 - Sequential (Deep Learning)	0.969	0.964	0.973	0.956

Table 12 below summarises the confusion matrix for the best algorithm of both experiments. The confusion matrix illustrates that the Experiment 2 has exceled at each of the four classes True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) in comparison to the Experiment 1. The FN of 248 for the Experiment 2 means that it has wrongly identified 248 malware apps as benign (4.38%), whereas Experiment 1 has wrongly identified

1393 malware apps as benign (24.63%). Similarly, FP of 160 means that the Experiment 2 has wrongly classified 160 benign apps as malware (2.21%), whereas Experiment 1 has wrongly classified 194 benign apps as malware (2.68%).

Table 12: Confusion Matrix for Experiment 1 and 2.

Best Algorithm / Experiment No	**TP**	**FP**	**FN**	**TN**
Experiment 1 - Sequential (Deep Learning)	4263 (75.37%	194 (2.68%)	1393 (24.63%)	7054 (97.32%)
Experiment 2 - Sequential (Deep Learning)	5408 (95.62%)	160 (2.21%)	248 (4.38%)	7088 (97.79%)

The result of the experiment 2 confirmed that it performed better than the Experiment 1 - Performance Comparison. Although both experiments have achieved our target FPR of less than 3% but experiment 1 has missed target of over 90% accuracy. Similarly, experiment 1 has performed poorly against F1 Score and Recall metrics. The result demonstrates that Droid Fence performed better when utilising the features combination proposed in Table 6, Table 7, and Figure 24.

7.10 Experiment 3 – Performance Comparison

The purpose of this experiment is to ascertain the performance of Droid Fence when it is trained and tested on malware and benign datasets being drawn from different time periods. For the purpose of this experiment, we trained Droid Fence on Malgenome and Drebin datasets (2010 – 2012) and tested it on the GitHub samples which contained samples from different time period (2015 – 2019).

The performance of each algorithm is compared by observing accuracy, F1 score, precision, and recall metrics. Moreover, a confusion matrix for each algorithm is utilised for evaluation purposes. Figure 35 below shows an accuracy metric for training and validation data sets. The result shows that all algorithms have achieved a high accuracy score; seven out of nine algorithms have scored 0.91 or above. The Decision Tree Classifier has performed very well

against the training data set measuring accuracy score of 0.991; however, it performed poorly against the validation data set. The Sequential (Deep Learning) algorithm is the best performing algorithm on the validation data set, achieving an accuracy score of 0.969. The Logistic Regression is not far behind achieving an accuracy score of 0.958.

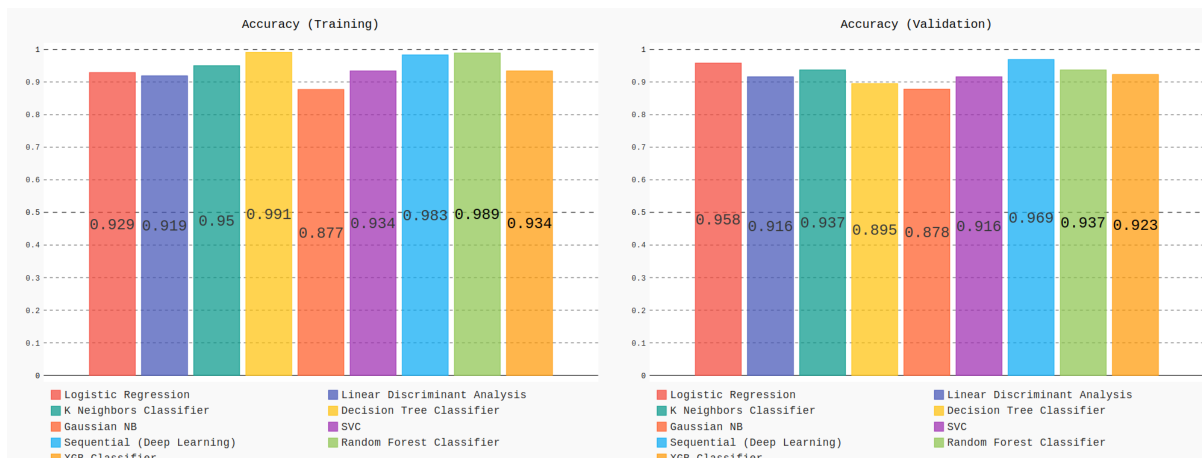

Figure 35: Accuracy Metric for Experiment 3 for Training and Validation Data Set

Both K-Neighbors and Random Forest Classifiers are the joint third-best performing algorithm on the validation set, just ahead of the XGB Classifier. Gaussian NB has acquired a comparatively lowest score of 0.878. Linear Discriminant Analysis, SVC, and Gaussian NB have performed almost in the same fashion on both training and validation data sets.

Figure 36 below reflects each algorithm's performance when measured against the F1 Score metric. Most algorithms have performed very well, achieving over 0.90 scores for six out of nine algorithms. The Decision Tree Classifier is again the best performing classifier on the training data set, producing an almost perfect score; however, the Sequential (Deep Learning) algorithm is the clear winner on the validation data set again, as it obtained F1 Score of 0.965. The Logistic Regression is again the second-best performing algorithm on the validation dataset acquiring an F1 Score of 0.952. Gaussian NB appears last when measured against the F1 Score metric. Linear Discriminant Analysis, SVC, XGB Classifier, and Gaussian NB have performed almost in the same fashion on both training and validation data sets.

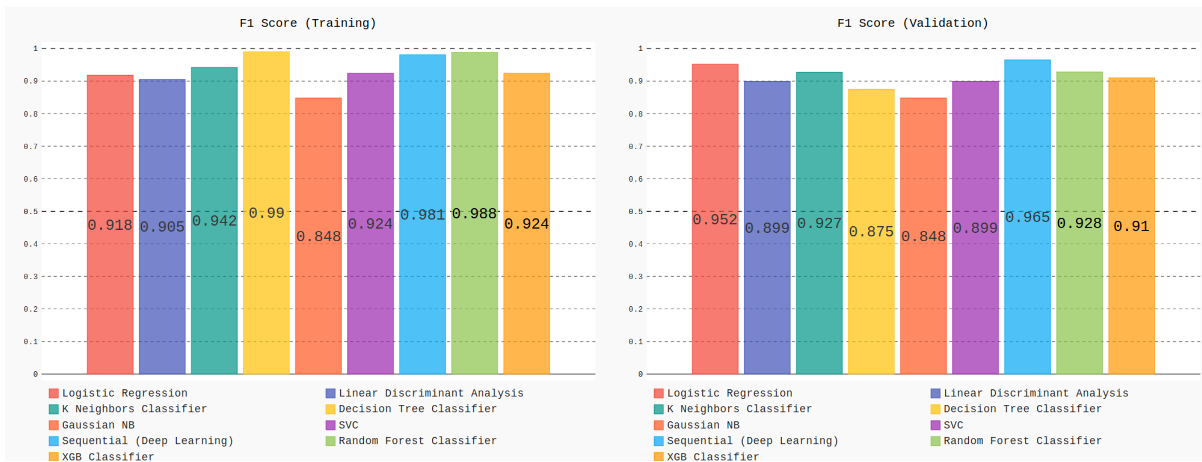

Figure 36: F1 Score Metric for Experiment 3 for Training and Validation Data Set

Figure 37 below illustrates the Precision metric results for all nine algorithms. All algorithms have acquired over 0.94 scores against the Precision metric. The Random Forest Classifier, Decision Tree Classifier, and Sequential (Deep Learning) algorithms have attained almost perfect scores on the training data set. However, Logistic Regression algorithm has attained the top position on the validation data set, followed closely by Linear Discriminant Analysis and SVC at joint-second position. The Sequential (Deep Learning) obtained the third position on the validation data set. The Decision Tree Classifier is the worst performing algorithm in comparison to the remaining classifiers, even though it achieved an impressive score of 0.955.

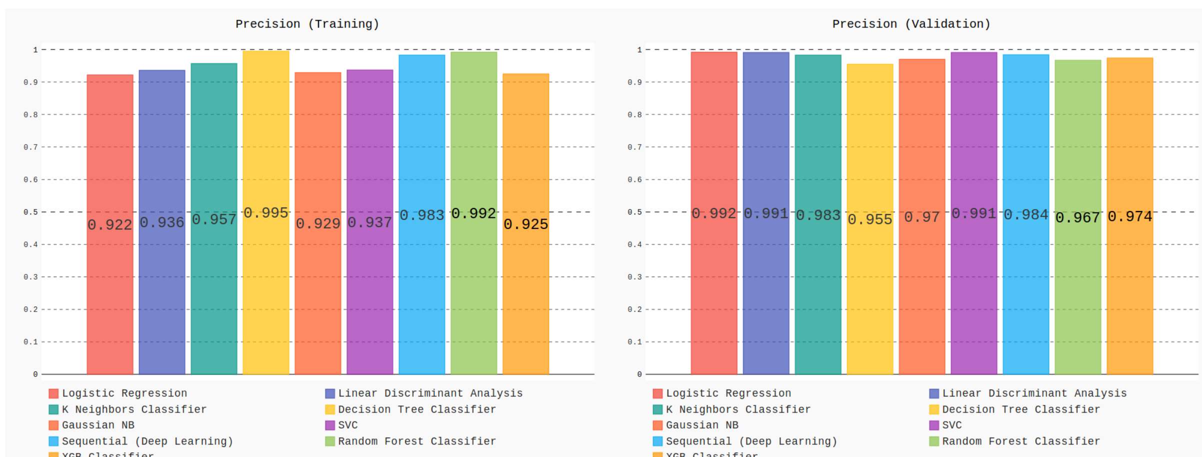

Figure 37: Precision Metric for Experiment 3 for Training and Validation Data Set

Figure 38 below reveals outcomes for all algorithms measured against the Recall metric. Two out of nine algorithms have gained a score of 0.91 or higher. The Decision Tree Classifier has obtained the highest recall score on the training data set, followed closely by the Random Forest Classifier and the Sequential (Deep Learning) algorithm. Linear Discriminant Analysis and Gaussian NB are the lowest-performing algorithms on both training – acquiring recall scores of 0.875 and 0.782, respectively – and validation data set – achieving recall scores of 0.823 and 0.754, respectively. The best performing algorithm on the validation data set is Sequential (Deep learning), obtaining a recall score of 0.946, followed closely by Random Forest and Decision Tree Classifiers.

