Intelligent Mobile Malware Detection Using Permission Requests and API Calls

Moutaz Alazab[1], Mamoun Alazab[2], Andrii Shalaginov[3], Abdelwadood Mesleh[1], Albara Awajan[1]

[1]Faculty of Artificial Intelligence, Al-Balqa Applied University, Jordan
{m.alazab, wadood, a.awajan}@bau.edu.jo

[2]College of Engineering, IT & Environment, Charles Darwin University, Australia
Mamoun.alazab@cdu.edu.au

[3]Faculty of Information Technology and Electrical Engineering, Norwegian University of Science and Technology, Norway
andrii.shalaginov@ntnu.no

Abstract- Malware is a serious threat that has been used to target mobile devices since its inception. Two types of mobile malware attacks are standalone: fraudulent mobile apps and injected malicious apps. Defending against the cyber threats of mobile malware requires a strong understanding of the permissions declared in applications and application program interface (API) calls. In this paper, we propose an effective classification model that combines permission requests and API calls. As Android apps use a large number of APIs, we propose three different grouping strategies for choosing the most valuable API calls to maximize the likelihood of identifying Android malware apps: the *ambiguous* group, *risky* group, and *disruptive* group. The results demonstrate that compared with benign apps, malicious applications invoke a different set of API calls and that mobile malware often requests dangerous permissions to access sensitive data more often than benign apps. Empirical results obtained with a real malware dataset containing 27,891 Android apps suggest that our proposed method is effective at detecting mobile malware apps and achieves an F-measure of 94.3%. Our model can significantly assist in the process of malware forensic investigation and mobile application analysis.

Keywords: Mobile Malware, Malware Detection, Mobile Security, IoT, API Calls, Android Permissions.

1. INTRODUCTION

Mobile devices are now the most common means of accessing the Internet. The explosive growth of the Internet with recent increasing trends in automation using intelligent applications provides a fertile playground for malicious software (malware) attackers. The significant growth of cloud computing and the Internet of Things (IoT) worldwide has increased malware threats. Such malicious actions can impact the confidentiality, integrity, or availability of mobile systems (Farivar 2019). Cybercrime was estimated to cost $600 billion in 2018 worldwide, and the preferred choice of access for the majority of the world's population is a mobile device (Mcafee 2019). Mobile malware authors have taken malware that targets PCs and added new capabilities to create new threats on mobile platforms (Alazab 2014). Implementing forensic identification security controls will certainly lower the risk of digital systems of being compromised (Ahvanooey 2020).

Mobile malware is malicious software specifically designed to target mobile devices, such as smartphones and tablets. It refers to any kind of malicious code that affects the integrity and the functionality of the mobile system without the user's knowledge or consent; the types of malware include ransomware, trojans, worms, spyware, rootkits, and botnets (Alazab, M 2015). Mobile malware is increasingly sophisticated and presents a serious threat due to malicious activities, such as stealing user's data, sending premium messages, making phone calls, etc. Malware authors have turned their attention to

mobile devices, leading to an increase of 1,800% in mobile malware in 2016 (Caviglione et al. 2015). According to Check Point's international survey of 850 organizations, all the businesses surveyed had experienced a mobile malware attack. The anti-malware engine Kaspersky reported in 2019 that the number of users who encountered Android malware more than tripled to 1.7 million globally (Kaspersky 2019). McAfee Labs detected over 16 million mobile malware events in the third quarter of 2017 alone (Mcafee 2019), and Juniper stated that the malware available for Android increased by 400% (Juniper 2012). Since 2010, SophosLabs has observed more than 1.5 million Android malware apps (Sophos 2019).

Mobile malware is continuously updated with new features, shifting into new distribution methods and investing in the development of detection avoidance techniques, such as the use of obfuscation methods (Alazab, et al. 2012) stealth techniques and repackaging (Arshad et al. 2016). A recent study (Faruki 2015) indicated that the majority of Android malware is repackaged in other legitimate (popular) apps to bypass security barriers (Zhou& Jiang 2012). In the repackaging process, as illustrated in Figure 1, mobile malware authors download popular benign applications from the Google Play Store, decompile them, inject malicious content into them, and finally, reupload the injected apps in the third-party markets for user uptake.

Current anti-malware scanners are not effective against these evading techniques (Irshad et al. 2018; Li et al. 2018; Naway & Li 2019; Quarta et al. 2018; Visu et al. 2019). In (Zhou & Jiang 2012), four mobile security applications were tested on more than 1,200 Android malware apps. They found that existing mobile anti-malware applications cannot detect obfuscated or repackaged malware apps. Access control is usually considered a very good way to control the security of mobile devices, such as protecting against DoS attacks, but has many significant limitations (Kayes, et al. 2019a; Watters et al. 2016; Kayes, et al. 2019b; Watters et al 2019; Kayes, et al. 2019c).

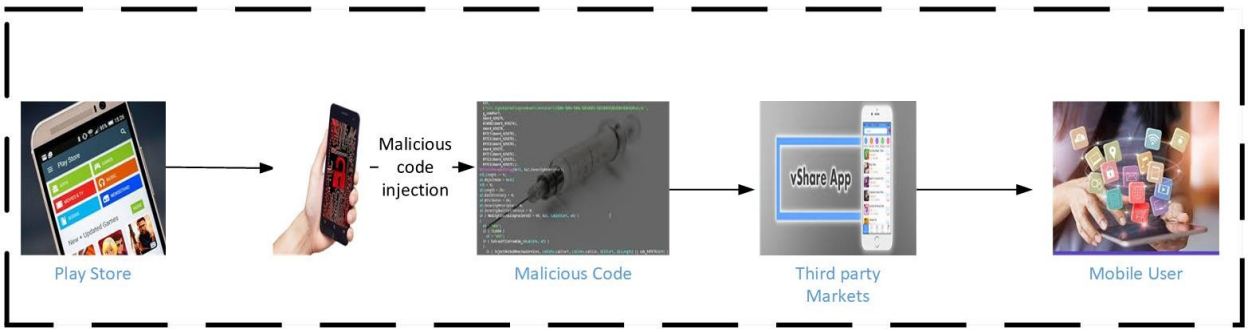

Figure 1. An overview of injection of malicious content

To assess the efficiency of mobile anti-malware scanners, in (Rastogi, Chen & Jiang 2013), researchers applied different obfuscation techniques in ten malicious apps from six different families, and they ran these new obfuscated binaries against 10 well-known anti-malware scanners. The result indicated that none of the anti-malware scanners could detect any of these malicious apps. With the large number of mobile applications, it is imperative to swiftly analyse and check the available applications in the marketplaces in an automated and intelligent fashion. It is crucial to establish an automated system that can identify and detect malicious applications in order to remove them from both official and non-official markets and make them unavailable for further download. In (Badhani & Muttoo 2018), eight obfuscation techniques of hiding malicious contents inside images were applied, and then malicious contents were injected into the resources of the Android application to check if the malware could be caught by ten anti-malware scanners. Their results confirmed that only one anti-malware scanner could detect two hiding techniques, while the rest of the anti-malware scanners failed to detect malicious content.

1.1 Vulnerability of Android smartphone devices

The Android-based embedded operating system platform is intended for use with low-power, memory-constrained IoT devices. The opportunities afforded by low-risk, low-cost, and profitable criminal activity attract cybercriminals (Thomas et al. 2015). Thus, Android's open and adaptable platform is more vulnerable to cyberattacks. As reported by (Oh et al. 2012), the main three reasons that the Android operating system is one of the most vulnerable platforms are the following: i) the openness of the Android platform, ii) the limited reviews of applications on the Google Play Store, and iii) the device's compatibility with applications from third-party vendors. As outlined below, these reasons explain why mobile malware is prominent and persistent.

- *Performance and profit.* Smart mobile devices, such as smartphones and tablets, are becoming an essential tool in today's world. According to the real-time intelligence data of the Global System for Mobile Communications Association (GSMA), there are now over 5.13 billion people with mobile devices worldwideBankmycell 2019). Thus, 66.53% of the world's population has a mobile device (cell phone, smartphone, tablet or cellular-enabled IoT device). Mobile users share sensitive data with apps all the time, such as heart rate data from fitness apps; information from banking apps, shopping apps and social media apps; and email messages, photos, etc. Mobile devices are profitable targets, and cybercriminals can acquire extensive financial and non- financial gain from such sensitive data. Mobile malware offers a high return with little investment. According to McAfee Mobile Threat Report Q1, 2018, mobile malware could create revenue for malware authors that could reach the billion-dollar range by 2020 (Mcafee 2019).

- *Usage and open source.* Since its launch in 2008, Android has grown to be one of the top-selling smartphones. The Android OS, similar to the other most popular OSs, such as iOS and BlackBerry, has expanded from running on smartphones to include tablets, music players, and other Android devices, thus making them available to a larger audience. The popularity of smartphones has grown exponentially and is still growing. Google reported that there are currently 2.5 billion active Android devices (Brandom 2019). With the high sales rates for smartphones, the market for applications running on these platforms also grows exponentially. As a consequence of their accessibility and widespread use globally, smartphones have become the new target for cybercriminal activities. Android is an open-source operating system for mobile devices and a corresponding open-source project led by Google. With Google Android's policy of an open-source kernel, malware authors potentially can gain a strong understanding of the mobile platform; this creates opportunities for cybercriminals to create and propagate mobile malware. Further, with the increased number of users downloading and installing apps, the likelihood of installing mobile malware increases as well.

