Review Paper

The Android malware detection systems between hope and reality

Khaled Bakour[1] · Halil Murat Ünver[1] · Razan Ghanem[1]

Abstract
The widespread use of Android-based smartphones made it an important target for malicious applications' developers. So, a large number of frameworks have been proposed to tackle the huge number of daily published malwares. Despite there are many review papers that have been conducted in order to shed light on the works that achieved in Android malware analysing domain, the number of conducted review papers do not fit with the importance of this research field and with the volume of achieved works. Also, there is no comprehensive taxonomy for all research trends in the field of analysing malicious applications targeting the Android system. Furthermore, none of the existing review papers contains a schematic model that makes it easy for the reader to know the methods and methodologies used in a particular field of research without much effort. This paper aims at proposing a comprehensive taxonomy and suggesting a new schematic review approach. To this end, a review of a large number of works that achieved between 2009 and 2019 has been conducted. The achieved study includes more than 200 papers that have different goals such as apps' behaviour analysis, automatic user interface triggers or packer/unpacker frameworks development. Also, a comprehensive taxonomy has been proposed so that most of the previous works can be classified under it. To the best of our knowledge, the suggested taxonomy is the widest and the most comprehensive in terms of the covered research trends. Moreover, we have proposed a detailed schematic model (called Schematic Review Model) illustrates the process of detecting the malignant applications of an Android in the light of the studied works and the proposed taxonomy. To our knowledge, this is the first time that the Android malware detection methods have been explained in this way with this amount of detail. Furthermore, the studied researches have been analysed according to multiple criteria such as used analysing method, used features, used detection method, and used dataset. Also, the features used in the studied works were discussed in detail by dividing it into multiple classes. Moreover, the challenges facing Android's malware analysing methods were discussed in detail. Finally, it has been concluded that there are gaps between the size and the goal of the conducted works and the number of malicious apps published every day, so some future works areas have been proposed and discussed.

Keywords Android malware · Static analysis · Dynamic analysis · Malware visualisation · Obfuscation · Malware detection

1 Introduction

Smartphones have become one of the most important devices that currently relied on to accomplish many important activities in our daily lives. Therefore, to keep abreast of the growth and rapid development of smartphone technology, many advanced applications have been developed and presented in both of the official and third-party app markets. Consequently, smartphones demand has been increased dramatically over time. According to Gartner's

Khaled Bakour (Halit Bakir): The author has a dual citizenship, so his name is written in two different ways.

✉ Khaled Bakour, khaledbakour@kku.edu.tr; Halil Murat Ünver, unver@kku.edu.tr; Razan Ghanem, razan@kku.edu.tr | [1]Department of Computer Engineering, Kırıkkale University, Kirikkale, Turkey.

Received: 19 April 2019 / Accepted: 21 August 2019 / Published online: 29 August 2019

2017 report about the worldwide smartphones' sales, a global sale of smartphones reached 366.2 million units in the second quarter of 2017 with 6.7% increase over the same period in 2016 [1]. Furthermore, it was stated that Google's Android extended its lead by capturing 86% of the total market in 2017 with 1.1% increase over 2016 [2].

The Android OS is counted as the most popular Mobile OS because that it is a free and open source OS. In addition, Android has facilitated downloading its applications from each of the official and third-party app markets. The Android official market (Google Play) was originally launched in October 2008 under the name Android Market, and according to the statistics portal of Statista website, the number of available apps in the Google Play app store reached to 3.3 million apps in March 2018, after surpassing 1 million apps in July 2013 [3]. Also, it is stated that the fourth quarter of 2017 representing an 8.84% growth in number of apps compared to the third quarter [4]. Moreover, they (Statista researchers) said that the number of apps that downloaded from the Google Play app store between August 2010 and May 2016 reached 65 billion apps [5].

On the other hand, the widespread of Android and its open-source nature have made it a major target for malicious software developers. According to the Pulse Secure Mobile Threat Report in 2015, nearly one million unique malicious applications that target Android OS were launched in 2014 with a 391% increase over 2013 [6]. And they stated that Android is ranked as the first smartphone OS in terms of the number of malicious programs, where the number of malwares that targeting Android reached 97% of all mobile devices' malware. Also, according to Symantec's 2016 internet security threat report [7], the number of Android malware families added in 2015 grew by 6% compared with the 20% growth in 2014. Moreover, it was indicated that the Android malware start to use obfuscation techniques to bypass static analysis-based frameworks and it can bypass the most of dynamic analysis tools by checking if it is running on real phones or any kind of emulators or sandboxes that used by security analyser. Additionally, according to the G DATA Security blog report in 2017 [8], 750,000 malicious programs that target Android have been discovered in the first quarter of 2017 only and in total more than 3 million new malware samples targeting the Android operating system were discovered in 2017. Furthermore, McAfee's Mobile Research team has found a new Android malware in 144 "Trojanized" applications on Google Play, this threat has been named as Grabos [9]. Grabos was initially found in Android application called "Aristotle Music audio player 2017" which claimed to be a free audio player. McAfee Mobile Research notified Google about Grabos in September 2017 and confirmed that Google promptly removed the reported application. After further research, they found another 143 applications, before they have been removed from Google Play.

Due to the importance of the Android OS and increasing its security threats the Android malware detection field has become one of the most important academic research areas. So, a large number of frameworks has been proposed and developed since 2009 until these days. In this paper, a comprehensive review has been conducted for more than 200 papers that published between 2009 and the beginning 2019 in order to shed light on the security reality of the Android operating system and the research trends that should be focused in future works.

1.1 Related reviews

A good number of reviews have been conducted previously to highlight the achieved works in Android security domain. In this section, the most important of these reviews and their weaknesses will be discussed.

In [10], Android's security mechanisms and its issues, as well as the malware analysis evasion techniques, have been discussed. Also, the general malware analysing methods, the most important used tools and some of the previously conducted state of art frameworks have been studied. In the end, a hybrid framework's schematic model has been proposed to be applied in future work. This study does not contain any taxonomy for the previous related works. Also, there are no discussion or taxonomy for the features that can be used in the Android malware analysing domain. In [11], the Android OS weaknesses have been listed, a taxonomy for the previous works has been proposed and some of future works directions have been discussed. The proposed taxonomy is overly detailed and contains overlapped information in some sections. Furthermore, some of research trends such as user interface triggering tools and image-based malware detection frameworks have not been included on the proposed taxonomy. Also, there is no mention to evasion techniques that used by the malicious code developers in order to avoid the malware detection systems. Moreover, there is no discussion or taxonomy for the features that can be used in each of Android malware analysis methods. In [12], the techniques that can be used in Android malware analysing systems have been discussed. Also, the techniques which used by the malware developers for avoiding the detection methods have been explained. In this paper, there is no clear taxonomy for the works that conducted in Android malware analysing domain. In [13], a detailed background about the Android operating system was introduced. Also, the Android's security mechanisms and its threats have been discussed. In this survey, a very simple taxonomy for previously proposed frameworks has been introduced without any mention to the features that used in the different analysing methods. In [14], a taxonomy for Android malware detection frameworks has been proposed and a systematic analysis of more than 300 papers have been conducted. Also, some gaps in the proposed approaches have been discussed and some future works trends were proposed. Although the proposed taxonomy is overly detailed it is not comprehensive for all research directions in the Android security domain such as Policy enforcement frameworks, user interface triggering tools and packer/unpacker tools. Furthermore, the survey does not discuss a taxonomy for the features that can be used on each type of analysing approaches. In [15], 100 papers have been studied in term of the used features and features' selection methods only, and the used features have been classified into multiple classes. In [16], a systematic review for 124 static analysis works that published between 2011 and 2015 has been conducted. The paper contains detailed study for static analysis techniques and some of its challenges, but it does not contain enough discussion for the features that used in this type of analysis methods and there is no clear taxonomy for the related works. Also, in [17], more than 80 static analysis works have been studied and a well taxonomy has been proposed for the features that used in the studied works. Furthermore, the challenges of static analysis method were discussed, and a case study was conducted to test the robustness of some anti-malware systems against some obfuscation techniques.

It is worth mentioning that the malware visualisation-based malware analysis research trend has not been discussed in any of the previously conducted review papers.

1.2 Motivations and goal

Although there are many review papers have been done in the Android security field, there is no comprehensive review for all this research area's aspects. Particularly, there is no review paper that contains a comprehensive taxonomy for all research directions so that it can help the researchers to know the research trends in the domain. Also, none of the previous reviews contains a comprehensive schematic description showing all the used techniques on a clear phases' form so that any researcher who has no experience in the field can take a comprehensive and concise overview about it. We believe that any review on any particular research area (not only in the field of android security) should contain a clear schematic model that serves as a guide for researchers to understand the research problem and the mechanisms to address it. Furthermore, there are some important work trends in the Android malware analysis domain have not been discussed in any of previous conducted reviews. For example, the works that aim at converting the malicious apps into images (whether grayscale or RGB) and analysing them using image processing techniques (a.k.a malware visualisation-based analysis frameworks) have not been discussed in any of previous reviews. So, we have set the following objectives for this paper:

1. Conducting a review study that covers as many as possible of researches that have been conducted in the Android malware analysing domain between 2009 and the beginning of 2019.
2. Proposing a comprehensive taxonomy so that includes as most as possible research trends in Android security domain.
3. Proposing a detailed taxonomy for the features that used in the malware analysis methods.
4. Concluding a detailed schematic description model (in the studied works' light) that allows a well understanding of the used techniques in this domain without much effort.
5. Evaluating the covered works and studying its weaknesses in order to figure out the existing research gaps and proposing some available future research areas.

The following research questions have been drawn in the paper's objectives light:

RQ1 What are the most important techniques and approaches that used in the field of Android malware detection, and what are the most important research trends of the previous works?

RQ2 What is the most appropriate way to classify the studied works within a comprehensive taxonomy that includes most of the previous works in the domain?

RQ3 Is it possible to express the techniques and methods that used in Android malware detection frameworks using a comprehensive schematic model inspired from the studied works?

RQ4 What are the weaknesses of the studied frameworks, the existing research gaps and the trends that should be covered the future work?

The main contributions of this paper are:

- More than 200 research papers that were published in a period extending from 2009 until the beginning of 2019 have been studied from multiple aspects.
- A comprehensive taxonomy for Android malware analysing works has been proposed so that it can cover most of the research in this research field.

- The studied researches have been analysed according to multiple criteria, such as a used analysing method, used features, used detection method, used dataset, etc.
- The process of detecting Android malware was explained under four phases and using a novel detailed schematic manner. To the best of our knowledge, this is the first time that the Android malware detection techniques have been explained in this way with this amount of detail. We called this proposed method a Schematic Review Model.
- The features that used in the studied works were discussed in detail by dividing it into multiple classes.
- Furthermore, the challenges that facing Android malware analysing methods were discussed in detail.
- Finally, some future works areas have been proposed and discussed.