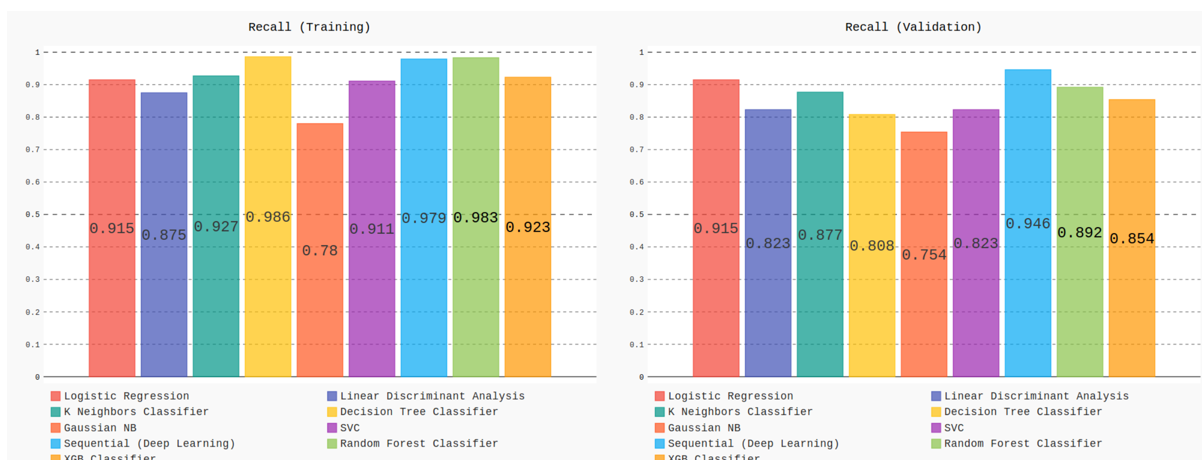

Figure 38: Recall Metric for Experiment 3 for Training and Validation Data Set

Table 13 below displays algorithms in the order of best performing to the worst. It summarises the result measured against accuracy, F1 score, precision, and recall metrics for each algorithm applied to the validation data set. All algorithms have performed well, achieving a score of 0.90 or more on twenty three out of thirty-six possible outcomes.

Table 13: Detection Result Comparison of Experiment 3 on Validation Data Set

Algorithm	**Accuracy**	**F1 Score**	**Precision**	**Recall**
Sequential (Deep Learning)	0.969	0.965	0.984	0.946
Logistic Regression	0.958	0.952	0.992	0.915
Random Forest Classifier	0.937	0.928	0.967	0.892
K Neighbours Classifier	0.937	0.927	0.983	0.877
XGB Classifier	0.923	0.91	0.974	0.854

SVC	0.916	0.899	0.991	0.823
Linear Discriminant Analysis	0.916	0.899	0.991	0.823
Decision Tree Classifier	0.895	0.875	0.955	0.808
Gaussian NB	0.878	0.848	0.955	0.754

Sequential (Deep Learning) is the best performing algorithm as it obtained 0.946 or more against all four metrics. The top machine learning algorithm is the Logistic Regression acquiring 0.915 or more score on all four metrics. The second-best machine learning algorithm is the Random Forest Classifier attaining 0.892 or more score on all four metrics when applied on the validation data set. The close third algorithm on machine learning family is the K Nearest Neighbours Classifier achieving 0.877 or more score on all four metrics. Gaussian NB remains in the last position; however, it is worth mentioning that using our approach, even the last positioned algorithm has achieved a score of 0.848 to 0.955 in three out of four matrices.

We are using a confusion matrix for summarising the performance of all classification algorithms that we used in our experiment. The confusion matrix provides helpful insight into inaccuracies made by our algorithms as well as the types of mistakes that are made by the classifiers. This insight is beneficial in overcoming constraints of using accuracy metric alone. Figure 39 below provides a confusion matrix for each algorithm obtained on the validation data set.

Model			Actual	
			Malware	Benign
Logistic Regression	Predicted	Malware	TP = 119 (91.54%)	FP = 1 (0.64%)
		Benign	FN = 11 (8.46%)	TN = 156 (99.36%)
Linear Discriminant Analysis	Predicted	Malware	TP = 107 (82.31%)	FP = 1 (0.64%)
		Benign	FN = 23 (17.69%)	TN = 156 (99.36%)
K Neighbors Classifier	Predicted	Malware	TP = 114 (87.69%)	FP = 2 (1.27%)
		Benign	FN = 16 (12.31%)	TN = 155 (98.73%)
Decision Tree Classifier	Predicted	Malware	TP = 105 (80.77%)	FP = 5 (3.18%)
		Benign	FN = 25 (19.23%)	TN = 152 (96.82%)
Gaussian NB	Predicted	Malware	TP = 98 (75.38%)	FP = 3 (1.91%)
		Benign	FN = 32 (24.62%)	TN = 154 (98.09%)
SVC	Predicted	Malware	TP = 107 (82.31%)	FP = 1 (0.64%)
		Benign	FN = 23 (17.69%)	TN = 156 (99.36%)
Sequential (Deep Learning)	Predicted	Malware	TP = 123 (94.62%)	FP = 2 (1.27%)
		Benign	FN = 7 (5.38%)	TN = 155 (98.73%)
Random Forest Classifier	Predicted	Malware	TP = 116 (89.23%)	FP = 4 (2.55%)
		Benign	FN = 14 (10.77%)	TN = 153 (97.45%)
XGB Classifier	Predicted	Malware	TP = 111 (85.38%)	FP = 3 (1.91%)
		Benign	FN = 19 (14.62%)	TN = 154 (98.09%)

Figure 39: Confusion Matrix for Validation Data Set - Experiment 3

Our confusion matrix illustrates that the top-performing algorithm, Sequential (Deep Learning), excels at each of the four classes True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) in comparison to the other eight machine learning algorithms. The FPR of 1.27%, thus achieving our target of FPR of less than 3% whilst achieving over 90% accuracy. The experiment results suggests that Droid Fence performs well when trained and tested on data samples being drawn from different time period.

7.11 Experiment 4 – Performance Comparison

The final experiment is conducted on all three datasets Malgenome, Drebin, GitHub. The performance of each algorithm is compared by observing accuracy, F1 score, precision, and recall metrics. Moreover, a confusion matrix for each algorithm is utilised for evaluation purposes. Figure 40 below shows an accuracy metric for training and validation data sets. The result shows that all algorithms have achieved a high accuracy score; eight out of nine algorithms have scored 0.92 or above. The Decision Tree Classifier has performed very well against the training data set measuring accuracy score of 0.991; however, the Sequential

(Deep Learning) algorithm is the best performing algorithm on the validation data set, achieving an accuracy score of 0.971. The Random Forest Classifier is not far behind achieving an accuracy score of 0.958.

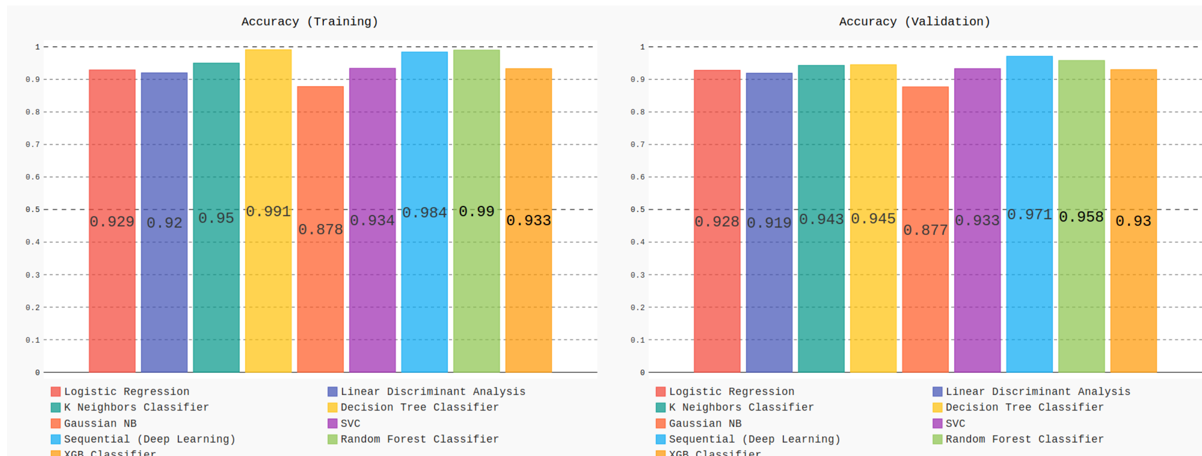

Figure 40: Accuracy Metric for Experiment 4 for Training and Validation Data Set

The Decision Tree Classifier is the third-best performing algorithm on the validation set, just marginally ahead of the K-Neighbours Classifier. Gaussian NB has acquired a comparatively lowest score of 0.877. Logistic Regression, Linear Discriminant Analysis, SVC, XGB Classifier, and Gaussian NB have performed almost in the same fashion on both training and validation data sets.

Figure 41 below reflects each algorithm's performance when measured against the F1 Score metric. All algorithms have performed very well, achieving over 0.90 scores for all but one algorithm. The Decision Tree Classifier is again the best performing classifier on the training data set, producing an almost perfect score; however, the Sequential (Deep Learning) algorithm is the clear winner on the validation data set again, as it obtained F1 Score of 0.967. The Random Forest is again the second-best performing algorithm on the validation dataset acquiring an F1 Score of 0.952. Gaussian NB appears last when measured against the F1 Score metric. Logistic Regression, Linear Discriminant Analysis, SVC, XGB Classifier, and Gaussian NB have performed almost in the same fashion on both training and validation data sets.

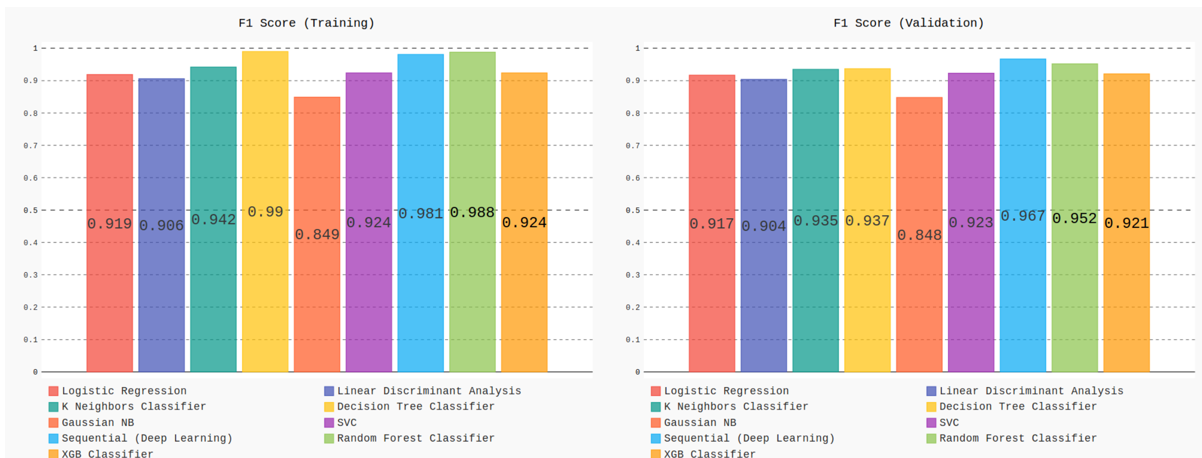

Figure 41: F1 Score Metric for Experiment 4 for Training and Validation Data Set

Figure 42 below illustrates the Precision metric results for all nine algorithms. All algorithms have acquired over 0.92 scores against the Precision metric. The Random Forest Classifier, Decision Tree Classifier, and Sequential (Deep Learning) algorithms have attained almost perfect scores on the training data set. The Sequential (Deep Learning) algorithm has attained the top position on the validation data set. The Random Forest Classifier obtained the second position on the validation data set. The K Neighbours Classifier is the third-best performing algorithm on the validation data set, marginally surpassing the Decision Tree Classifier. The XGB Classifier and Logistic Regression are the joint-worst performing algorithm in comparison to the remaining classifiers, even though both achieved an impressive score of 0.92.