The Google Play Store (previously called the Android Market) provides users with centralized access to download many types of apps (either free or paid). Recent reports show that the number of apps in Google Play Store was 2.9 million in December 2019 (Clement 2019). The sole Play Store is administered by Google. It has been reported by (Allix et al. 2016a) that the Google Play market might contain malware applications. A high number of third-party vendors and a non-Google market also started appearing, such as AppBrain, AppChina and Aptoide, that offer Android applications for users to download. The applications available in markets outside of the Google Play Store are managed by third parties, are at high risk of containing potentially malicious contents, and are not being monitored (Alazab & Batten 2015; Alazab et al. 2012; Moonsamy., Alazab. & Batten. 2012). The research work by (Allix et al. 2016b) shows that 22% of the apps in the Google Play Store have been identified as malware by at least one anti-malware program, while 50% of the apps in AppChina have been flagged as malware by at least one anti-malware software product. Mobile authors deconstruct and decompile popular apps, publish malicious versions and make them available for free in those third-party markets; this technique is known as "repackaging".

1.2 Our work

This paper focuses on the permissions and frequency distribution of API calls in order to differentiate malicious applications from non-malicious applications. The proposed methodology offers an automated tool for testing and evaluating Android applications using various statistical tests. A set of experiments related to the Android libraries in the software development kit (SDK) with static feature methods is provided. The main aims of this work can be summarized as follows:

- To introduce an efficient approach for describing Android malware that relies on the API in all packages and requested permissions.
- To examine the permissions and API call frequency distributions in order to classify applications as benign or malicious. Our findings show that the proposed approach can determine the similarities among malware families.
- To propose an efficient classification model for detecting mobile malware or risk factors of mobile malware. Our research findings demonstrate that API calls and permissions play a significant factor in classifying malware variants.
- To provide valuable insights about malware behaviour based on API calls and permission requests.
- To perform an in-depth analysis of different public and private packages, classes, and methods with the purpose of evaluating our proposed method in terms of the efficacy in dealing with large datasets and accuracy.

1.3 Paper organization

We have organized the overall structure of the paper as follows: Section 2 presents background about Android application permissions and API calls. Section 3 provides an overview of the related literature. Section 4 describes the dataset. Section 5 outlines our methodology and proposes a complete detection system. In Section 6, we perform some statistical analyses. In Section 7, we perform the classification and evaluate the model. Finally, we provide our conclusions and highlight the limitations of the study in Section 8.

2. BACKGROUND

2.1 Android application permissions

Android's operating system is responsible for a number of control access mechanisms. For instance, the operating system can allow or deny a specific process during the running time. The core security of the Android applications is the permission system, which protects the smartphone's resources; each application has a list of permissions needed to access resources. Table 1 presents the list of permission groups with the related permissions and levels. Assigning permissions to an Android application specifies an access policy to mobile resources. In the situation in which an application demands to interact with other applications or with resources, proper permission must be granted from the mobile user. A list of permissions is displayed during the installation time or during running time. The user has the authority to accept or ignore the installation of an application. Google continuously adds permissions to protect sensitive data and certain system features in every released version of a new API level. For example, the version of Android 4.4, KitKat, comes with 145 permissions pre-installed, while the earliest Android version—API level 1—contained only 76 permissions. The regular mobile user cannot be expected to understand the semantics and implications of all the associated permissions. The authors in (Gao et al. 2019) confirm that few users pay attention to, understand, and act on permission information during installation, which implies that the permission technique does not help users make appropriate decisions.

The main purpose of permission is to protect user's privacy. For Android mobile apps to perform, they must request permission to access sensitive user data (such as SMS, calls, etc.) and some system features (such as the camera and GPS location). Permissions are granted at the time of installation or through

runtime requests and give an application the ability to access relevant resources. The protection levels vary, and there are four levels:

1. **Normal Permission** is considered low risk (*i.e.,* permission to set the alarm).
2. **Dangerous Permission** is considered high risk (*i.e.,* permission to access the camera, SMS, camera, contacts, location, microphone, sensors, and storage).
3. **Signature Permission** is considered critical risk (*i.e.,* gives access if the application requesting access to the resources is signed with the same credentials as the permission).
4. **Signature or System Permission is** considered critical risk (*i.e.,* gives access to the applications having the same credentials, as in the previous level, or if they are in the system image).

Table 1. List of Android Permissions

PERMISSION GROUP	RELATED PERMISSION(S)	PERMISSION LEVEL
ALARM	- Set_Alarm - Set_Time_Zone	- Normal Permissions
MICROPHONE	- Record_Audio	- Dangerous Permissions
BLUETOOTH	- Bluetooth - Bluetooth_Admin	- Normal Permissions
CALENDAR	- Read_Calendar - Write_Calendar	- Dangerous Permissions
CAMERA	- Camera	- Dangerous Permissions
CONTACTS	- Get_Accounts - Read_Contacts - Write_Contacts	- Dangerous Permissions
LOCATION	- Access_Coarse_Location - Access_Fine_Location	- Dangerous Permissions
NETWORK	- Access_Network_State - Access_Notification_Policy - Access_Wifi_State - Change_Wifi_Multicast_State - Change_Wifi_State	- Normal Permissions
PHONE	- Read_Phone_State - Read_Phone_Numbers - Call_Phone - Answer_Phone_Calls - Read_Call_Log - Write_Call_Log - Add_Voicemail - Process_Outgoing_Calls	- Dangerous Permissions
SENSORS	- Body_Sensors	- Dangerous Permissions
SMS	- Receive_SMS - Read_SMS - Receive_Wap_Push - Receive_MMS	- Dangerous Permissions
STORAGE	- Write_External_Storage - Read_External_Storage	- Dangerous Permissions
WALLPAPER	- Set_Wallpaper - Set_Wallpaper_Hints	- Normal Permissions

With Android malware, the application frequently requests the permissions to receive, write, and send text messages as well as several other permissions during its installation phase. In the Android 6.0 and higher released versions, Google introduced the granting of dangerous permissions at runtime and shows the various categories of permissions. Google does not, however, deliver full documentation of the manner in which they classified those permissions as normal permissions or dangerous permissions. Permissions are ordered into groups related to a device's capabilities or features, and the list of permission groups with the related permissions and levels are listed in Table 1.

Several research papers such as (Brodeur 2012; Moonsamy & Batten 2012; Shao et al. 2016) have shown that some applications with no permissions can still access sensitive information, such as taking pictures in the background and recording keystrokes. (Shao et al. 2016) proposed a static tool, "Kratos", that can detect inconsistencies in security enforcement within the Android framework. They found malicious applications with no permissions or low-privileged permissions can lead to security breaches, such as crashing the entire Android runtime environment, arbitrarily ending phone calls, and setting up an HTTP proxy. Hence, the permissions feature alone might not provide the best picture of reality. As summarized by (Aafer et al. 2013), permission mechanisms have several limitations:

– The existence of a certain permission in the app manifest file does not necessarily mean that it is actually used within the code.

– A large number of requested permissions are actually not used within the application's code itself but, rather, are required by the advertisement packages.

– Malware can perform malicious behaviour without employing any permission (Shao et al. 2016).

2.1 **Android API calls**

In the research literature on malware targeting personal computers, one important detection approach has been to leverage information from the API calls for understanding what the code does (Alazab & Batten 2015; Alazab et al. 2011; Jung et al. 2018; Xie et al. 2019). The API calls can hypothetically apprehend the malicious activity since the API calls reflect how an application interacts with the system. Android's platform provides an API framework that apps can use to interact with the underlying Android system. The literature review demonstrates the use of API calls to learn the interaction behaviours between applications and the Android system and shows that this method is very helpful for distinguishing benign and malware apps. Hence, it is crucial to include other behavioural app features such as API calls and investigate whether a specific API call requires dangerous permissions or normal permissions. Additionally, it is necessary to conduct an in-depth study of the prominent API calls requested by the malware apps and check if those API calls are protected by the proper permission or not.

3. RELATED WORK

The analysis and detection of mobile malware has been heavily researched in the last decade. This section examines issues related to identifying mobile malware and explores significantly related approaches that have been proposed by the literature. There are three methods of extracting features from mobile apps that are frequently used by security vendors and researchers: static, dynamic, and hybrid (a combination of static and dynamic). As indicated by (Alazab 2015; Venkatraman & Alazab 2018), compared to dynamic analysis, static analysis is faster and more effective due to its advantages from the information captured relating to structural properties, such as the sequence of byte "signatures" and anomalies in the file content. Dynamic analysis can be effective with runtime information, such as running process or by using the control flow graph that could be less prone to obfuscated malware. Static analysis provides a greater understanding of the source code of the application under investigation. Dynamic analysis, however, provides a greater understanding of the behaviour of the application under investigation.

3.1 **Static and dynamic analysis**

Recent methods that have been designed to understand Android applications have focused on Android permissions. (Shao et al. 2016) proposed a static tool called "Kratos" to discover inconsistencies in security enforcement within the Android framework. The authors found that malicious applications with no permission or low-privileged permission can lead to security breaches such as crashing the entire Android runtime environment, arbitrarily ending phone calls, and setting up an HTTP proxy. Hence, the permissions as a feature alone might not provide the best picture of reality.