2 Android background

Android is a free open source operating system based on the Linux kernel developed by the Android Open Source Project AOSP managed by Google. Google purchased Android system from the main developers in 2005, while the Android's official announcement was in 2007, and the first Android device appeared on the market in 2008.

2.1 Android application

Generally, the Android application is written using Java programming language and some native code can be added to it. Then, the application is compiled to Java bytecode which translated into Dalvik bytecode and stored in.dex (Dalvik EXecutable) and.odex (Optimized Dalvik EXecutable) files. In the end, the app is compiled to an APK archive which contains the application code (.dex files), resources, assets, and manifest file. There are four types of components can be defined in Android app, i.e. activities, services, broadcast receivers and content providers. The activity is the part that provides the user interface in the Android application. The service is a general-purpose entry point to keep the app running in the background for all kinds of reasons. The broadcast receiver is a well-defined entry point allow the app to receive a specific event from the operating system or another application. The content provider acts as a database management system manages shared data between apps.

2.2 APK archive architecture

The Android APK archive contains the following files and folders.

2.2.1 AndroidManifest.xml

It is one of the most important files in the Android application, and this file is the first part that read by the OS when running any application.

2.2.2 Classes.dex

Dex code is an optimized bytecode for Android applications that contains multiple constructs like file header, string table, local variable list, class definition table, method list…etc.

2.2.3 Resources.arsc

It is a file containing the application's resources in a binary format.

2.2.4 Lib/folder

This folder contains the native code libraries.

2.2.5 Assets/folder

The assets (i.e. images, files, etc.) can be placed in this folder and it will be accessed using AssetManager.

2.2.6 Res/folder

The app's resources (like icons, music, images etc.) are placed in this directory.

2.2.7 META-INF/folder

It contains the application's certificate and composed of three essential files namely, MANIFEST.MF, *.SF, and *.RSA.

2.3 Android's security mechanisms

The Android system is based on three protection mechanisms namely, permission framework, sandboxing and application signing.

2.3.1 Permission framework

It is designed by Google as a protection mechanism for system resources so that the program cannot access a certain protected part of the system or another application unless it has the right permissions. The granted permissions are assigned to application's sandbox and it will be inherited by all application's components. If an application wants to use an API to access a specific system resource, appropriate permission must be declared in its AndroidManifest.xml file.

2.3.2 Sandboxing

It is a technique used to isolate applications from each other and preventing the arrival of any application to the other applications' resources unless it has a specific permission. In other words, each Android application is executed within its virtual machine (VM) instance so that each of these instances is executed under a unique user ID to isolate each application from the other apps. The applications can only access other application's resources by using the IPC (Inter-Process Communication) binder mechanism. This mechanism can be bypassed by an attack called Privileges' escalation which illustrated in Fig. 1. Assuming that we have three applications A, B and C. The application A wants to access a component C_1 in the application C, but this component requires a permission P_1. Since the application A does not have the P_1 permission, it does not have a direct access to the C_1 component. But the app A can access app B which does not require any permissions and it has the permission P_1, thus the app A can access the required component (i.e. C_1 component) through the app B. Thus, application A can access to C_1 indirectly.

2.3.3 Application signing

The developers should sign the applications using their own self-keys. Therefore, this technology does not provide a protection mechanism against malicious applications as much as generating confidence in applications that developed by the same entity. It should be noted that the applications which signed with the same key can work within the same SandBox.

Previously, Google had been using a framework called Bouncer, which dynamically was analysing applications by executing the app in an isolated environment and examining their behaviour. Google has recently introduced a new framework called Play Protect, which is an always-on tool that scans the applications even after its installation by the end-user. In addition, Play Protect can scan even the applications that are downloaded from the third-party markets. Furthermore, it is stated that this framework can scanning and verifying over 50 billion apps every day [18].

3 Research methodology

In this section, we will discuss searching criteria that used to select the studied papers and the protocol that used to include/exclude the papers from the conducted study.

3.1 Search criteria

We initially identified the search terms that used to search famous academic search engines such as Google scholar, Springer, ScienceDirect and IEEE Xplore, ACM Digital Library, etc. We used keywords related to analysis methods such as malware analysis, static analysis, hybrid analysis, malware detection, and malware visualisation. Also, we used some terms related to smartphone and Android such as Android malware, mobile malware and smartphone attacks. Furthermore, we used general terms such as behaviour analysis, anomaly detection, signature-based detection and apps classification.

3.2 Papers selection criteria

Firstly, since 2008 is considered as the actual beginning date of the Android operating system, we excluded the papers that were published before 2008. After that, we excluded the papers which are not related to malware

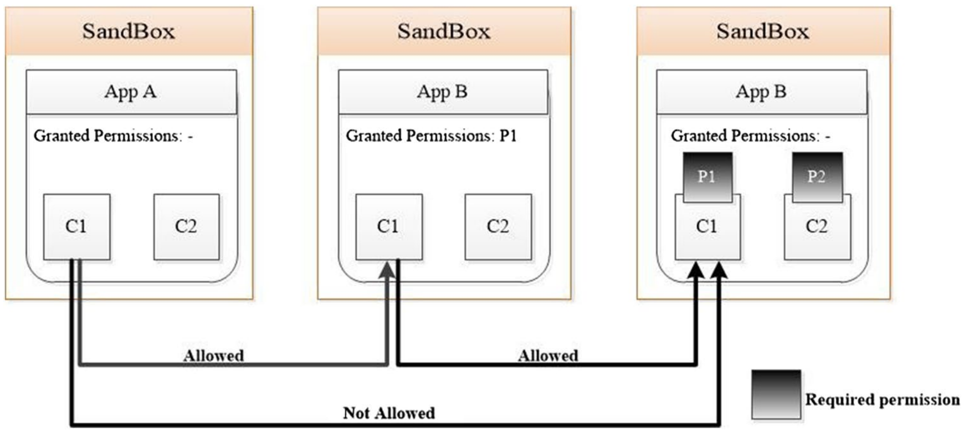

Fig. 1 Privilege escalation attack scenario

analysing domain. Then, the papers' abstracts were examined to exclude the papers related to personal computers' malware or other smartphone operating systems' malware such as IOS's malware. In the end, the following types of works are generally included in our survey:

1. The papers that aim at developing a malware behaviour analysing frameworks using any of the malware analysing methods (static, dynamic or hybrid analysis).
2. Papers that aim at developing apps' classification frameworks whether using signature-based, machine learning-based, etc.
3. Papers that aim to develop user interface events' generating tools (UI triggers).
4. The works that aim at proposing techniques for avoiding malware detection tools (code packer tools).
5. The works that aim at proposing tools that can be used to retrieve the original code from the obfuscated one (code unpacking tools).
6. The works that aim at proposing policy enforcement frameworks that can be applied whether at app installation or execution time.

On the other hand, we excluded the papers that are not related to the previous six trends. We will represent some examples of excluded works in the following paragraph.

For example, Xie et al. [19] aims at developing a malware behaviour detection framework for mobile devices generally and since we aim to study the works that aim at developing android's malware analysing frameworks, therefore this paper has been excluded. Also, Vidas et al. [20] and Bartel et al. [21] aims in developing tools that help the developers in specifying a minimum set of permissions required for a specific Android app. Since these two papers are out of our review's scope so it was excluded. In [22], a malware that able to retrieve the user credentials from the apps' memory has been developed. This type of papers is out of our review's scope, so it has been excluded. We also excluded [23] which aims to analyse the potential privacy and security risks of in-app advertisements. Furthermore, we excluded some works that do not have a direct android malware analysing frameworks development goal, for example, in [24] an assessment of the presence of malware in third-party Android markets using well-known anti-virus engines was presented. To this end, a dataset was collected from nine different third-party markets in three geographical regions (China, Europe, Russia) and multiple statistical analyses were performed on the collected dataset. Also, in [25], multiple studies including internet survey have been conducted in order to test the effectiveness of the Android permissions mechanism.

4 Proposed taxonomy

This paper aims to propose a comprehensive taxonomy such that include as many Android security research trends as possible. Our taxonomy is based on the papers' goals and the problems that are tried to solve within the works. We will explain the proposed taxonomy in detail, and in the same context, a taxonomy for the features that used in the various analysing methods will be proposed. Also, a classification for the weaknesses of the used analysing methods has been presented and discussed in detail. Figure 2 illustrates the proposed taxonomy. We will divide the studied works into five main trends each of which's details will be discussed.

4.1 Behavior analysis framewoks

This research trend covers all works that aim at proposing frameworks for analyzing and classifying the behaviour of malicious applications. We classified these works according to two criteria. The first one is the used technical methods, which were divided into four main phases each of which contains different methodologies and techniques that have been used in the studied works. The second criterion is the challenges facing the used analysing method which have been attempted to address within the specific studied work.

4.1.1 Used techniques phases based taxonomy

We have divided the Android malware detection process in the light of the studied works into four phases, namely, pre-processing phase, features extraction phase, features selection phase, and the detection phase. We will classify the studied works based on the techniques used in these phases. In the following sections, the details of each of these phases will be discussed.