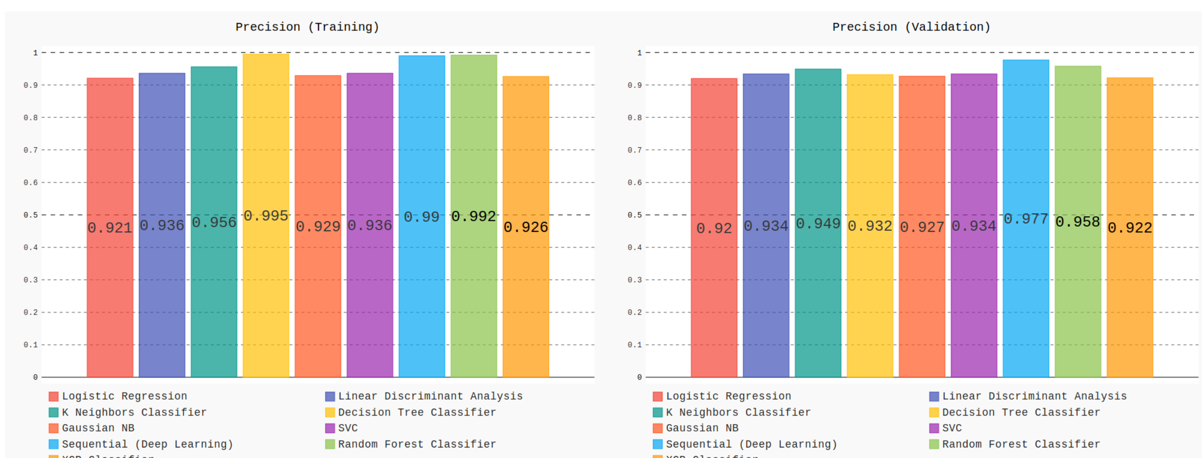

Figure 42: Precision Metric for Experiment 4 for Training and Validation Data Set

Figure 43 below reveals outcomes for all algorithms measured against the Recall metric. Seven out of nine algorithms have gained a score of 0.91 or higher. The Decision Tree Classifier has obtained the highest recall score on the training data set, followed closely by the Random Forest Classifier and the Sequential (Deep Learning) algorithm, thus keeping the result consistent with the other three metrics. Linear Discriminant Analysis and Gaussian NB are the lowest-performing algorithms on both training – acquiring recall scores of 0.878 and 0.783, respectively – and validation data set – achieving recall scores of 0.876 and 0.782, respectively. Again, the best performing algorithm on the validation data set is Sequential (Deep learning), obtaining a recall score of 0.956, followed closely by Random Forest and Decision Tree Classifiers.

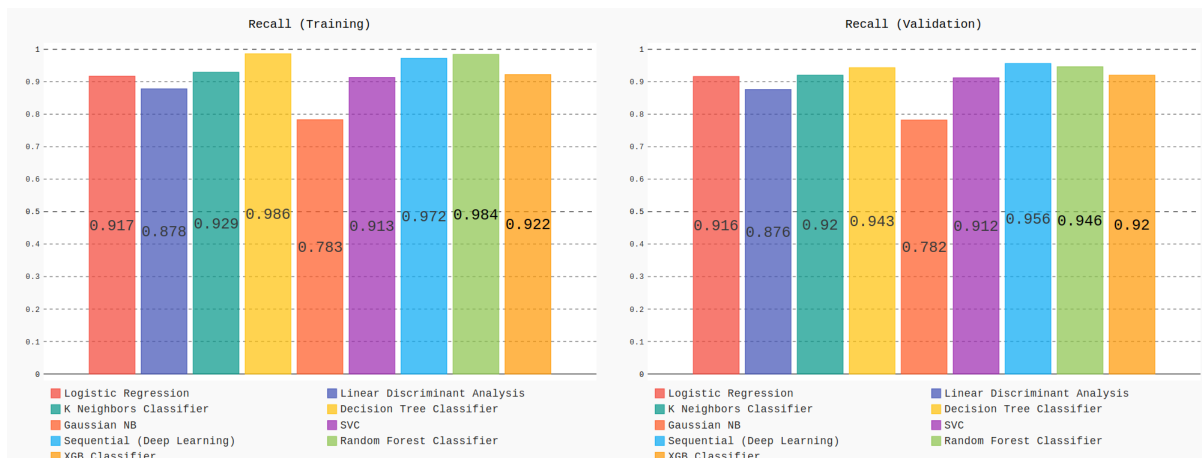

Figure 43: Recall Metric for Experiment 4 for Training and Validation Data Set

Table 14 below displays algorithms in the order of best performing to the worst. It summarises the result measured against accuracy, F1 score, precision, and recall metrics for each algorithm applied to the validation data set. All algorithms have performed exceptionally well, achieving a score of 0.90 or more on thirty-two out of thirty-six possible outcomes.

Table 14:Detection Result Comparison of Experiment 4 on Validation Data Set

Algorithm	Accuracy	F1 Score	Precision	Recall
Sequential (Deep Learning)	0.971	0.967	0.977	0.956
Random Forest Classifier	0.958	0.952	0.958	0.946
Decision Tree Classifier	0.945	0.937	0.932	0.943

K Neighbours Classifier	0.943	0.935	0.949	0.92
SVC	0.933	0.923	0.934	0.912
XGB Classifier	0.93	0.921	0.922	0.92
Logistic Regression	0.928	0.917	0.92	0.916
Linear Discriminant Analysis	0.919	0.904	0.934	0.876
Gaussian NB	0.877	0.848	0.927	0.782

Sequential (Deep Learning) is the best performing algorithm as it obtained 0.956 or more against all four metrics. The top machine learning algorithm is the ensemble method Random Forest Classifier acquiring 0.946 or more score on all four metrics. The second-best machine learning algorithm is the Decision Tree Classifier attaining 0.932 or more score on all four metrics when applied on the validation data set. The close third algorithm on machine learning family is the K Nearest Neighbours Classifier achieving 0.92 or more score on all four metrics. Gaussian NB remains in the last position; however, it is worth mentioning that using our approach, even the last positioned algorithm has achieved a score of 0.928 and mid to late eighties in three of the four metrics.

We are using a confusion matrix for summarising the performance of all classification algorithms that we used in our experiment. The confusion matrix provides helpful insight into inaccuracies made by our algorithms as well as the types of mistakes that are made by the classifiers. This insight is beneficial in overcoming constraints of using accuracy metric alone. Figure 44 below provides a confusion matrix for each algorithm obtained on the validation data set.

Model			Actual	
			Malware	Benign
Logistic Regression	Predicted	Malware	TP = 5298 (91.55%)	FP = 464 (6.27%)
		Benign	FN = 489 (8.45%)	TN = 6940 (93.73%)
Linear Discriminant Analysis	Predicted	Malware	TP = 5069 (87.59%)	FP = 356 (4.81%)
		Benign	FN = 718 (12.41%)	TN = 7048 (95.19%)
K Neighbors Classifier	Predicted	Malware	TP = 5325 (92.02%)	FP = 284 (3.84%)
		Benign	FN = 462 (7.98%)	TN = 7120 (96.16%)
Decision Tree Classifier	Predicted	Malware	TP = 5456 (94.28%)	FP = 401 (5.42%)
		Benign	FN = 331 (5.72%)	TN = 7003 (94.58%)
Gaussian NB	Predicted	Malware	TP = 4524 (78.18%)	FP = 354 (4.78%)
		Benign	FN = 1263 (21.82%)	TN = 7050 (95.22%)
SVC	Predicted	Malware	TP = 5276 (91.17%)	FP = 371 (5.01%)
		Benign	FN = 511 (8.83%)	TN = 7033 (94.99%)
Sequential (Deep Learning)	Predicted	Malware	TP = 5534 (95.63%)	FP = 129 (1.74%)
		Benign	FN = 253 (4.37%)	TN = 7275 (98.26%)
Random Forest Classifier	Predicted	Malware	TP = 5475 (94.61%)	FP = 239 (3.23%)
		Benign	FN = 312 (5.39%)	TN = 7165 (96.77%)
XGB Classifier	Predicted	Malware	TP = 5322 (91.96%)	FP = 452 (6.10%)
		Benign	FN = 465 (8.04%)	TN = 6952 (93.90%)

Figure 44: Confusion Matrix for Validation Data Set - Experiment 4

Our confusion matrix illustrates that the top-performing algorithm, Sequential (Deep Learning), excels at each of the four classes True Positive (TP), False Positive (FP), False Negative (FN), and True Negative (TN) in comparison to the other eight machine learning algorithms.

All but one algorithm (Gaussian NB) have achieved over 90% accuracy, as shown in Table 14 above, and none of the algorithms has FPR exceeding 3%, as shown in Figure 44 above. The FN of 253 for the Sequential algorithm means that it has wrongly identified 253 malware apps as benign (4.37%), similarly FP of 129 means that the algorithm has wrongly classified 129 benign apps as malware (1.74%). The FPR is not exceeding 3% and therefore these results conform to the machine learning methodology proposed by Soviani, Scheianu and Suciu [168] to aid in the detection and recognition of malware. The primary hypothesis of the study was that the technique used should be able to detect malicious applications before they cause damage. As such, the accuracy target for malware detection should be at least 90% with false alarm (FPR) not exceeding 3% for a classifier [168].

The above results answers both research questions: RQ5 and RP6.

RQ5: Will the usage of neglected features identified in literature review along with Android permissions and services allow the achievement of over 90% accuracy whilst keeping FPR less than 3% in detecting Android malware?

The Droid Fence has achieved an accuracy of 0.971 and a FPR of 1.74 (less than 3%) when utilised the above stated features.

RQ6: How does the performance of the proposed deep learning algorithm (in terms of accuracy, F1 score, precision, and recall) compared to that of existing machine learning algorithms?

Our experiment results show that deep learning algorithm performed slightly better than the machine learning algorithm. Sequential (Deep Learning) is the best performing algorithm in our experiments as it obtained 0.956 or more against all four metrics. The top machine learning algorithm is the ensemble method Random Forest Classifier, achieving a score of 0.946 or more on all four metrics.

7.12 Comparison with related methods

In this section, we attempt to answer our research question seven.

- **RQ7:** Does the approach developed as part of RQ5 performs better (in terms of Accuracy) than comparative methods?

We provide the performance of the Droid Fence against related methods which utilise static analysis and some of the similar features. We have used the results of experiment 4 for this purpose because this experiment contains the full dataset and our proposed features. There have been various Android malware detection techniques and tools presented over the years. An SVM-based approach has been used [111] that combines risky permissions and sensitive API calls and feeds them as features to the SVM algorithm. They have achieved 86% accuracy on a data set of 700 applications, consisting of equal distribution of malware and benign applications.

Sanz et al. [112] acquired an accuracy of 86.41% on a data set of 249 malware and 1811 benign applications. Their method extracts permissions from an Android manifest file and utilises various machine learning algorithms for detecting malware. Random Forest algorithm trained with fifty trees achieved the highest accuracy.