In (Yerima & Khan 2019), 350 features of API calls, permissions, intents, and other attributes were extracted, such as command strings and the presence of embedded executables, using the Android package (APK) analysis tool. The authors collected a dataset of 13,805 malicious applications and 22,378 benign applications that were first seen in the period from 2012-2016. The primary objective of their work was to explore the identification of malware over an extended period of time; they separated the dataset into four groups: 2012, 2013, 2014 and 2015-2016. The authors employed the following machine learning algorithms: naïve Bayes, support vector machines, random forest, J48 decision trees and simple logistic methods. To evaluate the detection accuracy over the time period, each of the machine learning algorithms was implemented in each of the four groups (2012, 2013, 2014 and 2015-2016). Their research findings demonstrated that the detection accuracy rate becomes much less accurate in distinguishing benign applications over the time period. The authors argue that some of the features could become less discriminative as more evasion techniques start appearing more frequently in malicious applications, thus making it challenging to extract the relevant features associated with malware apps. Hence, we focus throughout this paper on enforcing correlative strategies to isolate the most requested features of both API calls and permissions from the malware apps and then build a complex detection mechanism.

In (Yang et al. 2014), the researchers proposed a tool named DroidMiner that applied static analysis to detect malicious applications and then classified them into families. Their tool extracts API calls and recognizes programming logic segments in the graph that correspond to known suspicious behaviour. The researchers evaluated their tool using 10,403 benign applications and 2,466 malicious applications that belong to 68 malware families. To test their methodology, they included four machine learning algorithms: naïve Bayes, support vector machines (SVM), decision trees and random forest, and in testing, the algorithms obtained 82.2%, 86.7%, 92.4% and 95.3% average accuracies, respectively.

In (Shabtai et al. 2011), a framework called Andromaly was proposed that could detect malware on Android mobile devices. The components of the proposed system were clustered into four main groups: feature extractors, processors, the main service, and the graphical user interface (GUI). The researchers applied three feature selections: chi square (CS), Fisher score (FS), and information gain (IG). To classify the collected apps as being either malicious or benign, the researchers applied six machine learning algorithms. They conducted four different experiments. The first experiment used the same applications for training and testing the classifiers on the same device. In the second experiment, different applications were used in training and testing the classifiers on the same device. The third experiment used the same applications for training and testing the classifiers on different devices. In the fourth experiment, different applications were used in training and testing the classifiers on different devices. The first experiment achieved the highest scores using the decision tree algorithm, achieving 99% accuracy. The limitation of this work was the class imbalance problem due to the inclusion of 4 self-written malicious applications and 40 benign applications.

A significant number of studies have focused on finding the similarities and differences between two applications based on the analysis of API calls. For instance, in (Kanda et al. 2011), the researchers proposed a tool that can find the differences between two Android applications based on the API calling sequence by extracting only the libraries related to the Android. The authors selected two case studies to investigate whether the API calling sequence can be used to differentiate between two Android applications. Their results showed that API calling sequences can be useful in comparing the two

applications. They extracted the methods, whose fully qualified names start with “android” or “com.google.android”, as the Android API. The Android SDK, however, contains several libraries: Java library, Apache HttpClient library and Google library. Hence, our approach is not limited to the methods that start with “android” or “com.google.android” but also includes permissions and other libraries, such as “Android”, “Dalvik”, “java”, “javax”, “junit”, and “org”. A reference for these libraries is accessible from the “android.jar" in the Android SDK. Furthermore, our system is designed to find similarities and differences in a large set of apps rather than by comparing two Android applications.

The system calls have been used to monitor the behaviour of unknown malware. Several studies confirm the efficiency of their tools when profiling the system calls with self-written malware (malware that has been written by the authors) (Burguera, Zurutuza & Tehrani 2011). The detection accuracy rate is 100% for self-written malware. Nevertheless, when profiling the system calls with the real malware, the detection rate decreases, as shown when testing the HongToutou family (Labs 2014), where the test achieved a detection rate of 85%. The researchers in (Dimjašević et al. 2016) tested profiling the system calls with real malware apps collected from the wild; they achieved a 93% detection accuracy rate with a 5% benign application classification error.

3.2 **Hybrid analysis**

Hybrid analysis is a combination of static and dynamic analysis. The work of (Kaushik & Jain 2015) applied static analysis to extract the requested permissions from the manifest file using AAPT (Android asset packaging tool); they also applied dynamic analysis to trace system calls using the Strace tool. The researchers collected a dataset of 108 benign applications and 112 malicious applications. The authors combined the features from both static analysis and dynamic analysis. For the purpose of testing their tool, they included 4 different machine learning algorithms; a detection accuracy of 70.31% was the highest achieved. The work conducted by (Kapratwar, Di Troia & Stamp 2017) consisted of static and dynamic analysis based on permissions and system calls on 103 malicious applications and 97 benign applications. The authors extracted 135 permissions statically using the APK tool. IG was used to retrieve the top 87 permissions. The authors made use of the Android emulator together with the Strace tool to record the system calls dynamically. The authors used the frequency distribution of system calls to check if a system call invoked more in a malicious application than in a benign application. To test their methodology, they included 6 different machine learning algorithms. They achieved high detection accuracy using a random forest, with accuracy rates of 0.972 for static analysis and 0.884 for dynamic analysis. The authors concluded that a static feature based on permissions is significantly more informative than a dynamic feature based on system calls. Thus, we opt in this paper to use static analysis to investigate the API calls in conjunction with permissions-based features.

3.3 **Deep learning**

There have been attempts to apply deep learning (Alazab & Tang 2019) and machine learning based on the relationship of API calls to identify the different behavioural patterns of malicious and benign apps to build a detection system. The authors in (Zhang et al. 2019) achieved a 96% detection accuracy rate on a dataset of Drebin (benign 5.9K and malware 5.6K) and AMD (benign 20.5K and malware 20.8K). The authors of (Fereidooni et al. 2016) presented an automated tool they call *uniPDroid* that uses both static analysis and machine learning algorithms to classify malevolent applications families. They collected a dataset of 15,884 malicious applications. Using static analysis, they extracted 560 features. The authors used extra-trees classifiers and meta-transformer for selecting the best features. To evaluate their tool, they applied the XGBoost classifier to 78 different malware families and obtained 92% average classification accuracy.

The work of (Karbab et al. 2018) investigated the effectiveness of the raw sequences of API calls and deep learning techniques to detect malicious applications, and their tool is known as “*MalDozer*”. The researchers extracted the sequences of API method calls using Dexdump from 33,000 malicious

applications from Malgenome, Drebin, and MalDozer and 38,000 benign applications from the Google Play Store. To normalize the features vector, MalDozer employs two word embedding techniques: word2vec and GloVe. The authors achieved 96% to 99% detection accuracy and a false positive rate range from 0.06% to 2%. The authors did not consider other features of API calls, such as permission requests. In our work, we used both permissions and frequency analysis of API calls from the Android platform, such as packages, classes, constructors, and methods.

An in-depth study by (Naway & Li 2019) presents a deep learning approach for Android malware detection. The authors extracted permissions, intent filters, invalid certificates, the presence of APK files in the asset folder and API calls using a mobile security framework (MobSF). Then, they converted all five features to vector space. To evaluate their tool, they applied a neural network on 600 benign applications and 600 malicious applications. They used 80% for training and 20% for testing and obtained a 96.81% detection accuracy. Similarly, in our case, we integrate various kinds of features, such as the frequency of API calls and permissions, to build an automated detection mechanism.

4. DATASETS

As reported in Table 2Feil: Fant ikke kilden til referansen, we conduct the evaluation experiments under two types of datasets: i) a benign dataset, which contains benign apps, and ii) a malware dataset, which contains only malware apps. For the malware dataset, we leverage reference datasets, such as AndroZoo, Contagio, MalShare, VirusShare and VirusTotal. The total number of malicious apps in this dataset is 13,719; all of the malware dataset was scanned and flagged as malware by at least 10 anti-malware products in the VirusTotal. For the benign dataset, since there is no standard benign dataset, we generated our own dataset and then scanned it using VirusTotal to confirm its cleanness. The benign applications were collected from the PlayStore using AndroZoo. The total number of benign apps in this dataset is 14,172. The app topics are diverse to reflect the variety of applications collected in June 2019 and checked using VirusTotal; if all of the anti-malware vendors in VirusTotal identified an app as benign, we identify it as benign. We used the SMOTE (synthetic minority over-sampling technique) to address the class imbalance problem.

Table 2. Datasets

Mobile application	Number
Malware	13,719
Benign	14,172
Total	**27,891**

5. METHODOLOGY

The accuracy of mobile malware detection is based on how the permission request and API call behaviours exhibited by applications with malicious code could be extracted and correlated. To maximize the likelihood of identifying malware apps, different from previous works, we use the permissions and frequency analysis of API calls. Our proposed system is based on an automated process by using a scoring and grouping technique to identify the most important API calls requested by the Android malware. Detecting repackaged applications by matching the name, hash value, or blacklist database is an ineffective method. Instead, our proposed system can identify similar repackaged applications by comparing the frequency distributions of API calls and permissions between two applications. For analysis purposes, we model our proposed mobile malware analysis method to consist of three main phases as follows:

Phase 1: Pre-processing phase.

Phase 2: Extraction phase.

Phase 3: Grouping phase.