4.1.1.1 Pre-processing phase based taxonomy In this phase, the dataset is prepared and processed to be in a suitable format to extract the features that will be used to generate patterns that describe the behaviour of the applications. This phase is very important in the malware detection process because it determines the nature and strength of the features that will be extracted to construct apps' patterns and therefore this is reflected upon the strength of the used classifier. In general, there are four main methods used in this phase: visualisation-based analysis, static analysis, dynamic analysis and hybrid analysis. In the hybrid analysing method the application firstly analysed statically, and appropriate static features

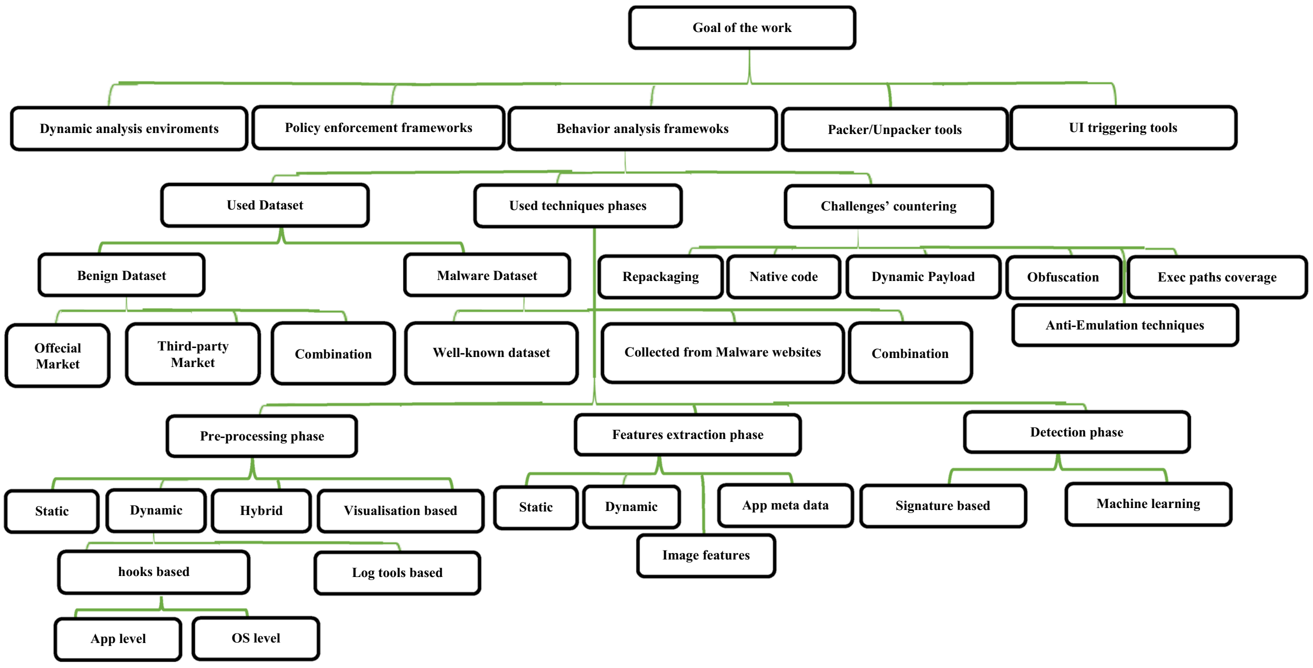

Fig. 2 The proposed taxonomy

are extracted, then the app is analysed dynamically to extract dynamic features. These four methods and some examples of each of which will be discussed in the following sections:

A. Visualisation-based analysis

Although image processing techniques are widely used in detecting malicious software targeting desktops, and despite these technologies have proven to be very effective in this area, these techniques were used very limitedly in the detection of malicious software targeting smartphones. However, this method has been used in some of the studied works, for example in [26], a framework has been proposed relies on converting the application's source code into an RGB image and using deep learning techniques to predict the app's class. To this end, the application was decompressed and its DEX code was extracted and represented as byte-code. After that, the colour channels' values (i.e. R, G, and B) were represented by splitting the hexadecimal representation of the instructions into three sections. At the end of this phase, the apps' source codes were converted to RGB images. Then the convolution neural network has been used as a prediction model in order to predict the apps' class. In [27], four files have been selected from the contents of the APK archive (i.e. Classes.dex, AndroidManifest.xml, Resources.arsc and CERT.RSA) to be converted to grayscale images. After that, GIST features were extracted from the constructed images and used as an input to Random decision forests classifier to classify the applications into malicious or benign. In [28], the source code of the application has been converted to grayscale images by decompressing the APK file and extracting the bytecode. After that, the Opcode Sequences with length 2 was extracted from the app source code. Then, the extracted code sequences were weighted based on its frequency in the training samples, and the weight values were considered as pixels in the app's images. After that, Latent Dirichlet Allocation (LDA) algorithm has been used to select the best sequences and reducing the dimensions in order to reduce the image noise and improving the detection accuracy. In the end, the optimized pixels were stacked in a vector and used as a signature to detect the Ransomware apps. In [29], a method has been suggested to simplify app's reverse engineering depending on the conversion of the application's structure into an image and manipulating apps' source using image processing techniques. The proposed method depends on app decompressing, DEX code extraction, dex code sections identifying and mapping each byte in the code into a single pixel in the image such that the pixel colour has been used to highlight the byte value. In [30], the static analysis was used to extract the application's source code and separating the instructions according to their importance into groups which were digitized based on Simhash and Djb2 hash functions. The obtained hash values were converted into an image. Finally, the convolutional neural network has been used to classify the applications into benign or malicious.

B. Static analysis

It is the most widely used and preferred method by many researchers, thanks for its low computational time, ease of implementation and effectiveness to some extent. In this method, the app source code is analysed without being executed in an emulator or real device. To this end, firstly, the APK archive is extracted to obtain classes, manifest file, meta-data information and media files. In this stage, the app's source code format is dex bytecode which not easy to be handled so it can be de-compiled to Java or Smali code to make it readable and easier for processing. Multiple tools can be used in this step, such as Apktool [31], which an open-source reverse engineering tool that can decompress APK archives and extract nearly the same of original contents of the applications, including the manifest file and all.Dex files as well as all other app's resource folders. As mentioned before, the DEX files can be converted to Smali code which is a middle representation between Dex bytecode and Java, easy to be read and effective features can be extracted easily from it. So, Smali code representation has been used extensively in the previous researches to extract the code-based features. Table 1 illustrates the most important tools that have been used in the static analysis-based researches which covered in this study.

It should be noted that these tools are commonly employed at app repackaging techniques to disassemble, modify, re-compile the APK archive. The re-packaging techniques are commonly used by malware developers, such that malicious code is added to regular applications and the repackaged malicious app is republished either in official or third-party stores.

As it was mentioned before the static method is the most popular analysing method, so this method is used in a lot of studied works. We will list some studied works that use the static analysis method and the rest of the works will be listed in one comprehensive table. In [32], a method for detecting apps' repackaging has been proposed. The proposed method depends on the fact that the attacker does not change some original application's data such as app name and app icon in order to benefit from the popularity of the original application. The framework consists of two tools: the first one is a client-side tool, which extracts the features from the application and sends them (the extracted features) to the second tool. The second tool is a server containing a database to be compared with the sent features to make a decision about the application. In [33], a static analysis-based framework has been proposed, where each of used permissions, sensitive APIs, monitoring system events and permission rate have been used as a feature for training and testing the used classifier. Then, a principal component analysis (PCA) algorithm was adopted for pre-processing the extracted features and an ensemble Rotation Forest RF has been used to classify the android apps into malware or benign. A dataset containing 2130 samples has been used to evaluate the proposed method's performance. Furthermore, the obtained results were compared with the results of a Support Vector Machine (SVM) model under the same experimental conditions. In [48], several challenges that facing the malware detection methods have been reviewed and some attacks that the conventional machine learning classifiers can fail to address have been discussed. Based on these considerations, three types of attacks that can poison datasets to demotivate the classifiers were presented and tested.

To address these attacks a detection system called KUAFUDET that significantly reduces false negatives and improves detection accuracy has been proposed. After that, Support Vector Machine (SVM), Random Forest (RF), and K-Nearest Neighbour (CNN) machine learning algorithms have been adopted as classifiers to distinguish malicious applications from benign ones. In [49], multiple static analysis features including permissions, requested permissions, filtered Intents, restricted API calls, hardware properties, code-related patterns, and suspicious API calls have been used to train and test the proposed model. In [50], a static analysis-based method for Android botnet detection

Table 1 The most important static analysis tools

Tool name	Description
Jd_gui [34]	It is a standalone graphical utility that displays.class files as a Java code
Dex2jar [35]	It is used to convert dex files to jar files and vice versa
Procyon [36]	It is a suite of Java metaprogramming tools focused on code generation and analysis
Ded, Dare [37]	It is used to convert dex bytecode files to.class files which can be processed by existing Java tools such as Jd_gui tool
Androguard [38]	A python tool that can be used to Disassemble/Decompile/Modify the DEC/ODEX/APK files' format
Jadx [39]	It is used to convert dex bytecode to java

has been proposed. First, the requested permissions and used features have been extracted from a dataset that contains benign and bot applications to create uniq++ue patterns that can identify botnet malicious activities. In the end, the machine learning techniques have been adopted to create classification models that able to classify applications as benign or bot based on the created patterns dataset. In [51], an automated malware detection system called MalPat that use permission-related APIs and Random Forest classifier has been proposed. To build the proposed model the APIs have been extracted from each app and the permission-API correlations have been revealed to construct unique patterns that can be used to distinguish malware from benign apps. The proposed method has been tested using a large-scale dataset and the obtained results have been compared with some previous approaches' results.

C. Dynamic analysis

Although the static analysis method is preferred by many researchers due to its speed, applying easiness and its low computational time it suffers from many weaknesses and shortcomings such as the inability to address the code obfuscation techniques or malicious content dynamic loading. The second method that can be used in the preprocessing phase is dynamic analysis. In this method, the application is executed in an isolated environment and the normal use of the application is simulated in order to collect as much information as possible about application behaviour. To this end, the application and a user interface events generating tool are installed in an emulator or a real device to simulate the app's normal execution and collect its behaviour. We have divided the works that used this method based on the techniques that used to track the behaviour of the application into hook-based and log tools based. We will discuss these two types in the following paragraphs:

1. Log tools based

In this methodology, the application is executed in a real device or emulator to monitor its behaviour using well-known logging tools. For example, in [52], a cloud-based dynamic analysis framework is proposed to detect android malware based on monitoring the Android applications' runtime behaviour, analysing the malicious URLs and correlate them with DNS service network traffic to find the presence of malware running at the network level. In this work, some open source tools have been used for app behaviour monitoring and analysing. In [53], a method for classifying android apps into benign and malware based on monitoring app's network behaviours has been proposed. The app outgoing network traffic has been monitored by running Tcpdump on the analysing environment. A machine learning classifier (i.e. Random Forest Enemy Learning Algorithm) has been applied to obtain behavioural models for each normal application category. If a new application's behaviour can be classified to any normal category it will be labelled as a benign app otherwise it will be labelled as a malware. In [54], a resource consumption features (i.e. network traffic, battery consumption and battery temperature) have been monitored using some logging tools. Then, multiple machine learning algorithms have been adopted in order to distinguish malware apps from benign ones. Several experiments have been conducted using different combinations of these three features and machine learning algorithms. In [55], a dynamic analysis-based framework has been proposed to detect android malware using machine learning techniques. To this end, a tool that extracts dynamic features automatically has been implemented and multiple experiments have been made to compare between emulator-based and phone-based malware detection. The features have been logged and extracted from the phone using Logcat logging tool and some scripts that written for this purpose. The features that extracted from each of the real phone and emulator have been used to train multiple machine learning classifiers separately and the obtained results were compared with each other.