DroidDet [66] employs the Rotation Forest algorithm and trains it with features extracted by conducting a static analysis of 1065 malware and 1065 benign applications. Static analysis extracted permissions, monitoring system events, sensitive APIs, and permission rates as features for constructing the machine learning model. Their method achieves 88.26% accuracy, and they report a 3.33% improvement to the SVM algorithm.

APK Auditor [92] is a permission-based Android malware detection tool; it utilises static analysis to extract permissions information and store it along with the analysis result into a signature database. The tool also contains an Android application that is stored on end-users' mobiles to allow them to request analysis. The application communicates with a central server, which provides the result of the analysis. APK Auditor has been tested on 6909 malware and 1853 benign applications dataset with a reported accuracy of 88%.

Deepa et al. [113] applies feature extraction and dimensionality reduction methods - namely Information Gain, Correlation Feature Selection (CFS), and Kruskal methods - in their static analysis approach. Their technique utilises the extraction of three types of features: method names, strings, and opcodes for constructing machine learning models. Their dataset consisted of 612 malware and 758 benign applications. Their techniques achieved 88.75% of accuracy by using AdaBoost with J48 as the base classifier.

AppContext [114] extracts security-sensitive behaviours by conducting a static analysis. The security-sensitive behaviour is classified as API calls to certain methods that are permission-protected or are sink method, i.e., require access to functions that write data to a file. This data is used to train an SVM classifier. AppContext achieved 93.2% accuracy when applied on 202 malware and 633 benign applications dataset.

Table 15 summarises the performance of the Droid Fence against other related methods. Droid Fence surpasses other approaches with an accuracy of 97.1%. The other methods acquired an accuracy rate between 86% and 93.2%. Droid Fence achieves an improvement of 3.9% against AppContext, the best performing approach among the comparative methods. Moreover, Droid Fence utilised a more extensive data set (13191 applications) in comparison to all comparative approaches and, therefore, been tested against an enhanced amount of data.

Table 15: Comparison with related methods

Method/Publication	**Accuracy**	**Malware**	**Benign**
Detecting Malware for Android Platform: An SVM based Approach [111]	86	350	350
PUMA [112]	86.41	249	1811
DroidDet [66]	88.26	1065	1065
APK Auditor [92]	88.28	6909	1853
Investigation of Feature Selection Methods for Android Malware Analysis [113]	88.75	612	758
AppContext [114]	93.2	202	633
Droid Fence	**97.1**	**5787**	**7404**

8. Conclusion and Future Work

The popularisation and openness of Android devices have paved the way for cybercriminals to intrude individual privacy by introducing malware applications [2]. Malware is described as a set of instructions that can potentially harm a computer-related system within a network [4]. As the number of Android devices and applications increase, the number of malware applications also increases [2]. In efforts to address this problem, researchers have come together to develop malware detection techniques to identify and counter intrusion by malicious artefacts.

8.1 Research Questions

We had identified the following research questions.

- **RQ1:** How do we classify Android application security analysis provided in the literature?
- **RQ2:** What is the current state of analysing Android malware detection techniques and technologies?
- **RQ3:** What challenges, gaps, and patterns might be deduced from the existing research attempts, which will inform further research?

The third research question has helped identified further research questions to devise our research path.

- **RQ4:** Is it possible to devise an efficient process for running experiments, decompiling the APK, obtaining the required features, and viewing and comparing the results?
- **RQ5:** Will the usage of neglected features identified in literature review along with Android permissions and services allow the achievement of over 90% accuracy whilst keeping FPR less than 3% in detecting Android malware?

- **RQ6:** How does the performance of the proposed deep learning algorithm (in terms of accuracy, F1 score, precision, and recall) compare to that of existing machine learning algorithms?
- **RQ7:** Does the approach developed as part of RQ5 performs better (in terms of Accuracy) than comparative methods?

8.1.1 Research Question 1 (RQ1)

In attempting to answer our first research questions: How do we classify Android application security analysis provided in the literature, we have identified and reviewed two primary analysis techniques: Signature-based detection, and Behaviour-based detection, as well as complimentary technique Machine/Deep learning [39].

Signature-based detection relies on creating signatures of benign and malware applications and compares these signatures with existing signatures stored in a database [90]. Signature-based methods contain less overhead and provide a fast result as applications do not need to run in a sandbox environment to generate signatures of an application [109]. The process may use one of the combinations of the following items to create a unique signature: string variables, function names, permissions, intents, package names, and broadcast receivers etc. The technique is not useful in detecting zero-day malware because the signature of such malware may not exist in a signature database [47]. However, this limitation can be countered by using a supplementary technique such as machine learning or deep learning as these techniques do not rely on a signature database for classification purpose [103].

Another limitation of the signature-based method is the detection of obfuscated malware where malware developers may change names of string variables or function names in the code, which would change the signature of the malware application [58]. This limitation may be countered by using information (such as permissions) held in the manifest file, as the permission names cannot be obfuscated due to the Android development model [121].

The behaviour-based technique relies on running applications in a controlled environment and observing their behaviours [95]. The method creates a behaviour-based profile for applications and flags them for further analysis if suspicious behaviour is observed [96]. The example of questionable behaviour could be accessing sensitive APIs that may not suit the description of an application, e.g. a weather application trying to access contacts. The technique is useful in detecting obfuscated malware as it does not rely on code-based signatures. However, behaviour-based detection is complicated as it requires manual steps to set up a controlled environment and execute each application. Another limitation is that the behaviour-based method does not cover the full code path execution [97].

Machine learning and deep learning-based methods offer complementary solutions to both signature-based detection and behaviour-based detection [75]. The supplementary solutions work by utilising features extracted from signature or behaviour-based solutions and feeding them to various machine learning and deep learning algorithms for classification purposes. The machine and deep learning methods overcome some of the shortcomings of the existing techniques as these methods do not rely on a signature database or behaviour profiles [76] [103].

8.1.2 Research Question 2 (RQ2)

In attempting to answer the first part (techniques) of the second research question: what is the current state of analysing Android malware detection techniques, we have identified and reviewed three primary analysis techniques: Static Analysis, Dynamic Analysis, and Hybrid Analysis. The applicability of different techniques depends on their accuracy levels, features used and limitations.

Static Analysis technique applies a signature-based and permission-based method to extract features from a set of benign and malware samples without running the applications [120]. The extracted features are, but not limited to, permissions, intents, method names, string variables, package information, and text mining etc [25]. The extracted features are passed as input to machine learning or deep learning algorithms to classify applications as benign or

malware [66]. The static analysis technique consumes fewer resources and efficient in detecting malware. Our survey indicates that static analysis has achieved up to 96% accuracy. However, the static analysis uses signature-based method so are prone to the limitation of the signature-based method, e.g. difficulty in detecting obfuscated malware. This limitation may be overcome if the analysis uses non-code-based features such as permissions, intents, and package information which are resilient to obfuscation due to Android's development model. Another limitation is that the technique is unable to extract features from dynamic code [123].

Dynamic Analysis technique extracts features from an application by executing them in a controlled environment [62]. The method extracts features such as API calls, CPU usage, memory usage, battery power [121]. It may extract network traffic features such as IP address, length of packet lengths etc. The technique is also useful in detecting obfuscated malware as it does not depend on code analysis [138]. The feature may be fed to various machine learning and deep learning algorithms to obtain a classification score against different metrics. Our survey indicates that the dynamic analysis has achieved up to 96.24% accuracy, which is slightly better than the static analysis (96%) [135]. A limitation of the dynamic analysis technique is that it requires a lot of resources to employ as applications need to be run in a controlled environment [137]. Another limitation is that it fails to detect intelligent malware which detects a sandbox environment and does not execute malicious code. Android morphing techniques often compromise the effectiveness of dynamic analysis during execution [139].

Hybrid Analysis technique, as the name implies, consist of static and dynamic analysis. There are mainly three steps for hybrid analysis, employing static analysis, next executing applications in a controlled environment for conducting dynamic analysis, and finally feeding features extracted from both analysis to machine learning and deep learning algorithms [44]. Our survey indicates that the hybrid analysis has achieved up to 96.92% accuracy [148], which is marginally better than static analysis (96%) and dynamic analysis (96.24%). Theoretically, combining both techniques should increase detection accuracy significantly; however, our

survey shows that this is not the case. The main limitation of the hybrid technique is the complexity of combining both static and dynamic analysis, while marginally increasing overall accuracy [53]. This complexity could be the reason for the decline in the usage of the hybrid analysis from 2017 onwards.

Several custom-built technologies have been identified in our attempt to answer the second part (technologies) of the second research question: What is the current state of analysing Android malware detection technologies? Several technologies have been proposed to facilitate malware detection and classification. These include CrowDroid, DroidOlytic, MigDroid, MalDroide, Dendroid, Androguard, Andromaly, TaintDroid, EvoDroid and SmartDroid. The technologies employ different tools and techniques to provide a high-performance platform for malware analysis. The context of application differs in terms of how they operate, features used for analysis and success rates. All the technologies presented in the Technologies for Android Malware Detection section above illustrated a capacity to attain more than 90% accuracy rate with varied sets of data. For instance, the Androguard framework employs reverse engineering to train and analyse requested permissions with an accuracy rate of 96% [127]. Although the technology can extract features from even broken applications, it can only work with malware samples; this implies that benign samples that may contain traces of malware can bypass the framework which questions its effectiveness [127]. Other technologies such as Dendroid, MigDroid and CrowDroid are based on data mining and retrieval for extraction of codes and classification [155] [157]. A notable strength of such a model is that it is scalable, fast, and accurate in analysing malware. From the literature reviewed, CrowDroid is the only framework with a capacity to attain a 100% accuracy rate [6]. However, the experiment was done on only 60 applications, so it is not known how it would behave when a bigger sample size is used. Despite this success, the rising privacy issues have constrained its usability.

Frameworks that combined multiple analysis methods illustrate a higher success rate. For instance, EvoDroid combines Android specific analysis and evolutionary algorithms novel

techniques to achieve higher code coverage [158]. Similarly, SmartDroid integrates dynamic and static analysis features to extract and transverse features which make it more effective compared to TaintDroid [159]. However, combining multiple analysis methods deems these technologies complex making it challenging to obtain native codes for unpacked malware samples. Further, from the literature reviewed, it is notable the accuracy of the discussed technologies depends on the sample size used with larger sample sizes being preferable.

8.1.3 Research Question 3 (RQ3)

Several gaps and limitations have been identified in our attempt to answer the final research question: what challenges, gaps, and patterns might be deduced from the existing research attempts, which will inform further research? One of the research gaps is where malicious code is hidden in more than one application, which may enable both applications to bypass detection individually [21]. Once both seemingly benign applications are installed on the same device, they may communicate in the background via services for malicious purposes. Therefore, the usage of Android services in any detection technology's feature set should minimise this threat.

Another limitation is the capabilities of the malicious applications to load malicious code at run time dynamically. Although dynamic analysis may protect against this technique, the static analysis does not provide any detection capability [43]. Therefore, the usage of detecting if dynamic code is used during the static analysis may provide some resilience to the static analysis against this type of threat.

Limited research has been done on the financial cost of developing malware detection systems. A further comprehensive investigation into this area will help software development companies to build the most cost-effective and efficient malware detection technologies.