5.1 **Pre-processing phase**

Android applications are written in Java, compiled into Java bytecode, and translated into DEX (Dalvik executable) bytecode (Dalvik virtual machine (VM)). The compiled Java code generates a number of files with the .class extension. Using the dx tool, the class files are merged into a single .dex file. The Android app is packaged into the APK content that is stored in binary format. Before any analysis, it is important to decompile the Android app. There are many reverse engineering tools to disassemble or decompile the Android app, such as Apktool, dex2jar, JADX, and Android MultiTool.

In this phase, we use Androguard (Desnos 2019), an open-source, static analysis reverse engineering tool, to disassemble and decompile Android apps. We used Androguard for a number of reasons:

- It is a powerful reverse engineering tool that can generate the control flow graphs for each method and provide access through Python-API on the command line and graphic interface.
- It can be implemented in Linux and Windows operating systems.
- It is able to identify hidden byte code at specific offsets.
- It enables the automatic extraction of specific packages, classes, constructors, methods and fields.

5.2 **Extraction phase**

The Android SDK provides developers with a framework of API calls (consisting of a core set of packages, classes, constructors, methods, and fields) to interact with the system, application or hardware. There are numerous APIs provided by the SDK that developers can use when developing the application. These API calls can be used illegally by malicious authors in ways that exploit mobile devices. For example, the benign or malicious app might request the same API call to access and retrieve specific data from an operating system. The Android SDK comprises various libraries such as Android, Dalvik, Java, Javax, Com, Java, Junit and Org, as shown in Figure 2. A reference for these libraries can be accessed from the “android.jar" in the Android SDK.

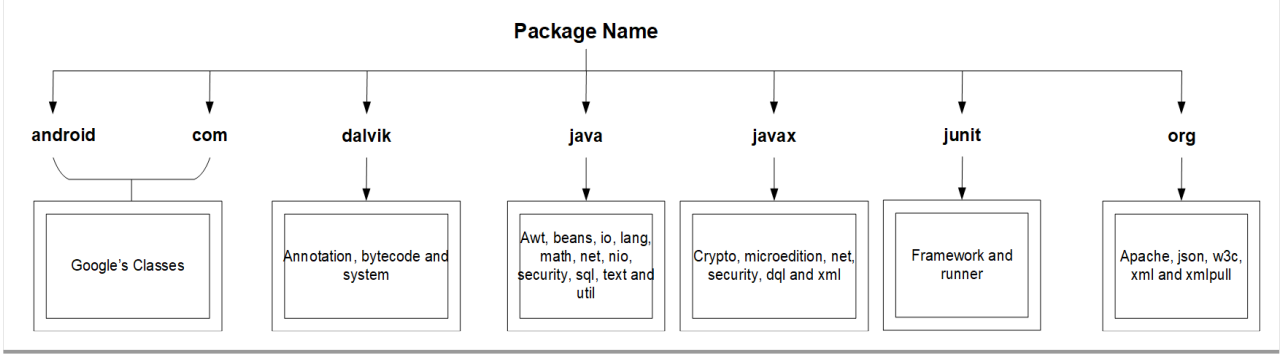

Figure 2. The Android Packages

Each Android application is mapped to a set consisting of the permission features and the API call features. For extracting API calls and permission requests from the APK files, we adopted the following process. A Python script has been developed to automatically execute and decompile the entire dataset as follows:

1. Generate all distinct packages invoked within each APK using Androguard.

2. Extract API calls and their package level information from entire packages, such as Java, Android and other packages; the main reason for including entire packages is that some contain significant methods and classes. More importantly, some of these sensitive API calls are protected by the Android permissions, which means they are important. For instance, the API entitled

"*android.webkit.resource.VIDEO_CAPTURE*" is protected by the following permission: "*android.permission.CAMERA*"

3. Extract the requested permissions of the apps; we adopted the same approach as presented in (Qiao et al., 2016) and define the set of all Android permissions and all of the requested API calls as follows:

$$P_i = \{P_1, P_2, \ldots\ldots, P_n\}$$
$$D_i = \{D_1, D_2, \ldots\ldots, D_n\}$$

4. Represent each application as a binary vector of API calls, namely, App_i, where

$$App_i, = \{1, if\ i\ th\ API\ is\ used \in the\ app, \wedge ¿ 0, if\ corresponding\ app\ does\ not\ use\ the\ API$$

5. To map between API calls D_i and permissions P_i, we define the association map as follows:

$$A = \{(P, D) \mid P \in P, D \in D, where\ D\ is\ controlled\ by\ P\}$$

6. Compute the number of API calls to each permission, represented as *MP*, and the numerical $C_i$to count each API call to each permission as follows:

$$MP = \{MP_1, MP_2, MP_3, \ldots., MP_n\}, where$$
$$MP_i, = \begin{cases} 1, \wedge if\ \exists D_i \\ 0, \wedge if\ \exists \not\ni D_i \end{cases}$$
$$C_i = \sum D_i \vee (P ¿ ¿ i, D_i) ¿$$

5.3 **Grouping phase**

To provide an expansive coverage of the detection performances, we employed a grouping strategy to further reveal subtleties of the use of API calls in malicious apps. We constructed groups of API calls that were capable of detecting malicious apps to a high degree. We labelled the API calls that appeared in the benign apps, considered inconsequential, as being irrelevant calls. We focused specifically on the API calls that appeared more frequently in the malicious applications. The features that we considered were prominent in the malicious apps. The aim of this system was to implement complementary procedures that circumvent fingerprinting the malware apps, with the goal of grouping three significant levels of API calls and categorizing them based on their threats, as shown in Figure 3.

We implemented a complete set of 27,147 API calls for the analysis because each call in the group could potentially be used by malicious applications to perform numerous jobs in the system. We defined 3 such groups according to their appearance in the malicious apps as follows: ambiguous API calls (A3), risky API calls (A2) and disruptive API Calls (A1).

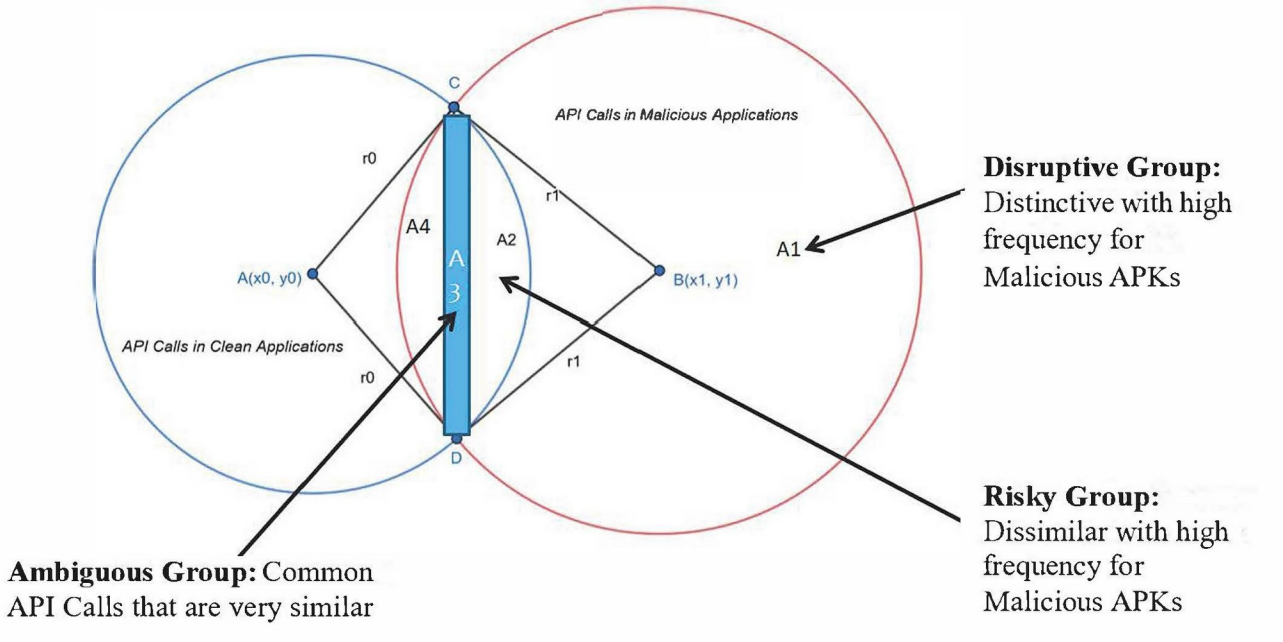

Figure 3. Architecture for ranking the API Calls

- Ambiguous group (A3)

We extracted the intersections of the API calls that are commonly used by both benign and malicious apps and that have almost the same number of occurrences of API calls in the entire set of benign and malicious applications. While considering the API calls exhibited in both benign and malware apps, the frequency of each API call is also included in both categories. To clarify, we considered how many times each API call appears among the benign applications. We employed the same reasoning for malicious calls. Thus, let $C=\{C1, C2, \ldots Ci\}$ be the set of API calls that appear in the benign apps with their frequency, and let $M=\{M1, M2, \ldots Mj\}$ be the set of API calls that appear in the malware apps with their frequency. Consequently, we could isolate the ambiguous group by extracting the API calls that had almost the same number of appearances in both benign and malicious applications. Figure 3 shows the ambiguous calls as Area 3 (A3).

From a set theory approach, we used the intersection operation to distinguish between benign and malicious groups; the intersection operation finds those API calls that are common and similar. A change in the frequency of the API calls, however, introduced a threshold value; due to this threshold value, the determination of benign or malicious API calls belonging to the ambiguous or risky groups vacillates. Only, 2,368 calls out of 27,147 appear in the ambiguous group.