2. Hooks based

The second used dynamic analysis methodology is known as instrumentation-based or hook-based systems. In this method monitoring points (hooks) are embedded within the code to record the application activities during the execution. These hooks can be used in monitoring the application execution, collecting information about the methods' pattern, tracing the executed instructions, retrieving the sequence of events or monitoring stored data flow. There are two trends for implementing instrumentation-based systems in the previous studies namely, app-level instrumentation and operating system-level instrumentation:

- App level instrumentation:

In the first method, the application is disassembled, and its source code is modified by adding methods that can log its behaviour (these methods called as hooks), and the app is reassembled. The imbedded hooks track the app behaviour during the execution and record a log for important behavioural information such as data flow or taint flow. For example, in [56], an APK-level instrumentation method has been used to monitor Android apps' suspicious API and understanding the malicious behaviours of Android apps. The proposed method does not require any OS level changes, so it is compatible with all versions of the Android operating system. The approach depends on APK file reverse engineering, instrument code addition and APK repackaging. After that, the APK file will be executed in an emulator to retrieve the potential suspicious behaviour based on monitoring the sensitive APIs. In [57], a hybrid system called AspectDroid that aims to detect apps' suspicious behaviours independent on Android runtime and system releases has been proposed. An instrumentation engine has been designed in order to achieve data flow analysis, resource abuse detection and suspicious behaviour analytics. In the static phase, the applications have been reverse engineered, the code that will be executed alongside the original code to perform custom logging and other analytical functions have been injected and finally the app has been recompiled. The instrumented application has been executed dynamically to track and log runtime events.

- Operating system level instrumentation:

In the second method, the operating system is modified by adding monitoring points so that it can log the application behaviour during app execution. In [58], the Android operating system's source code has been modified to insert hooks for API-level monitoring. Also, kernel level modifications have been conducted to make monitoring at the kernel-level. In [59], a hook-based taint analysis framework called TaintDroid has been developed on the top of Android system to track sensitive data flow within installed applications. The main goal of the proposed system is detecting and analysing sensitive information that leaving the system. Four taint propagation level namely, variable-level, message-level, method-level, and file-level have been tracked in the proposed system. In [60], DroidScope has been presented, which is a fine-grained dynamic binary instrumentation tool for Android that embeds two levels of hooks namely an operating system level hooks and Java level hooks. The proposed tool is able to demonstrate the interactions between Java and native components of malware samples. Also, the tool provides dynamic analysis of native instructions as well as Dalvik byte code. Table 2 illustrates the most important tools used in the dynamic analysis based researches that covered in this study.

D. Hybrid analysis

This method combines static and dynamic analysis in order to obtain a more accurate analysis of applications. Generally, the apps are reverse engineered to extract static features from the source code, after that, the apps are executed in an isolated environment, i.e. an emulator or a real device, to extract dynamic features. Therefore, despite its overhead and implementation complexity, this analysis method is considered the most profound and comprehensive method. Although the hybrid analysis method is relatively less commonly used in previous works, this type of analysis has been used in a number of works among the studied articles. For example, in [61], a hybrid Android malware analysis approach called mad4a has been proposed in order to benefit from the advantages of both static and dynamic analysis techniques. In the static analysis phase, the permissions have been extracted from the application's Manifest file and the extracted permissions have been mapped with the corresponding API calls in the Java source code. In the dynamic analysis phase, the malware

Table 2 The most important used dynamic analysis tools

Tool name	Description
MonkeyRunner [40]	Application UI trigger tool that can send random events such as touching the screen, swiping or pressing a widget…etc.
DroidBot [41]	Input generator that can send random or scripted input events to an Android app so that it can achieve higher test coverage and generate a UI transition graph (UTG) after testing
Android Debug Bridge [42]	A versatile command-line tool used to communicate with the device, installing, debugging apps or providing access to a Unix shell to run commands on a device
Logcat [43]	A command-line tool that dumps the log of system messages
Droidbox [44]	It is a dynamic analysis platform that employs an integrated system containing TaintDroid with a modification of Android's core libraries
Robotium [45]	Robotium is an Android test automation framework that has full support for native and hybrid applications. Robotium makes it easy to write powerful and robust automatic black-box UI tests for Android applications
Strace [46]	A Linux utility used to monitor interactions between processes and the Linux kernel. This tool is used to collect logs in order to record the executed Linux kernel system calls
Tcpdump [47]	It is used to capture and store all the network traffic in a Pcap file

and benign apps have been installed in the emulator and 500 different UI events have been generated using MonkeyRunner tool to simulate the app's normal usage. Then, the log file has been traced to monitor executed application's network usage (i.e. the size of downloaded or uploaded data and the number of incoming and outgoing connections). Also, an algorithm to detect the permissions' over-privilege has been proposed (permissions' over-privilege means that the applications demand more permissions than they actually use). In [62], a deep learning-based hybrid analysis framework called DroidDetector has been proposed to differentiate between benign and malicious Android applications. A total of 192 binary features were extracted using static and dynamic analysis, and the extracted features were used as input to the deep learning model. A number of experiments have been conducted using the proposed tool to verify the ability of the Deep Learning model to detect Android malware. The obtained results have been compared with the results of some conventional machine learning algorithms, namely Naive Bayes, C4.5, Logistic Regression, SVM and Multi-Layer Perceptron plexus. In [63], a hybrid analysis framework has been proposed to detect Android malware based on static features such as required permissions and sensitive API calls as well as some dynamic features such as network activity, file system access and interaction with the operating system. The extracted static and dynamic features have been used to train and test some machine learning algorithms i.e. Support Vector Machines (SVM), Decision Tree (C4.5), Artificial Neural Networks (MLP), Naive Bayes (NB), K-Nearest Neighbours and Bagging predictor. In [64], a hybrid analysis platform called Andro-Dumpsys has been proposed to isolate malwares from benign apps and classify the malwares into its families. The proposed method uses dynamic analysis for extracting Odex bytecode using RAM acquisition to obtain the fingerprint. Also, static analysis is used to extract multiple static features that have been used with the obtained dynamic features in order to classify Android applications. In [65], an attack tree was adopted to detect Android malware and a hybrid analysis prototype called AMDetector was proposed. The static analysis has been used to record app attacking abilities and suspicious applications' components, while in the dynamic phase, the events have been sent to the trigger tool based on the application components and the app's runtime behaviour was examined against attack capability. In [66], a hybrid Android malware analysing framework has been proposed. The permissions, API methods and classes have been extracted statically from APK archive, and system calls, event handler and network traffic have been traced dynamically. The extracted static and dynamic features have been used to generate behavioural patterns for Android apps classification.

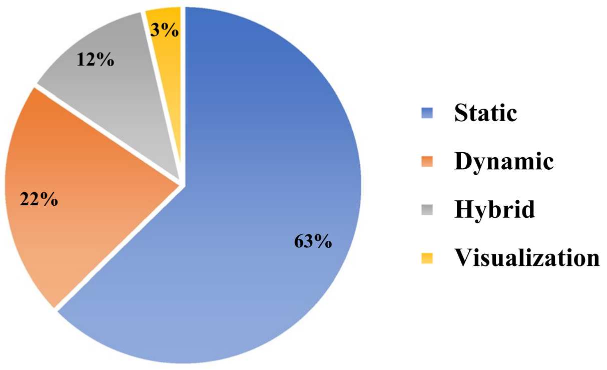

Fig. 3 The used analysis methods proportion

Figure 3 shows the proportion of the analysing methods that used in the pre-processing phase in the studied works.

4.1.1.2 Features extraction phase based After the pre-processing phase, the dataset will be more flexible and easier to be read and handled. In features extraction phase, appropriate features are extracted to form patterns that will be used for classifying applications and detecting any potential malicious behaviour. The extracted features' type and nature are varying according to the analysis method which performed in the pre-processing phase (visualisation, static, dynamic or hybrid). In this work, the features were divided into static features, dynamic features, hybrid features and image-based features, to the best of our knowledge, some of the suggested features' subclasses is novel.

A. Static features

In general, static features depend on parsing the app's source code to extract important information such as sensitive instructions, variables names, methods, classes, packages, strings, code context and sequence, control flow, or data flow. We will classify the static features as follows:

1. Manifest based features

The manifest file is the most important file in the Android application and considered as a controller or a roadmap that specifies how the application will be executed. Moreover, all components of the application i.e. activities, services and content providers must be declared in this file before using within the code. This file also contains a lot of other information that describes application behaviour such as, actions, intents, intent-filters, package name, category…etc. Most importantly, this file contains the permissions that required by the application at installation time and which are assumed to be important for the application to work correctly. Due to the importance of this file, its contents have been used extensively as features in the studied works. The Manifest-based features that used in the studied works can be classified as follows:

- *Permissions* A large number of the studied works depend on this type of features, such as, required permissions which are defined using <uses-permission>. For example, in [67], permission-based static analysis Android malware detection system has been proposed. The proposed system consists of three components: the first component is a signature database that stores behaviour fingerprints and the analysing results. The second component is an Android client used by end users to provide analysis requests. The third component is a central server used to communicate with both the signature database and the smartphone customer and manage the entire analysis process. The logistic regression has been used to classify a program as malware or benign, and 88% detection accuracy rate has been gotten. Furthermore, in some works, the permissions have been used with some other types of features such as code-based features or app metadata-based features. For example, in [68], the app's permissions and some other metrics have been used as features and the K-means algorithm has been adopted to cluster 18,147 benign Android applications into business apps or tool apps. Also, in [69], a lightweight static analysing method has been used to extract multiple features such as permissions, API calls and network addresses, then the extracted features have been embedded in a vector to create specific patterns that used in apps classification.
- *Intent filter* It is a powerful feature that can be used to detect suspicious behaviour of applications, as the intent filter can describe the exact details of the intentions of the application, including the actions, data, and intent's categories. For example, the Intent filter is usually used by malware apps to receive events such as BOOT_COMPLETED to launch malicious activity. This type of features was used extensively in previous works, for example [70] aims to study the efficiency of intents (explicit and implicit) in Android malware detection. To this end, a static analysis tool called AndroDialysis has been presented. The proposed tool extracts the intents (implicit and explicit), intent-filters and permissions from the applications, then multiple experiments based on different feature combinations (i.e. just intents, just permissions and intents-permissions) were performed. It was concluded that the android's intents are more effective than android's permissions in malware detection.
- *Hardware requirements* The app's hardware access requests (which are translated by requesting specific permission in the Manifest file) have been used as a feature to detect malicious behaviour of the applications in some works, like [69, 71].
- *Other manifest features* In some works, some other manifest file's information has been used, such as activities, services, package name and Intents. For example, in [72] each of number of activities, number of services and number of receivers were used with some other code-based features to distinguish between benign and malware apps.