Another limitation is the abilities of malware authors to hide malicious code through encryption in packed samples. The encryption hinders static analysis to be done correctly. It also affects the dynamic analysis, when encryption is combined with intelligent malware which detects a

controlled environment and does not execute the malicious encrypted code [130]. Therefore, detection of encrypted code usage should be used as a feature set during static analysis to mitigate this threat.

Another gap identified is the development of advance network architectures that ceases the usage of legacy communication protocols which may be averse to malware. It further includes the principles of configuration, guidelines, and operation to be defined for malware detection, e.g. active monitoring of network assets [149].

Malware have been evolving over the years; therefore, a gap has been identified to use genetic algorithms for detecting malware. Understanding the evolution of malware to create signature features to be used as vaccines for future malware detection is an important area that requires further research [118].

Another weak link in the chain is the end-user, as malware authors are increasingly applying social engineering tactics into deceiving end-users to install the malicious application. Further research into this area for finding the best practices to educate end-users on the tactics employed by adversaries and to be vigilant of the dangers posed by unofficial Android markets is necessary [49].

Another gap identified is the manual steps required, from acquiring benign and malware samples to extracting features and storing results add to the cumbersomeness of designing and reviewing malware detections systems. Therefore, more research is needed in the creation of automated tools which could reduce or remove the complexity involving the detection process [16].

The literature review suggests that different studies have used features such as permissions, API calls, system events, strings, and method names etc. in their static analysis. However, features such as services and the presence of useful functionality such as detection of code for cryptography, dynamic code loading, native code, HTTPs, database, and reflection have not been utilised in detail, especially in the same experiments.

Some of the research gaps cited in the previous paragraphs can be fulfilled by creating an approach that amalgamates the features for detecting Android Malware. The literature review also suggests that the process for extracting features from multiple malware and benign applications is not straightforward as it requires various steps. These steps are decompiling APK, obtaining the needed features, storing those features into some storage medium and feeding them to machine learning algorithms. There is a requirement to automate this process using a visual interface.

Taking all this into consideration, we developed a technique that amalgamates these neglected features: a set of permissions, services, and six other features (usage of https, database, dynamic code, native code, reflection, and cryptography) to generate a matrix that is used for detecting malware effectively. To the best of our knowledge, this is a novel approach that combines these features to detect malware.

8.1.4 Research Question 4 (RQ4)

In order to answer RQ4 Is it possible to devise an efficient process for running experiments, decompiling the APK, obtaining the required features, and viewing and comparing the results?), we developed Droid Fence, a web-based framework, which allows users to run experiments to extract various static features, store them in a relational database, and apply different machine learning and deep learning algorithms to detect malware and commit the result into a database. Droid Fence provides a web front-end to view, evaluate, and compare the results of nine algorithms stored against each experiment.

8.1.5 Research Question 5 (RQ5)

We utilised the Droid Fence to use our proposed technique in experiments to answer RQ5: Will the usage of neglected features identified in literature review along with Android permissions and services allow the achievement of over 90% accuracy whilst keeping FPR less than 3% in detecting Android malware?

Droid Fence is evaluated on a dataset of 13191 applications consisting of 5787 malware and 7404 benign applications. Our results show that Droid Fence is particularly useful when it utilises a Sequential (Deep Learning) algorithm to detect malware, achieving accuracy, F1-measure, precision, and recall scores of 0.971, 0.967, 0.977, and 0.956, respectively. The FPR acquired by Sequential is 1.74, less than 3%.

8.1.6 Research Question 6 (RQ6)

Our experiment, demonstrates that deep learning Sequential algorithm scored consistently highly when compared against eight machine learning algorithms, whilst achieving a FPR of 1.74. However, the difference between the accuracy scores achieved by the Sequential (97.1%) and Random Forest Classifier (95.8%) is minimal in comparison with the remaining algorithms used in our experiments. We used a stratified k-fold cross-validation method, and the result was compared for four metrics: accuracy, F1 score, precision, and recall.

8.1.7 Research Question 7 (RQ7)

The answer to our seventh research question: 'Does the approach developed as part of RQ5 performs better (in terms of Accuracy) than comparative methods?' is Yes. Droid Fence surpasses other approaches with an accuracy of 97.1%. The other methods acquired an accuracy rate between 86% and 93.2%. Droid Fence achieves an improvement of 3.9% against AppContext, the best performing approach among the comparative methods.

8.1.8 Supplementary Questions

As part of finding answers to our research questions, we have identified a couple of supplementary questions to evaluate Droid Fence further. The first question was to demonstrate whether our proposed features are better or not by using the same algorithms on different features. To answer this question, we conducted two experiments detailed in sections Experiment 1 - Performance Comparison and Experiment 2 - Performance Comparison. The experiment 2 utilised our proposed features whereas the experiment 1 utilised different features. The results of both experiments demonstrated that using our

proposed features has performed much better. Although both experiments have achieved our target FPR of less than 3% but experiment 1 has missed target of over 90% accuracy. Similarly, experiment 1 has performed poorly against F1 Score and Recall metrics. The result demonstrates that Droid Fence performed better when utilising out proposed features combination detailed in Table 6, Table 7, and Figure 24.

The second supplementary question was to ascertain the performance of Droid Fence when it is trained and tested on malware and benign datasets being drawn from different time periods. To answer this question, we conducted an experiment detailed in section Experiment 3 – Performance Comparison, whereby we trained Droid Fence on Malgenome and Drebin datasets (2010 – 2012) and tested it on the GitHub samples which contained samples from different time period (2015 – 2019). The result confirms that the Droid Fence performs well on dataset from different time period. The best performing algorithm was again Sequential (Deep Learning) as it obtained 0.946 or more against all four metrics. The accuracy was 96.9% marginally behind the accuracy 97.1% of experiment 4.

8.2 Future Work

The future work can be proposed into two domains: Droid Fence and Android Malware Detection area. In this section, we briefly proposed the future direction for both domains.

Droid Fence can be enhanced by implementing the following improvements:

1) Droid Fence supports permissions, services and six other features (usage of https, database, dynamic code, native code, reflection, and cryptography) for analysis. New features such as package information, function names etc. can be added for further analysis.
2) Droid Fence supports static analysis, and it can be extended to include dynamic and hybrid analysis for further research.
3) Droid Fence supports Nine machine learning and deep learning algorithms. It can be extended to include support for more algorithms.

4) Droid Fence can be hosted in a cloud, and an external-facing API interface can be developed for Droid Fence, which can allow other researchers to access features and malware dataset which already exist in the database.

A future direction for Android malware detection can be enhanced by researching the communication between seemingly different applications installed on the same device. The multiple application or point of entry attack is orchestrated from different applications to complete a malicious process. Applications which look benign and pass the detection stage are designed to function with other similar applications, where they have severe impacts on affected devices. Once both seemingly benign applications are installed on the same device, they may communicate in the background for malicious purposes. Hence, there is a need to dynamically detect the communication between different applications to detect whether there is a malicious behaviour among multiple applications.

Another future direction for Android malware detection area is to detect malware which installs malicious code dynamically. Seemingly, benign applications can download and install dynamic code once an application has passed the detection phase. There is a need to scrutinize the downloaded dynamic code closely before application has a chance to execute it.

9. References

[1] A. Apps, “Google Play Store: number of apps 2019 | Statistic,” 2019. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Accessed 15 Mar 2020].

[2] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto and F. Roli, “Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection,” *IEEE Transactions on Dependable and Secure Computing,* pp. 1-1, 2017.

[3] IDC Corporate, “IDC - Smart Phone Market Share - OS,” 2020. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os. [Accessed 15 Mar 2020].

[4] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” *2012 IEEE Symposium on Security and Privacy,* pp. 95-109, 2012.

[5] “Mobile Malware Report,” G DATA CyberDefence AG, [Online]. Available: https://www.gdatasoftware.com/news/2019/07/35228-mobile-malware-report-no-let-up-with-android-malware. [Accessed 03 Nov 2019].

[6] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for android.,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2011, pp. 15-26.

[7] T. Atkinson and L. Cavallaro, “Hunting ELFs: An Investigation Into Android Malware Detection,” 2017.

[8] M. La Polla, F. Martinelli and D. Sgandurra, “A Survey on Security for Mobile Devices,” *IEEE Communications Surveys & Tutorials,* vol. 15, pp. 446-471, 2013.

[9] G. Robinson and G. R. S. Weir, “Understanding Android Security,” *Communications in Computer and Information Science,* pp. 189-199, 2015.

[10] “Android (operating system) - Wikipedia,” Wikimedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/Android_(operating_system). [Accessed 10 Mar 2020].

[11] “HTC Dream - Wikipedia,” Wikimedia Foundation, 2020. [Online]. Available: https://en.wikipedia.org/wiki/HTC_Dream. [Accessed 10 Mar 2020].

[12] T. Lee, “Android Now Has 2 Billion Monthly Active Users,” Ubergizmo, [Online]. Available: http://www.ubergizmo.com/2017/05/android-2-billion-monthly-users. [Accessed 12 Mar 2020].

[13] O. S. I. U. Statisa, *Smartphone OS share installed base worldwide 2015-2017 | Statistic,* 2018.

[14] “Platform Architecture | Android Developers,” Alphabet Inc., [Online]. Available: https://developer.android.com/guide/platform/index.html. [Accessed 13 Mar 2020].

[15] "Android Runtime (ART) and Dalvik | Android Open Source Project," Alphabet Inc., [Online]. Available: https://source.android.com/devices/tech/dalvik. [Accessed 13 Mar 2020].

[16] E. Chi, A. P. Felt, K. Greenwood and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ACM, 2011, pp. 239-252.

[17] L. Davi, A. Dmitrienko, A.-R. Sadeghi and M. Winandy, "Privilege escalation attacks on android," in *13th International Conference, ser. ISC_10*, Springer Berlin Heidelberg, 2010, pp. 346-360.

[18] M. Dietz, S. Shekhar, Y. Pisetsky and D. S. Wallach, "QUIRE: Lightweight provenance for smart phone operating systems," in *USENIX Security Symposium*, 2011.

[19] S. Bugiel, L. Davi, A. Dmitrienko and B. Shastry, "Towards Taming Privilege-Escalation Attacks on Android," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[20] A. P. Felt, M. Finifter, E. Chin, S. Hanna and D. Wagner, "A survey of mobile malware in the wild.," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2011, pp. 3-14.

[21] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna and E. Chin, "Permission re-delegation: Attacks and defenses.," in *20th Usenix Security Symposium*, USENIX, 2011.

[22] R. Mayrhofer, J. V. Stoep, C. Brubaker and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security (TOPS),* vol. 24, no. 3, pp. 1-35, 2021.

[23] S. Hutchinson, B. Zhou and U. Karabiyik, "Are we really protected? An investigation into the play protect service," in *IEEE International Conference on Big Data (Big Data)*, 2019.

[24] A. Apvrille, "The evolution of mobile malware," *Computer Fraud & Security,* vol. 2014, pp. 18-20, 2014.

[25] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh and L. Cavallaro, "The Evolution of Android Malware and Android Analysis Techniques," *ACM Computing Surveys,* vol. 49, pp. 1-41, 2017.