- Risky group (A2)

We extracted the intersection of the API calls that were found more often in the malware apps than in the benign applications. We refer to this group as the 'risky API calls' group since malicious applications constantly invoke these API calls rather than benign apps. Let $C=\{C1, C2, \ldots Ci\}$ be the set of API calls that appear in the benign apps with their frequency; let $M=\{M1, M2, \ldots Mj\}$ be the set of API calls that appear in the malware apps with their frequency. Since we had the number of appearances for each API call in both benign and malicious applications, we could extract Area 2 (A2) by extracting the API calls that had a higher frequency in the malicious files than in the benign files, as shown in Figure 3.

In the 'risky' group, we implemented the same set theoretical intersection operation between benign and malicious groups as that employed for the first group. The distinctive feature of this group, however, was that the frequency of such API calls for malicious apps was greater than API calls of clean apps. The

intersection operation finds those API calls that are more similar; however, a change in the frequency of the API calls introduces the need for a threshold value. Moreover, even with the threshold value, there is less of a propensity for API calls belonging to benign apps to be active, determined, and prominent. In other words, there is more of a propensity towards malicious API calls.

Let C and D be two points at the intersections between the two circles (malicious and benign circles). Let A be the centre of the benign circle$(x_0, y_0)$of radius r_0 and B be the centre of the malware circle $(x_1, y_1)$of radius r_1.We want to calculate the three sub-areas in the intersection portion $(A_2, A_3 \wedge A_4)$, as shown in Figure 3. The area comprises the three sub-areas$(A_2, A_3 \wedge A_4)$: the left, centre and right portions of the intersection. The ambiguous API calls can be found in area A3, while the risky API calls can be found in area A2.

Step 1: To calculate the entire intersection area, we can compute

$$Area = (A_2 + A_3 + A_4)$$

Step 2: To calculate the three sub-areas $(A_2, A_3 \wedge A_4)$, we can compute

$$A_2 = A_{pie}(CBD) - A_{CBD}$$

$$A_3 = A_{pie}(CD) - A_{CD}$$

$$A_4 = A_{pie}(DAC) - A_{DAC}$$

Step 3: Since the intersection is related to the two circles, the angle of the intersection area of the circles (the pie-shaped area) is expressed with the following relation:

$$A_{pie} * 2\pi = \alpha * A_{¿¿}$$

$$\alpha * \frac{A_{¿}}{2\pi} ¿$$

$$\frac{\alpha * \pi r^2}{2\pi}$$

$$\frac{\alpha * r^2}{2}$$

$$A_{pie} = 0.5 * \alpha * r^2$$

$$A_{pie}(DAC) = 0.5 * \widehat{DAC} * r_0^2$$

$$A_{pie}(CBD) = 0.5 * \widehat{CBD} * r_1^2$$

$$CD) = 0.5 * \widehat{CD}$$

Step 4: To determine the angles, we can follow the cosine rule:

$$r_0^2 = r_1^2 + AB^2 - 2 * r_1 * AB * \cos(CBA)$$

Step 5: To determine the distance AB, we can calculate it from the coordinates of A and B:

$$AB = \sqrt{(x_1 - x_0)^2} - \sqrt{(y_1 - y_0)^2}$$

$$\cos(BAC) = ¿ r_0^2 + AB^2 + \frac{r_1^2}{2} * r_0 * AB ¿$$

$$\widehat{BAC} = acos\left(r_0^2 + AB^2 + \frac{r_1^2}{2} * r_0 * AB\right)$$

$$\widehat{ABD} = acos\left(r_1^2 + AB^2 + \frac{r_0^2}{2} * r_1 * AB\right)$$

Step 6: To determine the triangles, since we have the distance between the two triangles and the angles, we can compute the following:

$$A_{DAC} = 0.5 * r_0^2 \sin\left(\widehat{DAC}\right)$$

$$A_{CBD} = 0.5 * r_0^2 \sin\left(\widehat{CBD}\right)$$

Step 7: Finally, we can determine the entire area:

$$Area = \left(A ¿¿ 1 + A_2 + A_3\right) ¿$$

$$A = A_{pie}(DAC) - A_{DAC} + A_{pie}(CBD) - A_{CBD} + A_{pie}(CB) - A_{CB}$$

$$A = 0.5 * \widehat{DAC} * r_0^2 - 0.5 * r_0^2 * \sin\left(\widehat{DAC}\right) + 0.5 * \widehat{CBD} * r_1^2 - 0.5 * r_0^2 * \sin\left(\widehat{CBD}\right) + 0.5 * \widehat{CD}$$

Subsequently, the extraction of the associated API calls with malware apps is enabled, as shown in Figure 3. The creation of the intersection of value sets and frequency exhibited the information of these features that occurred in most of the malicious apps. The value set of the given API calls in malicious apps is greater than the number of appearances in benign apps. To illustrate this point, suppose that a specific API call was requested ten times by the malware set and requested only twice in the benign set. Consequently, we assume the representative API call is associated with malware apps because it appears more frequently in the malicious dataset. Moreover, only 1,321 calls out of 27,147 appear in the risky calls.

We find that compared to the benign dataset, the malware dataset requests frequent API calls to interact with the system. For instance, the API calls related to telephony manager, SMS manager, storage, system service, logs, database, telephony manager and device information occur more frequently in the collected malware apps than in the benign apps. Due to the space limitations, in Table 3, we present a subset of some of the API calls that appeared in this group that are used more frequently in malware apps than in benign apps.

We find the features used by malicious apps request sensitive API calls to access the system. For instance, the ("detDeviceId" and "getSubscriberId") methods can be used to steal sensitive data, such as International Mobile Equipment Identifier (IMEI) and International Mobile Subscriber Identity (IMSI), codes and then later send them over the network using setWifiEnabled or execHttpRequest. The result also shows that the malware apps are influenced by the methods related to sending and receiving messages (*i.e.*, "sendTextMessage", "getDefault" and ".setMessage"). Furthermore, we noticed that the malware dataset influences obfuscation and other hiding techniques to evade static analysis (*e.g.,* Cipher.getInstance).

We assumed that some of the classes in this group might have needed appropriate permissions to protect them against the harmful apps (*i.e.*, "Getdeviceid ", "TelephonyManager", "SmsManage", "SmsMessage", "getSubscriberId"). We found that some of these API calls were already protected by Google permissions, such as "Getdeviceid" "Getsubscriberid", and "Setwifienabled".

Table 3. Ranking Difference of the API Calls

API Calls Name	Meaning
Getdeviceid() Getsubscriberid()	For accessing sensitive data (phone's unique device ID)
Setwifienabled() Exechttprequest()	For communicating over the network
Sendtextmessage() Smsmanager() Setmessage()	For sending and receiving SMS messages

RuntimeException()	For execution of external commands
Cipher.Getinstance()	For obfuscation purposes
Java/Lang/Stringbuffer;.Insert.	For obfuscation purposes
Java/Io/Bytearrayoutputstream;.Reset	For accessing sensitive data
Java/Util/Gregoriancalendar;.Set	For accessing sensitive data (current time)
Android/Telephony/Telephonymanager; .Getnetworkoperator	For accessing sensitive data

- Disruptive group (A1)

We have selected only the API calls that commonly appeared in malicious apps and did not appear in benign apps, as shown in Figure 3. Let $C=\{C1, C2, \ldots Ci\}$ be the set of API calls that appear in the benign apps with their frequency, and let $M=\{M1, M2, \ldots Mj\}$ be the set of API calls that appear in the malware apps with their frequency. To extract the disruptive calls, we compute the following:

$$R = M ¿$$

With respect to the above equation, the features regarding disruptive calls are more clearly oriented towards malicious apps. In contrast to the prior two categories with respect to Figure 3, there is no specific condition for the frequency of API calls because this group is definitively more biased by maliciousness. Since there is no specific frequency condition and there are more propensities in the majority of these cases for malicious API calls, the previously implemented threshold becomes inconsequential, and its role is negligible in this scenario.

The described system was used to generate the API calls that were frequently used only in the malware dataset. The experimental results show some of the API calls (*i.e.*, (Lorg/w3c/dom/DOMException.getMessage), (Android/location/LocationManager;.getLastKnownLocation), (Java/lang/Thread;.setContextClassLoader),(Android/content/Context;.deleteFile), (Android/database/sqlite/SQLiteDatabase;.query),(Java/net/URL;.openConnection), and (Android/telephony/TelephonyManager;.getLine1Number)) are found only in the malware apps and do not appear in the benign apps. Only 5 calls out of 27,147 appear in the disruptive API calls; this result can be explained as follows:

- getLastKnownLocation: it returns the last known location from the given provider and transmits the location information from the device to a remote location; we have noticed that most of the Android.Geinimi family uses this method.

- getLine1Number: it steals private information, such as the phone number, as a string and sends it to a remote server; we have noticed that most of the Android.Fokonge family uses this method.

- setContextClassLoader: it loads the external classes or resources from a repository and can be used for the dynamic loading of malicious codes. Malicious apps could implement the ClassLoader class in order to evade the current countermeasures by replacing the intended code with malicious code. The malicious code is most likely hidden beneath the following path (/assets) or in the secure digital (SD) card itself. We have noticed that most of the Android.Steek family uses this method. During the installation of these malicious applications, only one permission request was made for full Internet access. With one permission request by these malicious applications during installation, it may seem to be less of a risk to potential victims. As soon as it is installed on the smartphone, the malware opens and brings the user to a screen related to the installed fraudulent applications. Malicious apps in the Steek family invoke the setContextClassLoader method the same number of times.