2. Code-based features

This type of feature refers to all types of features that extracted from the app's source code whether Java or native code. The features extracted from the source code are very important for conducting a deeper analysis. In the studied works, the code-based features have been extracted from a variety of the source code's representation including opcode, bytecode, Smali and java code. We will classify the used code-based features as follows:

- *Instructions and commands based features* The code is parsed to extract certain instructions that may describe the potential suspect behaviour of the application. The following instructions and commands-based features have been used in the studied works:

  1. *API calls* The Android OS provides a wide range of APIs that can be used by the developers to access operating system resources or hardware in the device. The API calls are one of the most widely used features in the malware detection domain. Generally, these features are extracted by preparing a list of suspicious APIs and comparing it with the used APIs in the application's source code. API calls were used in a lot of works such as in [73], where a static analysis system called DroidMat has been presented. The proposed framework has used each of permissions, intents, and API calls as features to detect Android malicious apps. In [74], a static analysis framework called MOCDroid has been proposed to discriminate malware and benign-ware. A semantic intention has been extracted from third-party API call combinations (Import terms) and two sub-models that keep only relevant behaviours for malware and benign applications have been created. A candidate program

will be evaluated against these two sub-models to measure whether it fits better with benign-ware or malware. In some other works, API calls have been used with API's package level information and parameters. For example, in [75] a method that depends on API level information within the bytecode has been proposed to convey substantial semantics about the app's behaviour. The proposed method focuses on frequently used critical API calls, API's package level information, as well as API's parameters. Some other works are based on mapping the API with the desired permissions. For example, in [76], a tool called Stowaway that can detect permissions over-privilege by extracting the API calls and matching them with the required permissions has been proposed.

2. *Specific instructions* Some specific instructions or library calls that used frequently by malicious software developers have been used in previous works to detect malware. For example, DexClassLoader is an API that can be used by attackers to load the malicious content and execute it at the app execution time. Also, Crypto API is a library that can be used to encrypt the strings or other contents in the application. Each of DexClassLoader and Crypto APIs have been used in [77]. Also, other features such as Import terms were used in [74]. Moreover, the method calls and function arguments and instructions were used in [78]. Furthermore, some dangerous Linux commands such as Su, Chmod and Exec have been used as features to reveal the apps' malicious behaviour in some static frameworks like [48, 79]. Also, in [79], the apps have been checked to detect the presence of embedded Dex, Jar, So, or ELF files which can reveal the apps' behaviour.

- *Strings and network addresses* The most of malicious applications connect to a command and control (C&C) server to send data which can be collected from the victim as well as receive commands from the attacker. To this end, the server's address is placed within the code, so the source code can be parsed to find any IP or DNS address that can be used as a feature for identifying the program's behaviour such as in [69, 80]. Also, the strings in the app's source code give a great indication to the application behaviour, thus, it (the strings) have been used in many previous studies such as in [81–83].

3. Semantic features

We have categorized some of the features that have semantic characteristics, or any combination of other features displayed in a semantic way as semantic features. In other words, the semantic features include code-based features, manifest-based features or even program description-based features that combined or represented in a semantic way. The most important used semantic features in the studied researches are:

- *Control flow graph* It is one of the most popular used applications' behaviours analysing method. In this method, the application's source code is represented as a directed graph so that the nodes represent the instructions or code blocks and the edges represent the control flow between two nodes, i.e. represents the execution path passes between the instructions. Thus, CFG is a directed graph represents all possible execution paths in order to analyse all the execution scenarios of the application. In [84], the same concept of control flow graph was used to build API calls' graphs and construct semantic signatures to detect unknown malware variants. Also, in [85], the control flow graphs have been built based on native code for constructing semantic signatures that can be used to detect malicious behaviour in both bytecode or native code.
- *Data dependency graph* DDG is a common program analysis structure which represents inter-procedural flows of data through a program [86]. DDG is a directed graph such that its nodes represent the instructions in the application, and its edges represent data dependency between the application's instructions. Data dependency is obtained by data flow analysis, where a node n_1 is connected to a node n_2 ($n_1 \rightarrow n_2$) if n_2 uses a variable defined by n_1. This type of features has been used in some previous works, for example, in [87], the data flow analysing method has been used to construct a data dependency graph for user inputs and API calls, and the control flow analysis was used to reveal the Intent-based inter-app or inter-component events.
- *Taint flow* In this type of analysis, a sensitive data that produced by an API is tracked from the source up to the target. The API which produces the data is called source and the API that send the data to network, file or another target is called sink. The source-sink data flow strings are generated to represent the spreading of sensitive data in the apps. So, these strings can be used as a pattern to identify apps' behaviours. This type of features has been used in some previous works. For example, in [88], a framework for detecting information leakage based on source-sink API tracking has been proposed. Also, in [89], a flow analysis-based framework has been proposed for detecting the potential malicious behaviour based on tracking

the sensitive information from the source method to the sink method.

- *Other semantic features* In some works, a semantic pattern was generated based on the app's description which crawled from the app store and the generated pattern has been compared with the actual behaviour of the app. For example, in [90], the app's description was used to expect the permissions which necessary for app's proper work. Then the expected permissions were compared with the actual permissions that used in the app. It should also be noted that the code sequence can be used as a semantic feature to differentiate between benign and malicious applications. For example, in [91], the short sequences of the application's opcodes (i.e. opcode n-grams) was used to construct feature vectors that used in Android apps' classification. Moreover, the number of common permissions between a given application and a specific category pattern has been used as a feature in [92].

It is worth noted that a mix of variety static features types was used in the studied works. For example, in some works, a mapping between API calls and requested permissions has been used to avoid permission-over privileged such as in [93, 94]. Moreover, in [95], the permissions have been used with some app metadata like app's price, a number of downloads, user rating, and app description to distinguish the benign form malware apps. Furthermore, in some works, the permissions have been used with some code-based features such as API, network addresses, intents…etc. such as in [69].

4. Application's metadata-based features

We have classified the features that extracted from the description of the application or any information attached to the app as metadata features, this type of features includes the following:

- *App certificate's information* It includes the contents of the META-INF folder, which contains the application's signature, certificate, and the key that used to sign the app. This information can be used to compare applications' developer in some works. For example, in [32], the developer's signature has been used with some other descriptive information to detect the applications' re-packaging.
- *Play store descriptive information* This type of features includes all application-specific evaluation information that can be extracted from the store such as price, number of downloads, users rating, and so on. For example, in [95], the program rating information was used alongside some other descriptive information for categorising the Android applications.
- *Other descriptive features* Some other descriptive features have been used in some works. For example, APK file's hash value, application's name and application's icon have been used in [32]. Also, the number of lines in the manifest file, size of the APK archive and number of files and folders within the APK archive have been in [77].

B. Dynamic features

This type of feature includes all features that can be collected during application execution such as system calls, network activity, file system usage, etc. We will explain the most important of these features in detail:

1. *System calls* It is one of the most used dynamic features as an application needs to connect to the OS using dedicated system calls (switch to kernel mode) to achieve some tasks. The system calls can be tracked and stored in a log file that can be used in analysing the behaviour of the application. In some previous works, the system calls sequence or system calls' frequency have been used to create patterns that reflect the behaviour of the applications in a semantic manner. For example, in [96], the sensitive APIs' sequence and the number of used API call have been extracted during program execution then an improved Naive Bayes classification model has been used in order to classify the apps into malware or benign. In some other works, the system service calls' sequences were used instead of the system calls because that the system calls are composed of the function name which misses parameters information and cannot reveal the exact application behaviour. For example, in [97], a dynamic analysis framework based on the co-occurrence matrix of the system service calls has been proposed to detect android malware. Firstly, service interface call information of the running Android applications has been extracted to obtain the system service call sequence co-occurrence matrix. The obtained matrix has been normalized to construct vectors which have been used to train multiple machine learning classifiers.
2. *Network behaviour* In general, all malicious applications connect to the network in order to send the collected data, receive commands from its remote server or any other reason. Therefore, the network traffic that generated during the app's execution gives a good indication to the app's behaviour, so, this feature has been used in many previous works. For example, in

[53, 98], network activities and some other dynamic features have been used to differentiate between malware and benign-ware.

3. *Resources consumption-based features* Mobile phones are generally limited in terms of resources, such as battery, processor and memory. Therefore, the resource consumption has been used as a feature to detect malicious applications in some of the previous works. Generally, this type of works is based on analysing the difference between benign and malicious applications in terms of resource consumption since malicious applications generally consume more resources than the benign ones. This can be explained in that most of these applications (malicious apps) perform a task in the background or access hardware resources such as CPU, memory, Bluetooth and wireless devices [54]. Also, this type of features has been used with some other features for training and testing many machine learning classifiers in [99].

C. Hybrid features

As its name implies, this type of features includes a combination of static features and dynamic features to get a more accurate analysis and detecting any potentially malicious behaviour of applications.

D. Image features

This type of feature includes all the features which extracted from images whether grayscale or RGB images. As mentioned previously, the analysing methods that based on converting the malware source into an image have been used in a limited number of previous works. There are two trends followed in this type of works, the first one, image-based features are extracted to be used in training of the conventional machine learning algorithms that used in apps' classification. In the second trend, the malware's images are fed to deep learning models which can extract appropriate image's features automatically. For example, in [27], the GIST feature was extracted from images and used in a Random decision forests classifier training. Also, in [100], a method that based on the visualization of APK files as various image formats (Grayscale, RGB, CMYK and HSL) has been presented. After that, the GIST feature has been extracted from each image to create a features dataset which used to train and test multiple machine learning algorithms (i.e. Decision Tree (DT), Random Forests (RF), and K-Nearest Neighbour (KNN)). In [28], weights have been given for each pixel in the image and the best sequence of pixels have been chosen and used as a signature for detecting the malicious applications. In the rest of works, the deep learning techniques have been used, for example, the convolution neural network has been used in each of [26, 30].

Figure 4 shows the proportion of features that used in the studied researches. Also, Fig. 5 illustrates the proposed taxonomy for the features that used in the studied works.