[26] D. Maslennikov, E. Aseev and A. Gostev, *Kaspersky Security Bulletin 2009. Malware Evolution 2009,* 2009.

[27] A. Apvrille, "Symbian worm Yxes: towards mobile botnets?," *Journal in Computer Virology,* vol. 8, pp. 117-131, 2012.

[28] G. Statista, "Smartphone OS global market share 2009-2018 | Statistic," 2019. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. [Accessed 22 June 2020].

[29] S. Arshad, M. Ali, A. Khan and M. Ahmed, “Android Malware Detection & Protection: A Survey,” *International Journal of Advanced Computer Science and Applications,* vol. 7, pp. 463-475, 2016.

[30] B. Amro, “Malware Detection Techniques for Mobile Devices,” *International Journal of Mobile Network Communications & Telematics,* vol. 7, pp. 1-10, 2017.

[31] F. Wei, Y. Li, S. Roy, X. Ou and W. Zhou, “Deep Ground Truth Analysis of Current Android Malware.,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017, pp. 252-276.

[32] A. Azmoodeh, A. Dehghantanha, M. Conti and K.-K. R. Choo, “Detecting crypto-ransomware in IoT networks based on energy consumption footprint,” *Journal of Ambient Intelligence and Humanized Computing,* pp. 1-12, 2017.

[33] C. Hicks and G. Dietrich, “An exploratory analysis in android malware trends.,” in *AMCIS 2016: Surfing the IT Innovation Wave - 22nd Americas Conference on Information Systems Association for Information Systems*, Association for Information Systems, 2016.

[34] G. Suarez-Tangil and G. Stringhini, “Eight Years of Rider Measurement in the Android Malware Ecosystem: Evolution and Lessons Learned,” *arXiv preprint arXiv:1801.08115,* 2018.

[35] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca and G. Giacinto, “Clustering android malware families by http traffic.,” in *Proceeding MALWARE '15 Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE Computer Society, 2015, pp. 128-135.

[36] T. Chakraborty, F. Pierazzi and V. S. Subrahmanian, “EC2: Ensemble Clustering and Classification for Predicting Android Malware Families,” *IEEE Transactions on Dependable and Secure Computing,* pp. 1-1, 2017.

[37] A. Calleja, A. Martín, H. Menéndez, J. Tapiador and D. Clark, “Picking on the family: Disrupting android Android malware triage by forcing misclassification,” *Expert Systems with Applications,* vol. 95, pp. 113-126.

[38] P. Battista, F. Mercaldo, V. Nardone, A. Santone and C. A. Visaggio, “Identification of Android Malware Families with Model Checking.,” in *2nd International Conference on Information Systems Security and Privacy- ICISSP 2016*, 2016, pp. 542-547.

[39] Y. Li, T. Shen, X. Sun, X. Pan and B. Mao, “Detection, Classification and Characterization of Android Malware Using API Data Dependency,” *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering,* vol. 164, pp. 23-40, 2015.

[40] S. Rasthofer, I. Asrar, S. Huber and E. Bodden, “How current android malware seeks to evade automated code analysis.,” in *IFIP International Conference on Information Security Theory and Practice*, Springer, Cham, 2015, pp. 187-202.

[41] N. Y. Kim, J. Shim, S.-j. Cho, M. Park and S. Han, “Android Application Protection against Static Reverse Engineering based on Multidexing.,” *Journal of Internet Services and Information Security (JISIS),* vol. 6, pp. 54-64, 2016.

[42] X. x. Yuan, P. He, Q. Zhu, R. R. Bhat and X. Li, “Adversarial Examples: Attacks and Defenses for Deep Learning.,” *arXiv preprint arXiv:1712.07107,* 2017.

[43] A. Spottka, “Evaluating Dynamic Analysis Methods for Android Applications,” 2017.

[44] F. Tong and Z. Yan, “A hybrid approach of mobile malware detection in Android,” *Journal of Parallel and Distributed Computing,* vol. 103, pp. 22-31, 2017.

[45] S. Verma and S. K. Muttoo, “An Android Malware Detection Framework-based on Permissions and Intents,” *Defence Science Journal,* vol. 66, p. 618, 2016.

[46] S. Y. Yerima, S. Sezer and I. Muttik, “Android malware detection using parallel machine learning classifiers.,” in *Conference: 8th International Conference on Next Generation Mobile Applications, Services and Technologies (NGMAST 2014)*, IEEE, 2014, pp. 37-42.

[47] P. Faruki, V. Laxmi, A. Bharmal, M. Gaur and V. Ganmoor, “AndroSimilar: Robust signature for detecting variants of Android malware,” *Journal of Information Security and Applications,* vol. 22, pp. 66-80, June 2015.

[48] A. Rodríguez-Mota, P. Escamilla-Ambrosio, S. Morales-Ortega, M. Salinas and E. Aguirre, “Towards a 2-hybrid Android malware detection test framework,” in *2016 International Conference on Electronics, Communications and Computers (CONIELECOMP). IEEE*, 2016.

[49] K. Elish, X. Shu, D. Yao, B. Ryder and X. Jiang, “Profiling user-trigger dependence for Android malware detection,” *Computers & Security,* vol. 49, pp. 255-273, 2015.

[50] F. Martinelli, F. Mercaldo and A. Saracino, “Bridemaid: An hybrid tool for accurate detection of android malware.,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ACM, 2017, pp. 899-901.

[51] S. Chaba, R. Kumar, R. Pant and M. Dave, “Malware Detection Approach for Android systems Using System Call Logs,” *arXiv preprint arXiv:1709.08805,* 2017.

[52] A. Altaher and O. Barukab, “Intelligent Hybrid Approach for Android Malware Detection based on Permissions and API Calls,” *International Journal of Advanced Computer Science and Applications,* vol. 8, no. 6, pp. 60-67, 2017.

[53] A. Reina, A. Fattori and L. Cavallaro, “A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors.,” in *ACM European Workshop on Systems Security (EuroSec)*, ACM, 2013.

[54] K. Grosse, N. Papernot, P. Manoharan, M. Backes and P. McDaniel, “Adversarial examples for malware detection.,” in *European Symposium on Research in Computer Security*, Springer, 2017, pp. 62-79.

[55] S. Y. Yerima, I. Muttik and S. Sezer, “High accuracy android malware detection using ensemble learning,” *IET Information Security,* vol. 9, pp. 313-320, 2015.

[56] M. K. Alzaylaee, S. Y. Yerima and S. Sezer, “Emulator vs real phone: Android malware detection using machine learning.,” in *3rd ACM International Workshop on Security and Privacy Analytics (IWSPA 2017) , Co-located with ACM CODASPY 2017*, ACM, 2017, pp. 65-72.

[57] H. L. Thanh, “Analysis of Malware Families on Android Mobiles: Detection Characteristics Recognizable by Ordinary Phone Users and How to Fix It,” *Journal of Information Security,* vol. 04, pp. 213-224, 2013.

[58] B. Sanz, I. Santos, C. Laorden and P. G. Bringas, “On the Automatic Categorisation of Android Applications.,” in *Proceedings of the 9th IEEE Consumer Communications and Networking Conference (CCNC2012)*, IEEE, 2012, pp. 149-153.

[59] G. Suarez-Tangil, S. Dash, M. Ahmadi, J. Kinder, G. Giacinto and L. Cavallaro, “DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware.,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 309-320.

[60] K. O. Elish, D. (. Yao and B. G. Ryder, “On the need of precise inter-app ICC classification for detecting Android malware collusions.,” in *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[61] M. Zheng, M. Sun and J. C. Lui, “Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware.,” in *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, 2013, pp. 163-171.

[62] B. Baskaran and A. Ralescu, “A study of android malware detection techniques and machine learning.,” in *MAICS 2016 Conference*, 2016, pp. 15-23.

[63] N. Milosevic, A. Dehghantanha and K.-K. R. Choo, “Machine learning aided Android malware classification,” *Computers & Electrical Engineering,* vol. 61, pp. 266-274, 2017.

[64] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song and H. Yu, “SAMADroid: A Novel 3-Level Hybrid Malware Detection Model for Android Operating System,” *IEEE Access,* vol. 6, pp. 4321-4339, 2018.

[65] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu and B. Li, “Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach,” *Computers & Security,* vol. 73, pp. 326-344, 2018.

[66] 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.

[67] L. Chen, M. Zhang, C.-Y. Yang and R. Sahita, “Semi-supervised classification for dynamic Android malware detection,” *arXiv preprint arXiv:1704.05948,* 2017.

[68] K. Allix, T. Bissyandé, Q. Jérome and Y. Traon, “Large-scale machine learning-based malware detection: confronting the 10-fold cross validation scheme with reality,” in

Proceedings of the 4th ACM conference on Data and application security and privacy, San Antonio, Texas, USA, 2014.

[69] P. R. K. Varma, K. P. Raj and K. V. S. Raju, “Android mobile security by detecting and classification of malware based on permissions using machine learning algorithms.,” in *Conference: 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, IEEE, 2017, pp. 294-299.

[70] A. T. Kabakus and I. A. Dogru, “An in-depth analysis of Android malware using hybrid techniques,” *Digital Investigation,* 2018.

[71] C. Smutz and A. Stavrou, “When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors.,” in *Conference: Network and Distributed System Security Symposium*, NDSS, 2016.

[72] R. Spreitzer, S. Griesmayr, T. Korak and S. Mangard, “Exploiting data-usage statistics for website fingerprinting attacks on Android.,” in *9th ACM Conference on on Security and Privacy in Wireless and Mobile Networks*, ACM, 2016, pp. 49-60.

[73] Y. Han and B. Rubinstein, “Adequacy of the Gradient-Descent Method for Classifier Evasion Attacks.,” *arXiv preprint arXiv:1704.01704,* 2017.

[74] M. Zhang, Y. Duan, H. Yin and Z. Zhao, “Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale, Arizona, USA, 2014.

[75] J. Sahs and L. Khan, “A machine learning approach to android malware detection.,” in *Proceedings of the 2012 European Intelligence and Security Informatics Conference*, IEEE Computer Society, 2012, pp. 141-147.

[76] B. Amos, H. A. Turner and J. White, “Applying machine learning classifiers to dynamic android malware detection at scale.,” in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, IEEE, 2013, pp. 1666-1671.

[77] W. Zhou, Y. Zhou, M. Grace, X. Jiang and S. Zou, “Fast, scalable detection of piggybacked mobile applications.,” in *Proceedings of the third ACM conference on Data and application security and privacy*, ACM, 2013, pp. 185-196.

[78] L. Chen, S. Hou, Y. Ye and L. Chen, “An Adversarial Machine Learning Model Against Android Malware Evasion Attacks,” in *Web and Big Data*, 2017.

[79] M. Barreno, B. Nelson, R. Sears, A. Joseph and J. D. Tygar, “Can machine learning be secure?,” in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, March 2006.

[80] L. Suhuan and H. Xiaojun, “Android Malware Detection Based on Logistic Regression and XGBoost,” in *IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 2019.

[81] K. A. Dahri, M. S. Vighio and B. A. Zardari, “Detection and prevention of malware in android operating system,” *Mehran University Research Journal Of Engineering & Technology,* vol. 40, no. 4, pp. 847-859, 2021.