- DOMException: it can be used when events occur. We found that the malicious apps invoked (Lorg/w3c/dom/DOMException) and (Lorg/w3c/dom/DOMException;.<init>.(S Java/lang/String;)V). We found that most of the Android. Steek family uses this method.

- OpenConnection: this method appears in the Android.Generisk family. This family contains open a connection from a predefined remote server and then loads and executes it.

6. STATISTICAL ANALYSIS

In our experiment, we considered all of the features from both groups, "risky calls" and "disruptive calls", as summarized in Table 4.

Table 4. Feature Distribution in the Groups

	Feature set	# of API calls
A3	*Ambiguous Group*	2,368
A2	*Risky Group*	1,321
A1	*Disruptive Group*	5

We calculated the IG of each feature to select important features from among the available ones. The selected features can be used in different kinds of attacks, as reported in Table 5. The main categories denote classifications of types of API calls that can be grouped reasonably. For each group, there are multiple APIs that might be requested. Each feature will be rated as "very important", "important", "normal", or "unimportant" according to the following rules:

While *not at end of list* **do**,

if the feature ∈ Disruptive group,

Then, the feature is very important

elseif the feature ∈ Risky group,

Then, the feature is important

elseif the feature ∈ Ambiguous group,

Then, the feature is normal

else

Then, the feature is unimportant

End

The IG has been applied in each feature, as shown in Figure 4. The score shows the feature importance according to the IG for the top 12 features in the risky group. It confirms that the features selected are very important in mobile malware detection. The algorithm computes splitting criteria for decision trees to discover the contribution of each feature to the given dataset. The IG of each permission is calculated as follows:

$$gain(c, r_i) = entropy(c) - entropy(c \vee r_i)$$

where c is the class value (*i.e.*, either malware or benign) and r_i is the ith feature. Here, entropy (c) is the information entropy. Depending on the classifier, there is an ideal set of features that is less than the overall number of available features. During the initial experiment, we initially use all the features in conjunction with the machine learning algorithms, and then we select the features from the (risky calls and disruptive calls) for the evaluation.

Table 5. Type of Attacks

Attacks	API call groups	
SMS attacks	• Android/telephony/SmsManager;.sendTextMessage • Android/telephony/SmsManager;.getDefault	• Android/telephony/gsm/SmsMessage;.createFromPdu • Android/telephony/gsm/SmsMessage;.getMessageBody

Telephone attacks	• Android/telephony/TelephonyManager;.getNetworkType • Android/telephony/TelephonyManager;.getCallState • Android/telephony/TelephonyManager;.getSimState • Android/telephony/TelephonyManager;.getSimCountryIso • Android/telephony/TelephonyManager;.getSimOperator • Android/telephony/TelephonyManager;.getDataState • Android/telephony/TelephonyManager;.getNeighbouringCellInfo • Android/telephony/TelephonyManager;.getCallState • Android/telephony/TelephonyManager;.getNetworkType • Android/telephony/TelephonyManager;.getSimSerialNumber	• Android/telephony/ TelephonyManager;.getLine1Number • Android/telephony/TelephonyManager;.getSubscriberId • Android/telephony/TelephonyManager;.listen • Android/telephony/ TelephonyManager;.getNetworkOperatorName • Android/telephony/TelephonyManager;.getPhoneType • Android/telephony/TelephonyManager;.getCellLocation
Location attacks	• Android/location/Address;.getLatitude • Android/location/Address;.getLongitude • Android/location/Geocoder;.getFromLocationName • Android/location/Location;.getAccuracy • Android/location/Location;.getLatitude • Android/location/Location;.getLongitude	• Android/telephony/TelephonyManager;.getCellLocation Android/location/LocationManager;.getLastKnownLocation • Android/location/LocationManager;.getProvider • Android/telephony/gsm/GsmCellLocation;.getLac
Camera attacks	• Android/hardware/Camera;.getParameters • Android/hardware/Camera;.open • Android/hardware/Camera;.setOneShotPreviewCallback	• Android/hardware/Camera;.setPreviewCallback • Android/hardware/Camera;.startPreview
Storage attacks	• Android/os/Environment;.getExternalStorageDirectory • Java/io/Externalizable;.readExternal	• Android/content/Context;.getExternalFilesDir • Java/io/Externalizable;.writeExternal
Load extra attacks	• Dalvik/system/DexClassLoader;.loadClass • Java/lang/ClassLoader;.getResource • Java/lang/ClassLoader;.getSystemResources	• Java/lang/Thread;.setContextClassLoader • Java/lang/ClassLoader;.loadClass • Java/net/URLClassLoader;.newInstance.

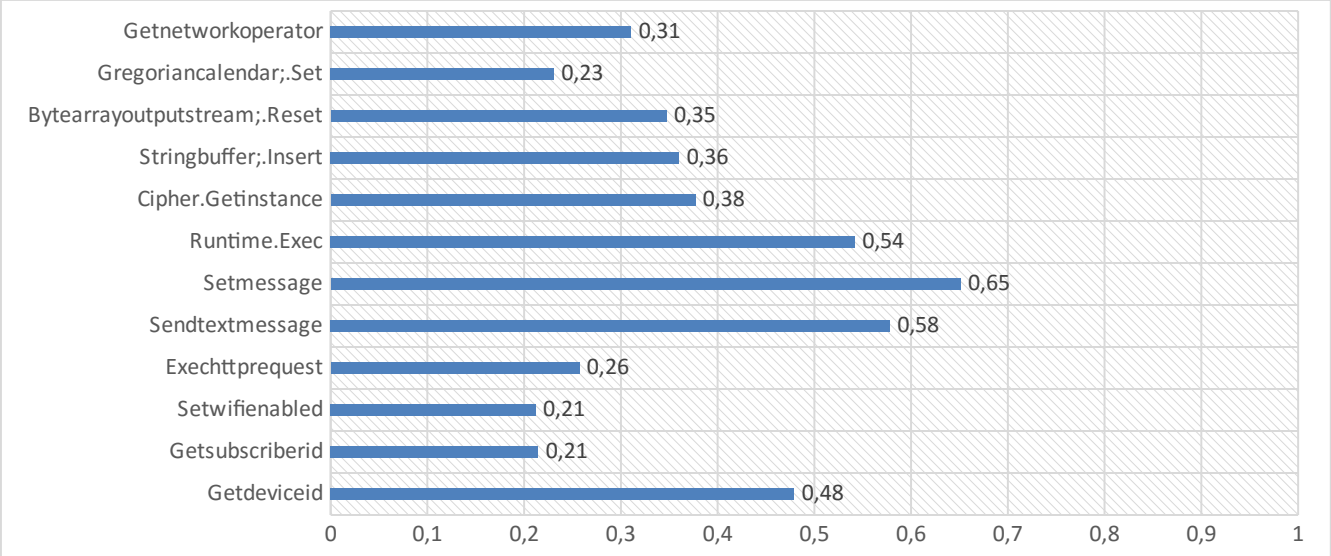

Figure 4. Information gain for the top features in the risky group

7. LEARNING-BASED DETECTION

7.1 Data normalization

Data normalization is an important step before applying machine learning algorithms. Since we have collected the important features (risky group features and disruptive group features), we need to weight and represent those features as a vector space. The term frequency (TF) normalization was implemented

to avoid cases where the classifier has different weights during the decision process. Let w be the extracted dictionary, where the dictionary is from both of the groups (risky groups and disruptive group). The TF representation of the applications is a vector space of weights ($w_{1n}, \ldots w_{¿w \vee n}$), $w_{¿} \in \{0, 1\}$ that represents the occurrence or non-occurrence of a specific feature in an application. This term signifies the frequency of the feature in the application. The step can also be useful to scale the TFs to values between [0-1] by taking the number of times that a feature appears in an application divided by the total number of features in the application, where every application has its own TF. The TF can be computed as follows:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_{k} n_{i,j}}$$

The TF normalization of the dataset allows the vector representation to be observed as a matrix, where rows denote the application vectors and columns are the features. By doing so, various machine learning algorithms can be applied, and we can also find the similarity and differences using the similarity measurement algorithms.

7.2 **Evaluation metrics**

To evaluate the performance of classifiers, we have considered the following: accuracy, precision, recall and F-measure standard metrics. These metrics are estimated based on the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) values.

- TP: represents the number of malware application apps correctly identified as malware applications.
- TN: represents the number of benign application apps correctly identified as benign application apps.
- FP: represents the number of benign application apps misclassified as malware application apps.
- FN: represents the number of malware application apps misclassified as benign application apps.
- The metrics such as accuracy, precision, recall and F-measure are defined as follows:

❖ **Accuracy**: It estimates the ratio of the correctly recognized connection records to the entire test dataset. If the accuracy is higher, the machine learning model is better. The accuracy serves as a good measure for the test dataset that contains balanced classes and is defined as follows:

$$Accuracy = (TP + TN) / (TP + TN + FP + FN)$$

❖ **Precision**: The percentage of correctly identified can be computed as follows:

$$Precision = TP / (TP + FP)$$

❖ **Recall**: The percentage correctly identified as malicious can be computed as follows:

$$Recall = TP / (TP + FN)$$

❖ **F-measure**: The combination of precision and recall can be computed as follows:

$$F.Measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

7.3 **Results & analysis**

Our main aim is to examine if the selected features from the risky group and disruptive calls could be used to build complex classification models to predict the classes or risk factors of mobile malware. All features were obtained from both the risky or disruptive groups (1,326 distinct features) remaining after neglecting those unimportant features in the dataset. For each group (risky group and disruptive groups), five machine learning algorithms were executed through 10-fold cross-validation: random forest (RF), J48, random tree (RT), k-nearest neighbours (k-NN) and naïve Bayes (NB).