4.1.1.3 Detection phase After the feature extraction phase, the patterns that can be used in detecting the apps malicious behaviour or in classifying benign apps into multiple categories will be generated based on the extracted features. In most of the studied works, these patterns are represented as binary vectors, where 0 represents the case when the feature is not used and 1 represents the case when the feature is used. It is possible to use labelled or unlabelled data according to the detection method. In general, there are two approaches were used in the researches that covered in this paper, namely, signature-based and machine learning based:

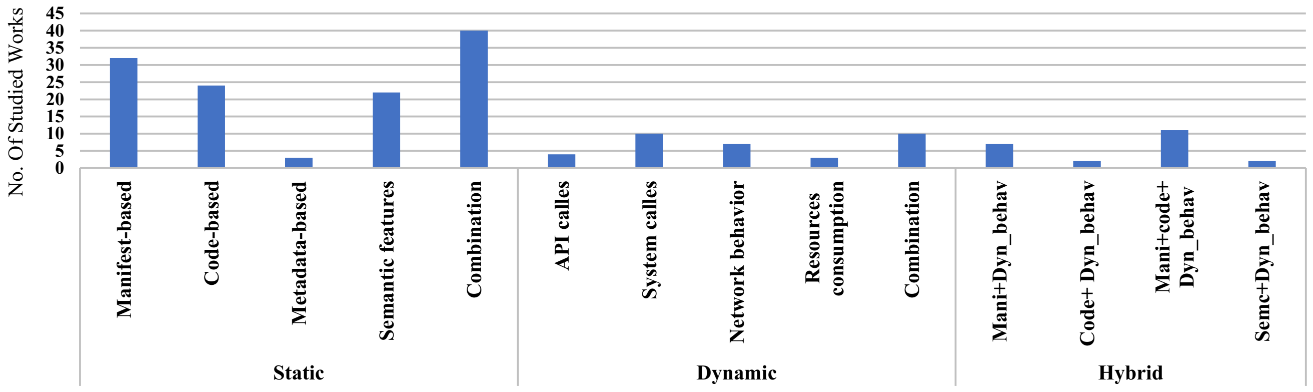

Fig. 4 The proportion of features that used in the studied researches. Combination sections in the static and dynamic features indicate to the works that use a mix of different types of static or dynamic features. Mani+Dyn_behav: maifest based features and dynamic features, Code+Dyn_behav: code based features and dynamic features, Mani+code+Dyn_behav: maifest based features and code based features and dynamic features, Semc+Dyn_behav: semantic based features and dynamic features

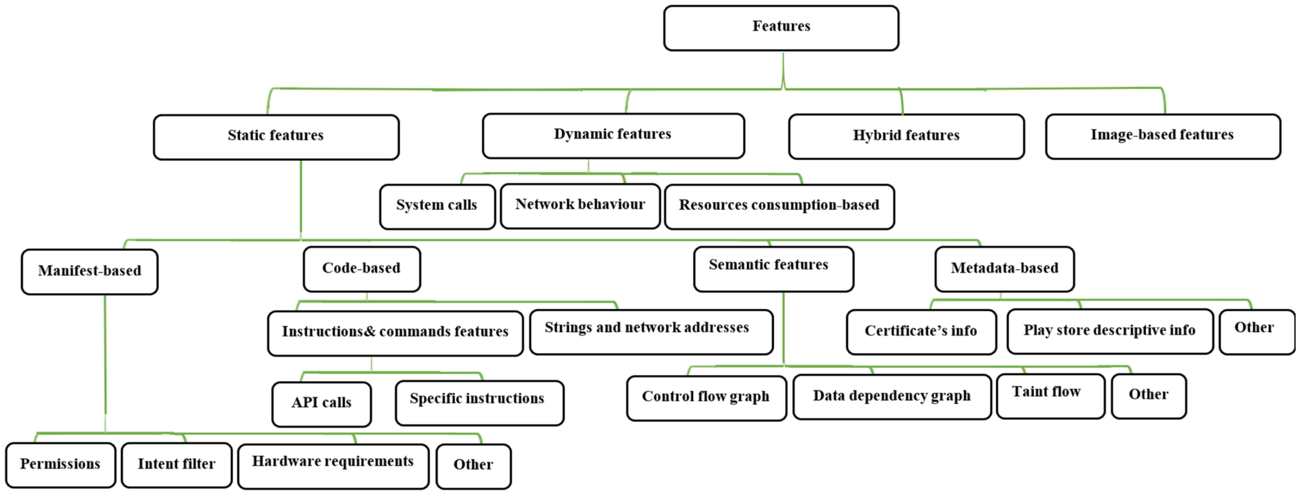

Fig. 5 The proposed used features taxonomy

A. *Signature-based approach* It is one of the most common traditional malware detection methods, where, a pattern for each application or set of patterns that describe the behaviour of a particular malware family are generated and stored in a signature database. After that, in order to examine the behaviour of any application, its pattern will be extracted and compared with the patterns that stored in the signature database. In case of matching the app's pattern with any malicious signature, it will be judged that the app contains a malicious behaviour. This method was widely used to detect malicious applications that targeting computers, for example in [101], a hybrid method that based on genetic algorithm and Tabu search algorithm has been proposed to build a signature database for detecting malware targeting computers. Moreover, this method is used to detect Android malware, for example, in [102], the patterns database was created depending on the topology graph that constructed based on APIs and classes to reflect the actual behaviour of the Android apps. Next, new applications were analysed by matching their topology graph with the signature database, so that if the examined app contains a subgraph which is monomorphic to one of the database's signatures the used API set in every node will be compared. If the similarity of API sets reaches a specified threshold the app will be considered as malicious. Also, in [103], a statistical features-based signature approach has been proposed to detect obfuscated and repackaged malware variants. The proposed method uses statistically robust features that constructed using similarity digest hashing scheme (SDHash scheme) to generate a variable-length signatures database. In [104], a code path-based signature database has been constructed to rank apps as high-risk, medium-risk and low-risk. In [105], a Program Dependence Graphs (PDG) has been used to construct semantic code-based signatures to detect the code similarity between apps. Also, in [85], semantic-based signatures have been generated based on the Annotated Control Flow Graph (ACFG) to detect suspicious behaviour in app's native code. The analysed applications have been broken up into a set of ACFGs to construct its signature, and if the constructed signature matches a malware pattern within a given threshold, the app is labelled as malware. In [106], a method that takes into account both app descriptions' information (which are indicative of apps' topics) and sensitive data flow information has been proposed to characterize malicious apps. The proposed method based on mining of the topic-specific sensitive data flow signatures to improve malware characterization. The topic-specific signatures have been generated by computing the information gain ratio for each data flow pattern that seen in the apps from a specific category (specific-topic).

Furthermore, in [82], a generalized signature-based method has been proposed to overcome the lack of robustness of the traditional signature-based approach. It has been proposed to create malware families-based signatures instead of malicious app-based signatures. The detection was made by estimating the similarity between the target app's DEX file and each family signature. It has been stated that the results of the proposed approach shown improvement in detection accuracy compared with the previous static approaches. It should be noted that the signature-based approach suffering from weaknesses such as the ability to detect only known malware types and failing to detect unknown malware, polymorphic malware or zero-day attacks.

B. *Machine learning based approach* Because of the previously mentioned limitations of the signature-based detection method, there is an urgent need for new detection methods that can handle the huge number of polymorphic malware and the new malware development technologies. Therefore, machine learning and data mining algorithms have been introduced into the malicious applications detection domain and these algorithms have proved its efficiency. The bulk of works that have been studied in this paper have used supervised learning algorithms and a small part of works has used some unsupervised clustering algorithms such as the k-means algorithm. Due to limitations in space we will briefly list some of the algorithms that have been used frequently in most of the studied works.

   1. *Classification algorithms* This type of algorithms is based on supervised learning, where a part of the dataset is used for training and the other part for testing, and the training dataset must be a labelled data. Since the malware classification problem is a binary classification problem, one of the most widely used methods is SVM (Support Vector Machine). SVM is a non-probabilistic supervised binary classification algorithm relies on finding such hyperplane that would separate the data classes in the best way. In other words, it aims to find a hyperplane that separate data with maximum margins. The second heavily used classification algorithm is Naive Bayes, which based on Bayes theorem and can be used in both binary and multi-class problems. This classifier evaluates the probability of each feature independently, regardless of any correlations, and makes its prediction based on the Bayes Theorem [107]. Also, Decision Tree one of the commonly used classification algorithm, this algorithm depends on building a decision tree based on the data entropy. Each node of the tree selects a feature and splits its sets of samples into subsets until the classes can be inferred [78]. Also, Random Forests (RF) algorithm is one of the most popular used classification algorithms. RF consists of collections of decision trees and aims to produce prediction accuracy better than what the normal decision tree can do. In some other works Bayesian network has been used, Bayesian network is a probabilistic graphical model that represents a set of variables and its dependencies using a directed acyclic graph (DAG). Furthermore, the logistic regression was used frequently in the studied works, which is a statistical regression model uses a dependent variable to estimate the probability of binary response based on multiple features [78]. Moreover, Adaptive Boosting (AdaBoost) algorithm also was used in some previous works. AdaBoost is an ensemble algorithm that can be used to enhance the performance of any machine learning algorithm and preferred to be used with weak learners. In other words, the Boosting refers to an ensemble method that creates a strong classifier from a number of weak classifiers. Also, the K-Nearest Neighbours algorithm has been used in multiple works, this algorithm depends on the majority of the closest neighbours to predict the sample's class. A variety of the above-discussed algorithms have been used in the works that covered in this paper, we will list some of these works in the following paragraphs. In [75], a generic data mining approach has been followed to create a classifier that can detect malicious behaviour in Android applications. A large set of malware and benign apps has been analysed and the API list for each class has been constructed. Then, the frequency analysis has been adopted to distinct the API set which is more used in malware apps than benign ones. After that, Decision Tree, K-Nearest Neighbours, and linear SVM have been adapted to differentiate between malware and benign ware. In [69], multiple static features have been extracted and a linear support vector machines (SVM) classifier has been adopted to distinguish between benign and malicious behaviour. In [108], static analysis tool called Manilyzer has been proposed. Manilyzer is based on the manifest file's information and machine learning techniques. Naive Bayes, Support Vector Machine (SVM), K-Nearest Neighbours (KNN), and C4.5 Decision Tree algorithms have been adopted to distinguish between the malicious and benign apps. In [109], an ensemble classifiers-based method is presented to detect Android malware. The proposed method is based on extracting multiple features from a data set and training the ensemble classifiers using a collaborative approach. State of the art ensemble schemes such as AdaBoost and Bagging have been adopted and the collaborative approach has been used for boosting some weak classifiers like J48 (Weka's implementation of the Decision tree) and Random trees. The proposed method's performance

has been compared with the performance of some state-of-art learning techniques. In [110], a machine learning-based static analysis framework called ANASTASIA has been proposed. To this end, an Androguard-based tool called uniPDroid has been implemented to extract as many informative features as possible from Android applications. After that, several machine learning techniques such as AdaBoost, Random Forest, SVM, K-NN, Logistic Regression, Naive Bayes, Decision Tree Classifiers and Deep Learning have been adopted to classify an Android application as malware or benign. In [111], an Android malware detection method that combines 2-level machine learning with static analysis techniques has been proposed to optimize malware detection. In the first level, the Support Vector Machine has been used, while three different algorithms have been adopted in the second level (i.e. SVM-NB (SVM and Naive Bayes), 2-level Linear-SVM and 2-level RBF-SVM). In [112], an Android malware detection method that use the control flow graph's community structure analysis has been introduced. The proposed method adopts three features extracted from community structures to be used in training and testing some machine learning classifiers namely Decision Tree, SVM, NaiveBayes, and BayesNet. Also, in [50], Naive Bayesian (NB), Support Vector Machine (SVM) and reduced error pruning tree (REPTree) have been used for botnets classification. Moreover, In [78], multiple classification and clustering algorithms such as SVM, Naive Bayes, Decision trees, AdaBoost and Simple K-means have been used to distinguish between android benign and malware apps. Furthermore, a variety of machine learning algorithms including SVM, Random Forest (RF), and K-Nearest Neighbour (KNN) have been used as classifiers in [48, 77].