[82] S. Alsoghyer and I. Almomani, “On the Effectiveness of Application Permissions for Android Ransomware Detection,” in *2020 6th Conference on Data Science and Machine Learning Applications (CDMA)*, 2020.

[83] D. Dagon, T. Martin and T. Starner, “Mobile Phones as Computing Devices: The Viruses are Coming!,” *IEEE Pervasive Computing,* vol. 3, pp. 11-15, 2004.

[84] 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, pp. 961-987, 2014.

[85] A. Sadeghi, H. Bagheri, J. Garcia and S. Malek, “A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software,” *IEEE Transactions on Software Engineering,* vol. 43, pp. 492-530, 2016.

[86] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici and S. Dolev, “Google Android: A State-of-the-Art Review of Security Mechanisms,” *eprint arXiv:0912.5101,* 2009.

[87] B. Kitchenham, “Procedures for performing systematic reviews.,” *Keele University,* vol. 33, 2004.

[88] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” *Journal of Systems and Software,* vol. 80, pp. 571-583, 2007.

[89] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil and S. Furnell, “AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection,” *Computers & Security,* vol. 65, pp. 121-134, 2017.

[90] L. Li, T. Bissyandé, D. Octeau and J. Klein, “Reflection-aware static analysis of Android apps,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, 2016.

[91] L. Deshotels, V. Notani and A. Lakhotia, “DroidLegacy: Automated Familial Classification of Android Malware,” in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, San Diego, CA, USA, January 22 - 24, 2014.

[92] K. A. Talha, D. I. Alper and C. Aydin, “APK Auditor: Permission-based Android malware detection system,” *Digital Investigation,* vol. 13, pp. 1-14, 2015.

[93] H. Howard, A. Pfeffer, M. Dalai and M. Reposa, “Predicting signatures of future malware variants,” in *12th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, 2018.

[94] S. Alam, S. A. Alharbi and S. Yildirim, “Mining Nested Flow of Dominant APIs for Detecting Android Malware,” *Computer Networks,* vol. 167, 2019.

[95] P. Calderon, H. Hasegawa, Y. Yamaguchi and H. Shimada, “Malware Detection based on HTTPS Characteristic via MachineLearning,” in *In Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)*, 2018.

[96] G. Marín, P. Casas and G. Capdehourat, “Deep in the Dark - Deep Learning-based Malware Traffic Detection without Expert Knowledge,” in *2019 IEEE Security and Privacy Workshops (SPW)*, San Francisco, CA, USA, 2019.

[97] A. M. Lungana-Niculescu, A. Colesa and C. Oprisa, “False Positive Mitigation in Behavioral Malware Detection Using Deep Learning,” in *2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP)*, Cluj-Napoca, 2018.

[98] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross and G. Stringhini, “MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version),” in *ACM Transactions on Privacy and Security*, 2019.

[99] H. Alptekin, C. Yildizli, E. Savas and A. Levi, “TRAPDROID: Bare-Metal Android Malware Behavior Analysis Framework,” in *2019 21st International Conference on Advanced Communication Technology (ICACT)*, PyeongChang Kwangwoon_Do, Korea (South), 2019.

[100] O. Olukoya, L. Mackenzie and I. Omoronyia, “Permission-based Risk Signals for App Behaviour Characterization in Android Apps,” in *5th International Conference on Information Systems Security and Privacy*, Prague, Czech Republic, 23-25 Feb 2019.

[101] S. Sourav, D. Khulbe and N. Kapoor, “Deep Learning Based Android Malware Detection Framework,” *A PREPRINT,* 2019 arXiv:1912.12122.

[102] M. Fan, X. Luo, J. Liu, C. Nong, Q. Zheng and T. Liu, “CTDroid: Leveraging a Corpus of Technical Blogs for Android Malware Analysis,” *IEEE Transactions on Reliability,* vol. 69, no. 1, pp. 124-138, March 2020.

[103] S. Y. Yerima, S. Sezer, G. McWilliams and I. Muttik, “A new android malware detection approach using bayesian classification.,” in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, IEEE, 2013, pp. 121-128.

[104] F. A. Narudin, A. Feizollah, N. B. Anuar and A. Gani, “Evaluation of machine learning classifiers for mobile malware detection,” *Soft Computing,* vol. 20, pp. 343-357, 2014.

[105] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar and A. Mohaisen, “Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems,” in *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, Dallas, TX, USA, 2019.

[106] Z. Ma, H. Ge, Y. Liu, M. Zhao and J. Ma, “A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms,” *IEEE Access,* vol. 7, pp. 21235-21245, 2019.

[107] M. Kumaran and W. Li, “ Lightweight malware detection based on machine learning algorithms and the android manifest file,” in *IEEE MIT Undergraduate Research Technology Conference (URTC)*, Cambridge, MA, USA, 2018.

[108] M. Grace, Y. Zhou, Q. Zhang, S. Zou and X. Jiang, “RiskRanker: Scalable and Accurate Zero-day Android Malware Detection,” in *in International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[109] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti and M. Rajarajan, “Android Security: A Survey of Issues, Malware Penetration, and Defenses,” *IEEE Communications Surveys & Tutorials,* vol. 17, pp. 998-1022, 2015.

[110] S. Y. Yerima, G. McWilliams and S. Sezer, “Analysis of Bayesian classification-based approaches for Android malware detection,” *IET Information Security,* vol. 8, pp. 25-36, 2014.

[111] W. Li, J. Ge and G. Dai, “Detecting Malware for Android Platform: An SVM-Based Approach,” in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, New York, NY, 2015.

[112] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. Bringas and G. Álvarez, “PUMA: Permission Usage to Detect Malware in Android,” in *Herrero Á. et al. (eds) International Joint Conference CISIS’12-ICEUTE´12-SOCO´12 Special Sessions. Advances in Intelligent Systems and Computing*, Berlin, Heidelberg, 2013.

[113] K. Deepa, G. Radhamani and P. Vinod, “Investigation of Feature Selection Methods for Android Malware Analysis,” *Procedia Computer Science,* vol. 46, pp. 841-848, 2015.

[114] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie and W. Enck, “AppContext: differentiating malicious and benign mobile app behaviors using context,” in *Proceeding ICSE '15 Proceedings of the 37th International Conference on Software Engineering*, Florence, Italy, May 16 - 24, 2015.

[115] F. M. Darus, N. A. A. Salleh and A. F. Mohd Ariffin, “Android Malware Detection Using Machine Learning on Image Patterns,” in *2018 Cyber Resilience Conference (CRC)*, Putrajaya, Malaysia, 2018.

[116] S. Chen, M. Xue, Z. Tang and H. Zhu, “Stormdroid: A streaminglized machine learning-based system for detecting android malware.,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ACM, 2016, pp. 377-388.

[117] A. Sharma, S. K. Sahay and A. Kumar, “Improving the detection accuracy of unknown malware by partitioning the executables in groups.,” in *Advanced Computing and Communication Technologies*, Springer, 2016, pp. 421-431.

[118] A. Fatima, R. Maurya, M. K. Dutta, R. Burget and J. Masek, “Android Malware Detection Using Genetic Algorithm based Optimized Feature Selection and Machine Learning,” in *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, Budapest, Hungary, 2019.

[119] Y. Feng, S. Anand, I. Dillig and A. Aiken, “Apposcopy: Semantics-based detection of android malware through static analysis.,” in *Proceedings of the 22nd ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 576-587.

[120] H. Kang, J.-w. Jang, A. Mohaisen and H. K. Kim, “Detecting and Classifying Android Malware Using Static Analysis along with Creator Information,” *International Journal of Distributed Sensor Networks,* vol. 11, p. 479174, 2015.

[121] A. Kapratwar, F. D. Troia and M. Stamp, “Static and Dynamic Analysis of Android Malware.,” in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, (ICISSPs)*, 2017, pp. 653-662.

[122] A. Aiken, Static Analysis of Mobile Programs., 2017.

[123] N. Chavan, F. Di Troia and M. Stamp, “A Comparative Analysis of Android Malware,” *arXiv e-prints,* 2019.

[124] A. Mehtab, W. Shahid and T. e. a. Yaqoob, “AdDroid: Rule-Based Machine Learning Framework for Android Malware Analysis,” *Mobile Networks and Applications,* vol. 25, pp. 180-192, 2019.

[125] J. Jiang, S. Li, M. Yu, G. Li, C. Liu, K. Chen, H. Liu and W. Huang, “Android Malware Family Classification Based on Sensitive Opcode Sequence,” *IEEE Symposium on Computers and Communications (ISCC),* pp. 1-7, 2019.

[126] A. O. Gamao, “Malware Analysis on Android Apps: A Permission-based Approach,” *Social Sciences and Humanities Journal,* vol. 2, no. 10, pp. 624-633, 2018.

[127] L. Li, T. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein and L. Traon, “Static analysis of Android apps: A systematic literature review,” *Information and Software Technology,* vol. 88, pp. 67-95, 2017.

[128] M. Dimjašević, S. Atzeni, I. Ugrina and Z. Rakamaric, “Evaluation of Android malware detection based on system calls,” in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, New Orleans, Louisiana, USA, 2016.

[129] S. Türker and A. B. Can, “AndMFC: Android Malware Family Classification Framework,” in *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, Istanbul, Turkey, 2019.

[130] I. Muhammad, M. H. Durad and M. Ismail, “Static and Dynamic Malware Analysis Using Machine Learning,” in *16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, Islamabad, Pakistan, 2019.

[131] L. Apvrille and A. Apvrille, “Identifying unknown android malware with feature extractions and classification techniques.,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, IEEE, 2015, pp. 182-189.

[132] A. Kapratwar, “Static and Dynamic Analysis for Android Malware Detection.,” 2016.

[133] C. Uppin and G. George, “Analysis ofAndroid Malware Using Data Replication Features Extracted by Machine Learning Tools,” *International Journal of Scientific Research in Computer Science, Engineering and Information Technology,* vol. 5, no. 5, 2019.

[134] T. Kumar, S. Sharma, H. Goel and S. Chaudhary, “A Novel Machine Learning Approach for Malware Detection,” in *International Conference on Advances in Engineering Science Management & Technology (ICAESMT)*, Dehradun, India, 2019.

[135] D. Sang, D. Cuong and L. Cuong, “An Effective Ensemble Deep Learning Framework for Malware Detection,” in *The Ninth International Symposium on Information and Communication Technology (SoICT 2018)*, December 6–7, 2018.

[136] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo and C. A. Visaggio, “Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis,” in *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, Funchal, Madeira, Portugal, 2018.

[137] K. Chumachenko, “MACHINE LEARNING METHODS FOR MALWARE DETECTION AND CLASSIFICATION,” 2017.

[138] A. Rodríguez-Mota, P. Escamilla-Ambrosio and M. Salinas-Rosales, “Malware Analysis and Detection on Android: The Big Challenge,” *Smartphones from an Applied Research Perspective,* 2017.

[139] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun and J. Zhang, “Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique,” *IEEE Transactions on Information Forensics and Security,* vol. 12, pp. 1529-1544, 2017.

[140] F. Martinelli, F. Mercaldo, A. Saracino and C. A. Visaggio, “I find your behavior disturbing: Static and dynamic app behavioral analysis for detection of android malware.,” in *14th International Conference on Privacy, Security and Trust (PST 2016)*, IEEE, 2016.