The empirical results suggest that our proposed method is effective at detecting mobile malware and achieved an F-measure of 94.3%, as shown in Figure 5. Our model can significantly assist in the process of malware forensic investigation and mobile application analysis.

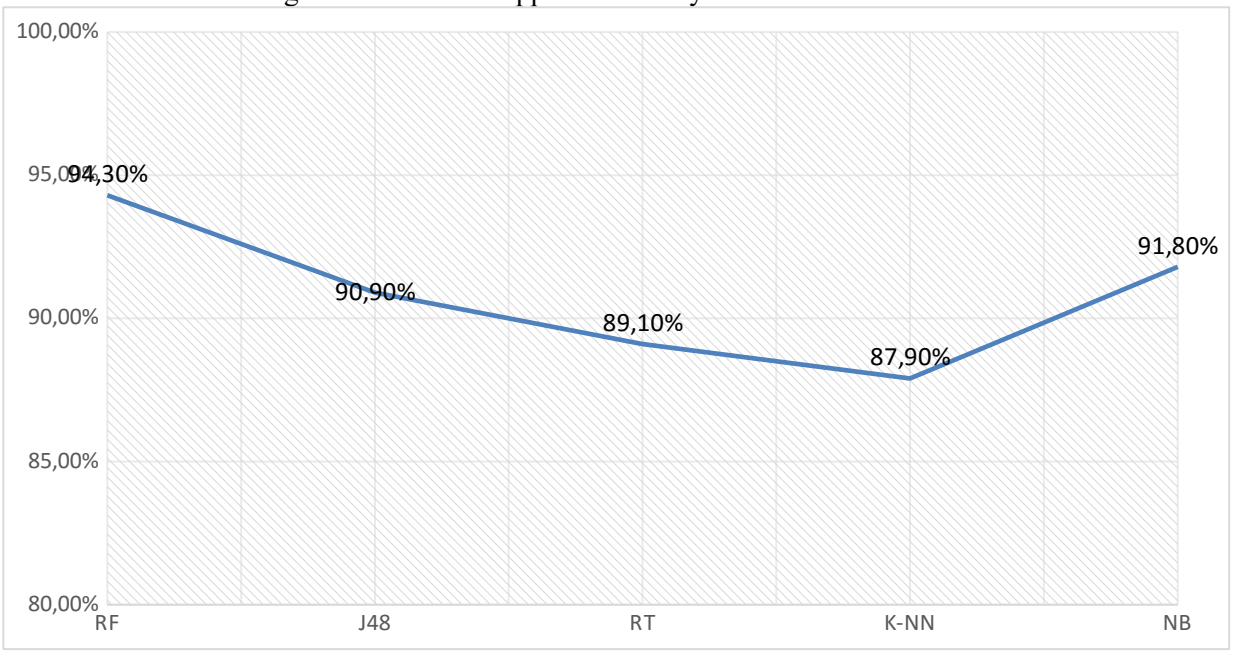

Figure 5. F-measure scores

In terms of the speed of training and testing of each classifier, the fastest algorithms are the random tree and k-NN; they both take 0.2 seconds. J48 requires the longest time, 0.92 seconds. Random forest takes 0.73 seconds for training and testing. Overall, the system is predictable and reliable, and the speed is acceptable for all 5 classifiers in real-time applications.

8. CONCLUSIONS

Identifying the most prominent features requested by malware apps is the key factor in building a safe mobile computing environment, protecting sensitive data and detecting malware. Permissions and critical API calls reflect the behaviour patterns of an Android application. Motivated by the increasing number of apps and the lack of effective malware detection tools, in this paper, we propose a reliable classification model that combines the consideration of permissions and API calls to detect malware apps. Our analysis method consists of three main phases: the pre-processing phase, extraction phase, and grouping phase. Because Android apps use a large number of APIs, we used a grouping strategy for choosing only the most valuable API calls to maximize the likelihood of identifying the Android malware apps as follows:

- Ambiguous group (common API calls in both malware apps and benign apps)
- Risky group (common API calls in malware apps that are less similar to those in benign apps)
- Disruptive group (API calls that appeared in malware apps and did not appear in benign apps)

The frequency analysis is performed on the important groups to find the most discriminating set of features for malware detection. The results demonstrate that malicious applications invoke a different set of API calls with different frequencies compared with the benign apps and that mobile malware applications request dangerous permissions more often than benign apps to access sensitive user data. For instance, the API calls for the telephony manager, SMS manager, storage, system service, logs, database, telephony manager and device information appear significantly more often in malware apps. Empirical results with a real malware dataset of 27,891 Android apps suggest that our proposed method is effective at detecting mobile malware and can significantly assist in the process of malware forensic investigation and mobile application analysis. The IG and API calling frequencies are calculated to select a valuable

subset of features, and then the TF is used for dimensionality reduction of the selected features. We apply five different machine learning techniques in our approach, namely, the RF, J48, RT, k-NN and NB algorithms. The experimental results demonstrate that our model achieves an F-measure of 94.3%.

9. REFERENCES:

Aafer, Y., Du, W., & Yin, H. 2013 'DroidAPIMiner: Mining API-level features for robust malware detection in android', In Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Revised Selected Papers (Vol. 127 LNICST, pp. 86-103). (Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST; Vol. 127 LNICST). Springer Verlag.

Alazab M, Venkatraman S, Watters P, Alazab M, Alazab A 2012, 'Cybercrime: The Case of Obfuscated Malware'. In: Georgiadis C.K., Jahankhani H., Pimenidis E., Bashroush R., Al-Nemrat A. (eds) Global Security, Safety and Sustainability & e-Democracy. e-Democracy 2011, ICGS3 2011. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 99. Springer, Berlin, Heidelberg

Alazab, M & Batten, L 2015, 'Survey in Smartphone Malware Analysis Techniques', in New Threats and Countermeasures in Digital Crime and Cyber Terrorism, IGI Global, pp. 105-30.

Alazab, M & Tang, MJ 2019, 'Deep Learning Applications for Cyber Security', Springer Nature Switzerland AG.

Alazab, M 2015, Profiling and classifying the behaviour of malicious codes', Journal of Systems and Software, vol. 100, pp.91-102.

Alazab, M, Monsamy, V, Batten, L, Lantz, P & Tian, R 2012, 'Analysis of Malicious and Benign Android Applications', in International Conference on Distributed Computing Systems Workshops (ICDCSW), 2012 32nd, pp. 608-16.

Alazab, M, Venkatraman, S, Watters, P & Alazab, M 2011, 'Zero-day malware detection based on supervised learning algorithms of api call signatures', in Proceedings of the Ninth Australasian Data Mining Conference-Volume 121, pp. 171-82.

Alazab, M. (2014). Analysis on Smartphone Devices for Detection and Prevention of Malware. Faculty of Science, Engineering and Built Environment. Deakin University, Deakin University. Doctor of Philosophy: 285.

Allix, K, Bissyandé, TF, Jérome, Q, Klein, J & Le Traon, Y 2016a, 'Empirical assessment of machine learning-based malware detectors for Android', Empirical Software Engineering, vol. 21, no. 1, pp. 183-211.

Allix, K, Bissyandé, TF, Klein, J & Le Traon, Y 2016b, 'Androzoo: Collecting millions of android apps for the research community', in 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 468-71.

Arshad, S, Shah, MA, Khan, A & Ahmed, M 2016, 'Android malware detection & protection: a survey', International Journal of Advanced Computer Science and Applications, vol. 7, no. 2, pp. 463-75.

Ahvanooey, M, Li, Q, Zhu, X, Alazab, M, Zhang, J 2020, ‘ANiTW: A Novel Intelligent Text Watermarking technique for forensic identification of spurious information on social media’, Computers & Security, vol 90.

Badhani, S & Muttoo, SK 2018, 'Evading android anti-malware by hiding malicious application inside images', International Journal of System Assurance Engineering and Management, vol. 9, no. 2, pp. 482-93.

Bankmycell 2019, How many phones are in the world?, retrieved JUN. 25 2019, <https://www.bankmycell.com/blog/how-many-phones-are-in-the-world>.

Brandom, R 2019, There are now 2.5 billion active Android devices, theverge, retrieved Sep. 04 2019, <https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote>.

Brodeur, P 2012, Zero-Permission Android Applications Part 2, leviathansecurity, retrieved Jun. 22 2013, <http://www.leviathansecurity.com/blog/zero-permission-android-applications-part-2/>.

Burguera, I, Zurutuza, U & Tehrani, SN 2011, 'Crowdroid: behaviour-based malware detection system for Android', in Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, Chicago, Illinois, USA, pp. 15-26.