2. *Clustering algorithms* This type of algorithms is based on unsupervised learning and used when the data is unlabelled or only a small part of it is labelled. The clustering algorithms are used to divide data into clusters depending on the amount of similarity between its samples. So, the distance measures methods such as the Euclidean distance or Cosine distance can be used in this type of algorithms to measure the similarities between the data samples. We found that the K-means algorithm was used in most of studied works that use this type of algorithms. This algorithm aims to assign each dataset's sample into one of K clusters by working iteratively and re-calculating the clusters' new centroids according to data locations. This algorithm was used in some works, for example, in [73], multiple static features have been extracted and the k-means algorithm has been used to divide applications into multiple clusters. Then the KNN algorithm has been adopted to classify applications as benign and malicious. Also, in [113], the k-means algorithm has been used as a first phase to cluster the extracted features into multiple clusters. After that, machine learning classification algorithms have been used for classifying the apps to multiple classes. Moreover, in [114], a hybrid classification method has been proposed to classify Android apps using k-means algorithm as a clustering phase followed by the J48 and ID 3 classifiers as a classifying phase. Also, the K-means clustering algorithm has been used with some classification algorithms in [78].

   It worth mentioned that N-fold cross-validation with $N = 10$ has been used in most of the proposed conventional machine learning algorithms-based detection methods. Moreover, the machine learning algorithms have been implemented using the WEKA tool or Scikit-learn Python library in most of the studied works.

3. *Deep learning* This model is a neural network consisting of a large number of neurons distributed over multiple layers namely, input layer, output layer and multiple hidden layers. Deep learning techniques outperform conventional machine learning techniques by its ability to extract features automatically for using in classification rather than extracting features by the analyser to insert it into the classifier in conventional techniques. Deep learning techniques were used in a small proportion of the studied works. For example, in [115], raw API method calls have been extracted from Android apps dataset. Then, a semantic vector was created for each application. Finally, the constructed vectors were used to train a multi-layered neural network which has been used in applications classification. In [116], a dynamic analysis framework has been developed to detect the apps' malicious behaviour. The proposed method is based on a deep learning architecture with Stacked AutoEncoders (SAEs) in order to classify android apps as malicious or benign. In [62], a Deep Belief Networks (DBN)-based deep learning model has been adopted to characterize Android apps. In [26], the classes.dex file which contains the core of the execution logic of Android app has been converted into RGB image and the constructed images are fed to a convolutional neural network for automatic feature extraction and clas-

sifying the apps to malicious and benign. In [30], the apps' source codes have been extracted and parsed to calculate and digitize the importance of terms in the total code. The digitized values have been converted into image and the constructed images have been fed to a convolutional neural network which adopted as a classifier. Also, in [89], a static analysis framework called DroidDeepLearner has been developed to characterize Android malware. Multiple static features have been extracted and a deep learning model has been adopted to distinguish the malware and benign-ware. Moreover, In [49], high-dimensional feature vectors have been built to increase the accuracy of malware detection and a multiple convolutional neural network (CNN) models have been adopted to detect Android malware. A serial convolutional neural network architecture (CNN-S) was used with a non-linear activation function to increase sparseness and dropout technique to prevent overfitting. Finally, a deep autoencoder has been used as a pre-training method of CNN to reduce the training time and it was stated that DAE-CNN can learn more flexible patterns in a short time. Furthermore, a Convolutional Neural Network (CNN) and Recurrent Neural Networks (RNN) with Long Short-Term Memory (LSTM) have been adopted as classifiers in [117]. Figure 6 shows the detection methods that used in the studied works.

It is worth noted that, we classified the works which use pattern matching method such as [52, 118] under the signature-based works. Moreover, the 'Combination' sub section in the machine learning section indicates to the works which used both of supervised and unsupervised machine learning algorithms, such as [113, 114].

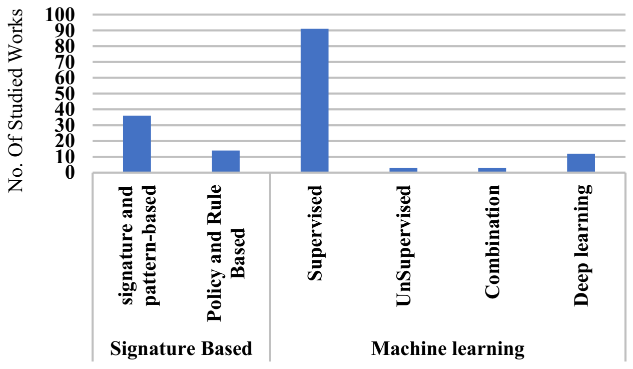

Fig. 6 The proportion of used detection method in covered studies

It is worth mentioning that there is a fourth phase that can be added to the previous three technical phases, especially in case of using the conventional machine learning algorithms whether supervised or unsupervised. This phase called feature selection phase we will discuss this phase in the following section.

4.1.1.4 Feature selection phase Features selection is performed to reduce the dimensions of the features dataset by filtering the redundant or irrelevant features that can be led to several problems such as, misleading the learning algorithm, reducing generality (overfitting), and increasing model complexity. So, the features are filtered according to its representative capacity for the entire dataset. The features selection algorithm can be said to be effective if it can increase performance, minimize the data set dimensions and reduce the execution time. There are many features selection techniques used in the works that studied in this paper, we will explain just the feature ranking techniques (which the most used features selection methods in the studied works) due to the space limitation.

Feature ranking algorithms In general, these algorithms use certain mathematical models for ranking and selecting the features which have the highest-ranking value. Information Gains is one of the most famous feature ranking algorithms. This algorithm depends on calculating the entropy values of the features and selecting the highest gain features to be used in training the classification model. This algorithm is the most used algorithm in the studied works, such as in [48, 95, 111, 113, 114]. In [119], the features have been ranked using mutual information method to select the top 10, 15, 20 and 25 features. The mutual information (MI) measures the amount of information that one random variable has about another variable. This definition is useful within the context of feature selection because it gives a way to quantify the relevance of a feature subset with respect to the output vector [120]. Mutual information method was used also in [121] with two other feature selection methods, the first one is Chi squared which based on ranking the features using the Chi square scores and choosing the top ranked features for training the model. The second one is One-way Analysis Of Variance (ANOVA) which based on ranking the features using the one-way ANOVA F test statistics and choosing the top ranked features for training model [122]. In [123], two different feature selection algorithms have been used, namely, Chi squared and Relief. Relief is a feature ranking method that based on weighting the features using values between -1 and 1 such that more positive weights indicating more predictive features [124]. Also, in [109], each of Chi Square, Relief (RF) and Information Gain (IG) algorithms have been tested. In [83], the features have been

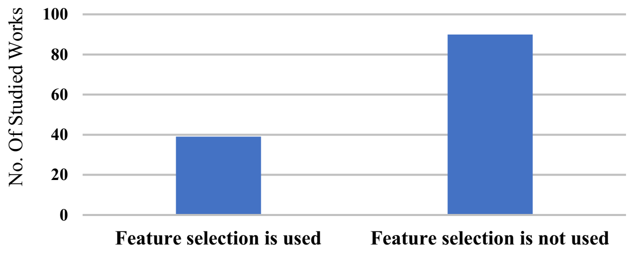

Fig. 7 The proportion of works that used features selection

selected based on the frequency of its appearance in given class's samples. In other words, if the number of feature's appearance times in a given class's samples was more than a specified threshold then the feature will be selected. Figure 7 illustrates the proportion of the behaviour analysing frameworks that used feature selection methods.

4.1.2 Used evaluation dataset based taxonomy

The dataset that used to train and test the proposed malware detection systems is one of the most important criteria for judging the systems' strength. So, we proposed to classify the studied works based on the used evaluation dataset which generally composed of malware and benign dataset.

4.1.2.1 Benign dataset Generally, the benign-ware dataset in most of the studied works such as [33, 51, 54, 61, 67, 71, 85–89, 125] was collected from the official Android app market (Google Play). In some other works, the benign dataset has been collected from the third-party markets such as in [49, 94, 126, 127]. And in some other works, a benign dataset that includes apps from each of the official and third-party markets has been used such as in [50, 128, 129].

4.1.2.2 Malicious dataset We found that well-known malware datasets such as Malgenome [130] have been used in the most of studied works. The well-known datasets have been used in many works such as [32, 48, 52, 89]. Also, special datasets that collected from the internet have been used in some works such as [33]. Furthermore, a combination of the well-known datasets and some malware samples that collected from the internet have been used in some other works, such as [48, 61]. Figure 8 shows the used datasets in the studied works.

4.1.3 Challenges' countermeasure-based taxonomy

All the analysis methods that commonly used in previous works (i.e. static, dynamic and hybrid) have some challenges and weaknesses. Therefore, we have proposed to

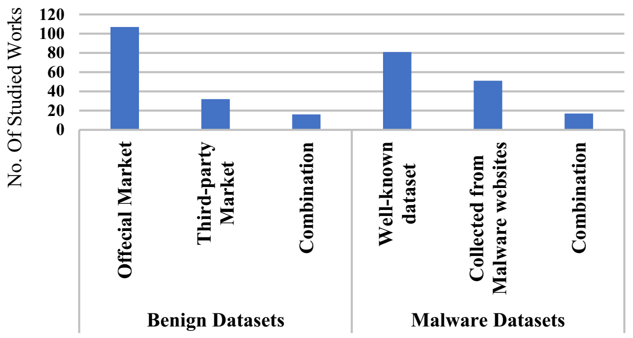

Fig. 8 The proportion of used dataset in previous works

classify the previous works based on these challenges and the countermeasures that used in the studied works to address these challenges.

4.1.3.1 Static analysis challenges Although the static analysis is a lightweight detection method that can detect Android malware quickly with low computation complexity and fairly high performance, it is still facing some problems and challenges such as obfuscation techniques and code's dynamic loading. The most important of these problems have been discussed in the following sections.