[141] R. Sihwail, K. Omar, K. Zainol Ariffin and S. Al Afghani, “Malware Detection Approach Based on Artifacts in Memory Image and Dynamic Analysis,” *Applied Sciences,* vol. 9, no. 18, p. 3680, Sep 2019.

[142] A. Bulazel and B. Yener, “A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web.,” in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium Article No. 2*, ACM, 2017.

[143] S. Fallah and A. J. Bidgoly, “Benchmarking Machine Learning Algorithms for Android Malware Detection,” *Jordanian Journal of Computers and Information Technology (JJCCIT),* vol. 5, no. 3, pp. 216-229, 2019.

[144] W. Huang, Y. Dong, A. Milanova and J. Dolby, “Scalable and precise taint analysis for android.,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, 2015, pp. 106-117.

[145] M. Xia, L. Gong, Y. Lyu, Z. Qi and X. Liu, “Effective real-time android application auditing.,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2015, pp. 899-214.

[146] J. Du, H. Chen, W. Zhon, Z. Liu and A. Xu, “A Dynamic and Static Combined Android Malicious Code Detection Model based on SVM,” in *A Dynamic and Static Combined Android Malicious Code Detection Model based on SVM*, Nanjing, 2018.

[147] L. Xu, D. Zhang, N. Jayasena and J. Cavazos, “HADM: Hybrid Analysis for Detection of Malware,” in *Proceedings of SAI Intelligent Systems Conference (IntelliSys)*, 2016.

[148] Y. Liu , K. Guo, X. Huang, Z. Zhou and Y. Zhang, “Detecting Android Malwares with High-Efficient Hybrid Analyzing Methods,” *Mobile Information Systems,* 2018.

[149] A. Martin, R. Lara-Cabrera and D. Camacho, “Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset,” *Information Fusion,* vol. 52, pp. 128-142, 2019.

[150] R. Riasat, M. Sakeena, A. H. Sadiq and Y. Wang, “Onamd: An Online Android Malware Detection Approach,” in *2018 International Conference on Machine Learning and Cybernetics (ICMLC)*, Chengdu, 2018.

[151] M. K. Alzaylaee, S. Y. Yerima and S. Sezer, “DynaLog: An automated dynamic analysis framework for characterizing Android applications.,” in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, IEEE, 2016, pp. 1-8.

[152] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones.,” *ACM Transactions on Computer Systems,* vol. 32, pp. 1-29, 2014.

[153] P. Faruki, V. Laxmi, V. Ganmoor, M. S. Gaur and A. Bharmal, “DroidOLytics : Robust Feature Signature for Repackaged Android apps on Officialand Third party android markets,” in *2013 Second International Conference on Advanced Computing, Networking and Security*, Mangalore, India, 2013.

[154] P. Faruki, A. Bharmal, V. Laxmi and M. Rajarajan, “Evaluation of android anti-malware techniques against dalvik bytecode obfuscation.,” in *In Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, 2014, pp. 414-421.

[155] X. Zheng, L. Pan and E. Yilmaz, “Security analysis of modern mission critical android mobile applications,” in *Proceedings of the Australasian Computer Science Week Multiconference Article No. 2*, ACM, 2017.

[156] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao and T. Han, “MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph,” in *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, Shanghai, 2014.

[157] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez and J. Blasco, “Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families,” *Expert Systems with Applications,* vol. 41, pp. 1104-1117, 2014.

[158] R. Mahmood, N. Mirzaei and S. Malek, “EvoDroid: segmented evolutionary testing of Android apps,” in *FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[159] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han and W. Zou, “SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications,” in

SPSM '12: Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, 2012.

[160] B. Torregrosa Garc_a, “A framework for detection of malicious software in Android handheld systems using machine learning techniques.,” 2015.

[161] M. Lindorfer, M. Neugschwandtner and C. Platzer, “MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis,” *2015 IEEE 39th Annual Computer Software and Applications Conference,* 2015.

[162] Y. Zhou, Z. Wang, W. Zhou and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets,” in *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[163] A. Malhotra and K. Bajaj, “A Survey on Various Malware Detection Techniques on Mobile Platform,” *International Journal of Computer Applications,* vol. 139, pp. 15-20, 2016.

[164] L. Vaishanav, S. Chauhan, H. Vaishanav, M. S. Sankhla and R. Kumar, “Behavioural Analysis of Android Malware using Machine Learning,” *International Journal Of Engineering And Computer Science,* vol. 6, pp. 21378-21389, 2017.

[165] 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,* vol. 38, pp. 161-190, 2011.

[166] B. A. Mantoo and S. S. Khurana, “Static, Dynamic and Intrinsic Features Based Android Malware Detection Using Machine Learning,” *Springer, Cham,* vol. 597, 2020.

[167] H. T. Weldegebriel, H. Liu, A. U. Haq, E. Bugingo and D. Zhang, “A new hybrid convolutional neural network and eXtreme gradient boosting classifier for recognizing handwritten Ethiopian characters,” *IEEE Access,* vol. 8, pp. 17804-17818, 2019.

[168] S. Soviany, A. Scheianu, G. Suciu, A. Vulpe, O. Fratu and C. Istrate, “Android Malware Detection and Crypto-Mining Recognition Methodology with Machine Learning,” in *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, Bucharest, Romania, 2018.

[169] S. Alam, S. A. Alharbi and S. Yildirim, “Mining nested flow of dominant APIs for detecting android malware,” *Computer Networks,* vol. 167, 2019.

[170] A. Desnos, G. Gueguen and S. Bachmann, “Androguard,” Androguard, [Online]. Available: https://github.com/androguard/androguard. [Accessed 08 Nov 2019].

[171] “Python Data Analysis Library - pandas,” NumFOCUS, [Online]. Available: https://pandas.pydata.org/. [Accessed 09 Nov 2019].

[172] “MySQL,” Oracle Corporation, [Online]. Available: https://www.mysql.com/. [Accessed 09 Nov 2019].

[173] “Permissions Overview | Android Developers,” Alphabet Inc., [Online]. Available: https://developer.android.com/guide/topics/permissions/overview. [Accessed 02 Nov 2022].

[174] "Services overview | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/guide/components/services?hl=en. [Accessed 02 Nov 2019].

[175] "ClassLoader | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/reference/java/lang/ClassLoader. [Accessed 02 Nov 2019].

[176] "Security tips | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/training/articles/security-tips. [Accessed 02 Nov 2019].

[177] "Android NDK | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/ndk/. [Accessed 02 Nov 2019].

[178] "android.database.sqlite | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/reference/android/database/sqlite/package-summary.html?hl=en. [Accessed 02 Nov 2019].

[179] "Security with HTTPS and SSL | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/training/articles/security-ssl. [Accessed 02 Nov 2019].

[180] "Cryptography | Android Developers," Alphabet Inc., [Online]. Available: https://developer.android.com/guide/topics/security/cryptography. [Accessed 02 Nov 2019].

[181] P. Agarwal and B. Trivedi, "Machine learning classifiers for Android malware detection," *Data Management, Analytics and Innovation, Springer,* pp. 311-322, 2021.

[182] S. Uddin, A. Khan, M. E. Hussain and M. A. Moni, "Comparing different supervised machine learning algorithms for disease prediction," *BMC Medical Informatics and Decision Making,* vol. 19, 2019.

[183] G. Bonaccorso, Machine Learning Algorithms: Popular algorithms for data science and machine learning, 2nd Edition, Packt Publishing, 2018.

[184] "Scikit Learn," scikit-learn developers, [Online]. Available: https://scikit-learn.org/. [Accessed 11 12 2021].

[185] F. Nie, Z. Wang, R. Wang, Z. Wang and X. Li, "Adaptive local linear discriminant analysis," *ACM Transactions on Knowledge Discovery from Data (TKDD),* vol. 14, no. 1, pp. 1-19, 2020.

[186] G. Baldini and D. Geneiatakis, "A Performance Evaluation on Distance Measures in KNN for Mobile Malware Detection," in *6th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2019.

[187] R. Chen, Y. Li and W. Fang, "Android malware identification based on traffic analysis," in *International Conference on Artificial Intelligence and Security, Springer*, 2019.

[188] A. B. Shaik and S. Srinivasan, "A brief survey on random forest ensembles in classification model," in *International Conference on Innovative Computing and Communications, Springer*, 2019.

[189] S. I. Kayum, H. Hossain, N. Tasnim, A. Paul and A. A. Rohan, "Malware detection using neural networks," in *PhD diss.*, Brac University, 2021.

[190] "The Sequential Model," Keras, [Online]. Available: https://keras.io/guides/sequential_model/. [Accessed 12 11 2021].

[191] N. Viennot, E. Garcia and J. Nieh, "A measurement study of google play," in *ACM international conference on Measurement and modeling of computer systems*, 2014.

[192] Y. Zhou and X. Jiang, "Android Malware Genome Project," North Carolina State University, [Online]. Available: http://www.malgenomeproject.org/. [Accessed 20 10 2019].

[193] D. Arp, M. Spreitzenbarth, M. H¨ubner, H. Gascon and K. Rieck, "Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket," in *21st Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 23-26, 2014.

[194] "Android Malware Datasets," GitHub, [Online]. Available: https://github.com/traceflight/Android-Malware-Datasets. [Accessed 13 10 2021].

[195] "Google Play," Alphabet Inc, [Online]. Available: https://play.google.com/store. [Accessed 20 Oct 2019].

[196] "Apps APK," [Online]. Available: https://www.appsapk.com/. [Accessed 20 Oct 2019].

[197] "IDC - Smartphone Market Share - OS," IDC Corporate USA, [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os. [Accessed 03 Nov 2019].

[198] "App Stores: number of apps in leading app stores 2019," Statista GmbH, [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Accessed 03 Nov 2019].

[199] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina and E. Moser, "FSquaDRA: fast detection of repackaged applications.," in *IFIP Annual Conference on Data and Applications Security and Privacy*, Springer, Berlin, Heidelberg, 2014, pp. 130-145.

[200] R. Zachariah, K. Akash, M. S. Yousef and A. M. Chacko, "Android malware detection a survey," in *2017 IEEE International Conference on Circuits and Systems (ICCS)*, Thiruvananthapuram, 2017.

[201] P. Yan and Z. Yan, "A survey on dynamic mobile malware detection," *Software Quality Journal,* vol. 26, p. 891–919, 2018.

[202] R. Unuchek, *Mobile malware evolution 2016,* 2017.

[203] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev and C. Glezer, "Google Android: A Comprehensive Security Assessment," *IEEE Security & Privacy Magazine,* vol. 8, pp. 35-44, 2010.

[204] Z. Hui-Juan, Y. Zhu-Hong, Z. Ze-Xuan, S. Wei-Lei, C. Xing 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.

[205] A. P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, “Android permissions demystified.,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security, ser. CCS _11*, ACM, 2011, pp. 627-638.

[206] A. En.wikipedia.org, *Android (operating system),* 2018.

[207] “Manifest.permission,” Android Developers, [Online]. Available: https://developer.android.com/reference/android/Manifest.permission. [Accessed 10 09 2021].