Caviglione, L, Gaggero, M, Lalande, J-F, Mazurczyk, W & Urbański, M 2015, 'Seeing the unseen: revealing mobile malware hidden communications via energy consumption and artificial intelligence', IEEE Transactions on Information Forensics and Security, vol. 11, no. 4, pp. 799-810.

Clement, J 2019, Number of apps available in leading app stores as of 2nd quarter 2019, statista, retrieved Nov. 04 2019, <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>.

Desnos, A 2019, androguard Reverse engineering, Malware and goodware analysis of Android applications, Code.google, retrieved Sep. 5 2019, <http://code.google.com/p/androguard/>.

Dimjašević, M, Atzeni, S, Ugrina, I & Rakamaric, Z 2016, 'Evaluation of android malware detection based on system calls', in Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pp. 1-8.

Farivar, F, Haghighi, M, Jolfaei, A, Alazab, M, 2019,’Artificial Intelligence for Detection, Estimation, and Compensation of Malicious Attacks in Nonlinear Cyber Physical Systems and Industrial IoT’, *IEEE Transactions on Industrial Informatics*.

Fereidooni, H, Moonsamy, V, Conti, M & Batina, L 2016, 'Efficient classification of android malware in the wild using robust static features', Protecting Mobile Networks and Devices: Challenges and Solutions, vol. 1, pp. 181-209.

Gao, H, Guo, C, Wu, Y, Dong, N, Hou, X, Xu, S & Xu, J 2019, 'AutoPer: Automatic Recommender for Runtime-Permission in Android Applications', paper presented to 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC).

HaddadPajouh, H, Dehghantanha, A, Khayami, R & Choo, K-KR 2018, 'A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting', Future Generation Computer Systems, vol. 85, pp. 88-96.

Irshad, M, Al-Khateeb, HM, Mansour, A, Ashawa, M & Hamisu, M 2018, 'Effective methods to detect metamorphic malware: a systematic review', IJESDF, vol. 10, no. 2, pp. 138-54.

Jung, J, Kim, H, Shin, D, Lee, M, Lee, H, Cho, S-j & Suh, K 2018, 'Android malware detection based on useful API calls and machine learning', in 2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), pp. 175-8.

Juniper 2012, Mobile-threats-report, Mobile-threats-report, retrieved Oct. 20 2019, <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwiqkobx7rLlAhWPJlAKHf4ZDrsQFjAAegQIARAC&url=https%3A%2F%2Fwww.juniper.net%2Fus%2Fen%2Flocal%2Fpdf%2Fadditional-resources%2Fjnpr-2011-mobile-threats-report.pdf&usg=AOvVaw1634BfbxoY780LimKdJ_1V>.

Kanda, T, Manabe, Y, Ishio, T, Matsushita, M & Inoue, K 2011, 'A Prototype of Comparison Tool for Android Applications Based on Difference of API Calling Sequences', IEICE vol. 111, no. 107, pp. 35-40.

Kapratwar, A, Di Troia, F & Stamp, M 2017, 'Static and dynamic analysis of android malware', in ICISSP, pp. 653-62.

Karbab, EB, Debbabi, M, Derhab, A & Mouheb, D 2018, 'MalDozer: Automatic framework for android malware detection using deep learning', Digital Investigation, vol. 24, pp. S48-S59.

Kaspersky 2019, Financial Cyberthreats in 2018, securelist, retrieved Sep. 04 2019, <https://securelist.com/financial-cyberthreats-in-2018/89788/>.

Kaushik, P & Jain, A 2015, 'Malware Detection Techniques in Android', International Journal of Computer Applications, vol. 122, no. 17.

Kayes, A, Rahayu, W, Dillon, T 2019a ‘Critical situation management utilizing IoT-based data resources through dynamic contextual role modeling and activation’, Computing 101(7), pp. 743-772

Kayes, A, Rahayu, W, Dillon, T, Chang, E, Han, J 2019b ‘Context-aware access control with imprecise context characterization for cloud-based data resources’, Future Generation Comp. Syst. 93, pp. 237-255

Kayes, A, Han, J, Rahayu, W, Dillon, T, Islam, M, Colman, A 2019c ‘A Policy Model and Framework for Context-Aware Access Control to Information Resources’ Comput. J. 62(5), pp. 670-705

Labs, A 2014, Analysis Report on Android Trojan HongTouTou (ADRD), retrieved Feb. 11 2014, <http://www.antiy.net/media/reports/android_adrd_analysis.pdf>.

Li, J, Sun, L, Yan, Q, Li, Z, Srisa-an, W & Ye, H 2018, 'Significant permission identification for machine-learning-based android malware detection', IEEE Transactions on Industrial Informatics, vol. 14, no. 7, pp. 3216-25.

Linares-Vásquez, M, Bavota, G, Bernal-Cárdenas, C, Penta, M, Oliveto, R & Poshyvanyk, D 2013. API change and fault proneness: a threat to the success of Android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, 477-487.

Lyvas, C, Lambrinoudakis, C & Geneiatakis, D 2018, 'Dypermin: dynamic permission mining framework for android platform', Computers & Security, vol. 77, pp. 472-87.

Mcafee 2019, Mobile Threat Repor, McAfee Mobile Threat Report Q1, 2018, retrieved JUN. 20 2019, <https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf>.

Moonsamy, V & Batten, L 2012, 'Zero permission android applications - attacks and defenses', paper presented to ATIS 2012 : Proceedings of the 3rd Applications and Technologies in Information Security Workshop, Melbourne, Vic.

Moonsamy., V, Alazab., M & Batten., L 2012, 'Towards an Understanding of the Impact of Advertising on Data Leaks', International Journal of Security and Networks (IJSN), vol. 7 no. 3.

Naway, A & Li, Y 2019, 'Android Malware Detection Using Autoencoder', arXiv preprint arXiv:1901.07315.

Oh, T, Stackpole, B, Cummins, E, Gonzalez, C, Ramachandran, R & Lim, S 2012, 'Best security practices for android, blackberry, and iOS', in 2012 The First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT), pp. 42-7.

P. Faruki et al. 2015, ‘Android Security: A Survey of Issues, Malware Penetration, and Defenses’, in IEEE Communications Surveys & Tutorials, vol. 17, no. 2, pp. 998-1022, 2015.

Poeplau, S, Fratantonio, Y, Bianchi, A, Kruegel, C & Vigna, G 2014, 'Execute this! analyzing unsafe and malicious dynamic code loading in android applications', in Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS).

Qiao, M., Sung, A. H., & Liu, Q. (2016). Merging permission and API features for Android malware detection. In 2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI) (pp. 566-571).

Quarta, D, Salvioni, F, Continella, A & Zanero, S 2018, 'Toward Systematically Exploring Anti-malware Engines', in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 393-403.

Rastogi, V, Chen, Y & Jiang, X 2013, 'DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks', in 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013) Hangzhou, China.

Shabtai, A, Kanonov, U, Elovici, Y, Glezer, C & Weiss, Y 2011, '"Andromaly": a behavioural malware detection framework for android devices', Journal of Intelligent Information Systems, pp. 1-30.

Shao, Y, Chen, QA, Mao, ZM, Ott, J & Qian, Z 2016, 'Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework', paper presented to NDSS, San Diego, CA, USA Copyright 2016 Internet Society, ISBN 1-891562-41-Xhttp://dx.doi.org/10.14722/ndss.2016.23046.

Sophos 2019, When Malware Goes Mobile, security-news-trends, retrieved JUN. 20 2019, <https://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile.aspx>.

Thomas, K, Huang, D, Wang, D, Bursztein, E, Grier, C, Holt, TJ, Kruegel, C, McCoy, D, Savage, S & Vigna, G 2015, 'Framing dependencies introduced by underground commoditization'.

Venkatraman, S & Alazab, M 2018, 'Use of Data Visualisation for Zero-Day Malware Detection', Security and Communication Networks, 2018.

Visu, P, Lakshmanan, L, Murugananthan, V & Cruz, MV 2019, 'Software-defined forensic framework for malware disaster management in Internet of Thing devices for extreme surveillance', Computer Communications.

Watters, P, Ziegler, J 2016 'Controlling information behaviour: the case for access control'. Behaviour & IT 35(4), pp. 268-276

Watters, P, Scolyer-Gray, P, Kayes, A, Chowdhury, M (2019) 'This would work perfectly if it weren't for all the humans: Two factor authentication in late modern societies'. First Monday 24 (7).

Xie, N, Wang, X, Wang, W & Liu, J 2019, 'Fingerprinting Android malware families', Frontiers of Computer Science, vol. 13, no. 3, pp. 637-46.

Yang, C, Xu, Z, Gu, G, Yegneswaran, V & Porras, P 2014, 'Droidminer: Automated mining and characterization of fine-grained malicious behaviours in android applications', in European symposium on research in computer security, pp. 163-82.

Yerima, S & Khan, S 2019, 'Longitudinal performance analysis of machine learning based Android malware detectors', paper presented to International Conference on Cyber Security and Protection of Digital Services (Cyber Security 2019), Oxford, United Kingdom, June 3-4, 2019

Zhang, H, Luo, S, Zhang, Y & Pan, L 2019, 'An efficient Android malware detection system based on method-level behavioural semantic analysis', IEEE Access, vol. 7, pp. 69246 - 56.

Zhou, Y & Jiang, X 2012, 'Dissecting Android Malware: Characterization and Evolution', paper presented to Security and Privacy (SP) IEEE Symposium on, 20-23 May 2012.