A. *Obfuscation techniques* Although these techniques are recommended to developers for many reasons such as protecting their applications from reverse engineering, this technique is one of the most important methods that used by malware developers to overcome static analysis methods and hiding the malicious behaviour of applications. In particular, the code obfuscation changes the size and the contents of the APK file without modifying the logical behaviour of the malicious app [131]. There are many techniques used to achieve this purpose, the most important ones will be discussed below:

1. *Name obfuscation* It is one of the simplest obfuscation methods. In this method the package name of malicious application or some other expressions in malware's code (such as class names or methods names) are changed to skip some analysis methods that use simple features or descriptive data in detecting the malicious apps.
2. *Control flow obfuscation* As mentioned previously, the control flow and data flow have been used extensively in the static analysis methods to track application execution paths or data flow paths in app's methods in order to detect any suspicious behaviour, such as in [84, 85, 132]. The call-graph

of the malicious application can be manipulated to defeat detection methods which based on the program's control-flow-related patterns. Goto-obfuscation is one of the most popular control flow manipulation methods, where, the Goto instruction is used to make jumps within the code and manipulate the original sequence of instructions. Furthermore, to get a deeper obfuscation, junk methods can be added and called within the code [131, 133].

3. *Strings encryption* As mentioned previously, the strings (such as IP addresses, domain names, or premium numbers, that can be used to connect the malware to its C&C server or send a premium message from the victim's device) that embedded within the source code have been used as a feature in many static analysis-based works. So, if the malware developer has performed string encryption, the plain string will never be found in the code, thus these analysing methods will often fail. The encrypted string will be decrypted only when it is processed during the app execution [131].
4. *Class encryption* It is an advanced obfuscation method, in which the entire class is encoded, compressed and stored in a data array. Then, a method is created to decrypt and load this class at the execution time such that obfuscated class will be decrypted, decompressed, and then loaded to memory [134]. This technique can greatly increase the overhead as a lot of instruction is added but it is one of the most effective ways to defeat static analysis techniques.
5. *Reflection* In this technique, the classes and methods can be accessed and inspected as well as new instances can be initiated, or the methods can be invoked at runtime without the need to its frank name at the compilation time. For example, the class invocation or class's new instance creation can be achieved using literal strings for obfuscating the code and making it harder to be analysed, As shown in list 1:

   List 1. Class invocation reflection's example [135].

```
Original code:
    System.out.println("Hello World.");

Obfuscated code:
    Class c = Class.forName ("java.io.PrintStream");
    Method m = c.getMethod("println",   new Class[] {
    String.class});
    m.invoke(null, new Object[] { "Hello World. " });
```

6. *Junk code insertion* A junk code can be injected into the classes and methods code so that the injected junk code can be executed without affecting the execution of the application (maintains the function of the application). This code is called dead code or no-operation code [136]. This method is one of the famous obfuscation methods that used to manipulate the sequence of the code in order to defeat the code sequence-based analysing methods.

The treatment of some obfuscation techniques which described in the previous section has been addressed in a very limited number of studied works. For example, in [119], an extensive mixed features set has been used with Random Forest classifier to provide robustness and resilience against code obfuscation and other anti-analysis techniques. Also, in [102], an Android components-based topological graph has been used to construct a new signature that aims at detecting malware variants produced using various obfuscation techniques. Also, in [103], a framework called DroidOLytics has been proposed to detect repackaging and code obfuscation in Android apps based on robust statistical features-based signatures. Moreover, in [104], a framework called RiskRanker that able to detect the usage of obfuscation and dynamic payload loading techniques has been proposed. Also, the proposed method can analyse whether a particular application has dangerous behaviours such as launching root exploits or sending background SMS messages.

B. *Native code execution* The Android system provides the possibility to write a part of application code as native libraries accessed using the JNI interface. This property has been exploited by malware developers to write the malicious part of the code using native libraries in order to make its analysis more difficult. An example of a malicious application that use this method is DroidDream, in this malware the malicious content was written using native code and placed in a non-standard location. Furthermore, the used native code can be encrypted and embedded in the app code, as in DroidKungFu malicious applications family [137]. Generally, the native code analysis is considered much harder than the bytecode analysis, so a few previous studies focus on the analysing of the native code. For example, in [138], a methodology that depends on statically analysing of native code's API calls using binary slicing and known compiler optimization methods has been proposed to analyse the use of native code for calling the Android APIs. In [127], an improved copy of Mobil-Sandbox hybrid analysis framework which developed in [139] has been implemented by adding machine learning techniques to the original framework. In the static analysis phase, the application has been de-compiled, and the Manifest file has been parsed in order to gather the permissions and intents that looks to be suspicious. In the dynamic analysis phase, the application has been executed in a sandbox to log all performed actions including Native API calls. In [140], each of the requested permissions, broadcast receivers, and the presence of embedded native code have been used as features and the random decision forests have been used as a classifier to detect the app's malicious behaviour. In [60], a dynamic analysis platform has been developed for tracking information leakage through both the Java and native components based on taint analysis.

C. *Dynamic payload* The Android system allows the applications to load DEX, JAR, SO and ELF files and execute these files at the execution time. This property has exploited by malware developers to embed the malicious content within an encrypted Dex, Jar or native code files so that it is called at the app execution time, decrypted and then executed. In some cases, malicious content is not embedded in the application, but downloaded from a remote server at the execution time, which, making impossible to detect the malware using static analysis methods. The dynamic payload has been addressed in some previous works, for example, in [141], a hybrid analysis framework has been proposed to detect the dynamically-loaded native code by collecting the system calls that made by the native code. In [142], a dynamic analysis system called DroidTrace that focuses on exploring the behaviour of dynamic payloads has been proposed. The Ptrace tool has been used to monitor the system calls and the forward execution has been used to trigger different dynamic loading behaviours on the target process that running the dynamic payloads. It has been stated that the proposed framework can monitor all dynamic payloads behaviours on both java and native code level, can be executed on real devices for all Android versions, and can detect four kinds of behaviours, i.e. file operation, network connection, inter-process communication, and privilege escalation. Also, in [139], a framework called DroidAnalytics that combat against malware which uses repackaging, code obfuscation or dynamic payloads has been developed.

D. *Applications repackaging* This method is one of the most common methods used for developing malicious applications. Simply, app repackaging is based on decompiling one of the popular applications and adding the malicious content to its source code. After that, the application is re-compiled, signed with a new signature and re-published on the official or third-party app stores. This type of techniques has been addressed in some of the studied works, for example, in [32], a method for detecting the repackaging of applications has been proposed. The proposed method depends on the fact that the attacker does not change some original application's data such as app name and app icon in order to benefit from the popularity of the original application. In [143], a static analysis approach called MIGDroid was introduced to detect App-Repackaging based Android malware. Firstly, the API calling sequences were extracted, then the method invocation graph was constructed. After that, the constructed graph was separated into subgraphs and the subgraph's threat score was calculated according to the sensitive APIs that invoked in each of these subgraphs. In the end, the sub-graph that exceeding the predefined threshold was labelled as harmful. Also, in [144], a prototype system called DroidMOSS has been developed to detect repackaged apps in the third-party markets. A fuzzy hashing technique has been adopted in order to localize and detect possible changes in the repackaged app. The proposed system was tested to identify repackaged applications in six different third-party markets and it has been found that 5–13% of the applications hosted in the tested markets are repackaged apps.

4.1.3.2 Dynamic analysis challenges The dynamic analysis is used to avoid the weaknesses of static analysis, but this type of analysis faces some challenges also, the most important of which are:

A. *Coverage of all execution paths* It is one of the most important challenges facing dynamic analysis method, as this method needs to cover all possible execution paths of the application in order to fully analyse the app's behaviour. Usually, any activity in the application contains more than one element (buttons, text box, radio button…etc.), and since most of the UI triggering tools such as Monkey Runner generate random events to interact with apps, so some of the execution's paths can be missed. Although it is not possible to ensure that this problem can be completely addressed, it was tried to address this challenge in some of the studied works. For example, in [57], it has been tried to mitigate the effects of this problem by building Python scripts which triggers a series of system and user events to more fully cover an app's functionality. The proposed tool combines some open source tools together with custom-built instrumentation programs. Also, in [145], a hybrid method was proposed to improve the automatic user interface trigger by hybridizing the Android MonkeyRunner with DroidBot UI trigger tool.

B. *Anti-emulation techniques* There are many ways and techniques that malware developers can follow to find out if an application is running in an analysis environment or normal environment. For example, a malicious application can detect the emulator simply by examining the value of certain hardware's identifiers such as the value of IMSI and IMEI (its values are usually zeros in the emulator). The malicious application can also identify the analysis environment based on the difference in resources' capacity (processor, memory…) between analysing environment and regular environment. This technique is based on that the emulators' resources are usually more than the real devices' resources (the emulators run on desktop computers) [146]. It is also possible to discover if the malicious application works in an analysing environment by observing the interaction of the user with the app. This method is based on the fact that the frequency of events which generated by the UI triggers is much higher and irregular compared to that generated by the regular user. If the malware detects that it is executed in an analysing environment it will hide the malicious behaviour and performs benign tasks to evade the detection. It is worth noting that, some malicious applications conceal their malicious behaviour for a period of time in order to skip the analysing time. There are no clear solutions for this problem was proposed in the previous works excepting some simple procedures that have been followed in a very limited number of works to mitigate the effects of some of these techniques. For example, in [55, 147], some procedures such as adding contact information and changing the emulator's IMEI number to a real IMEI number have been conducted to enhance the used sandbox emulator.

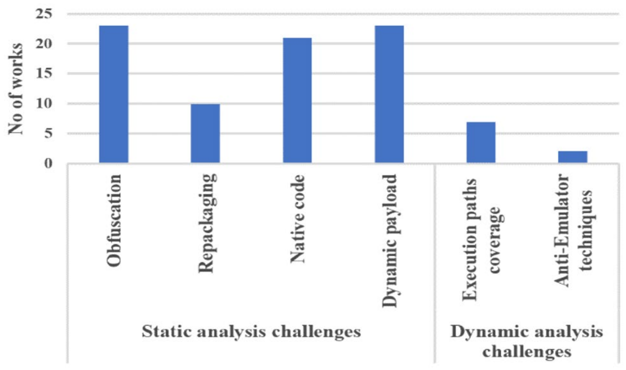

Fig. 9 Countering the challenges of the used analysis method in the studied works

Figure 9 illustrates the works that have been attempted to address the challenges facing the analysis method used in each of which. It is worth mentioning that a specific challenge was considered as 'addressed' in a particular work once there was any action within the work to mitigate the challenge's effects.

Furthermore, Fig. 10 shows the taxonomy of the used analysis methods' drawbacks (challenges).

4.2 Dynamic analysis environments

This category includes the works that aim at developing dynamic analysis environments that can be used in analysing the apps' behaviours. This type of works includes the works that aim at modifying the Android operating system to construct a new operating system that can monitor and analyse the applications' behaviours. Also, it includes the works which aim at designing and developing an analysis infrastructure (Test-Bed). For example, in [58] an environment for analysing Android applications called AppsPlayground has been proposed. The proposed framework aims to provide an automatic environment for dynamic analysis of applications by integrating a number of detection, exploration, and disguise techniques. The proposed

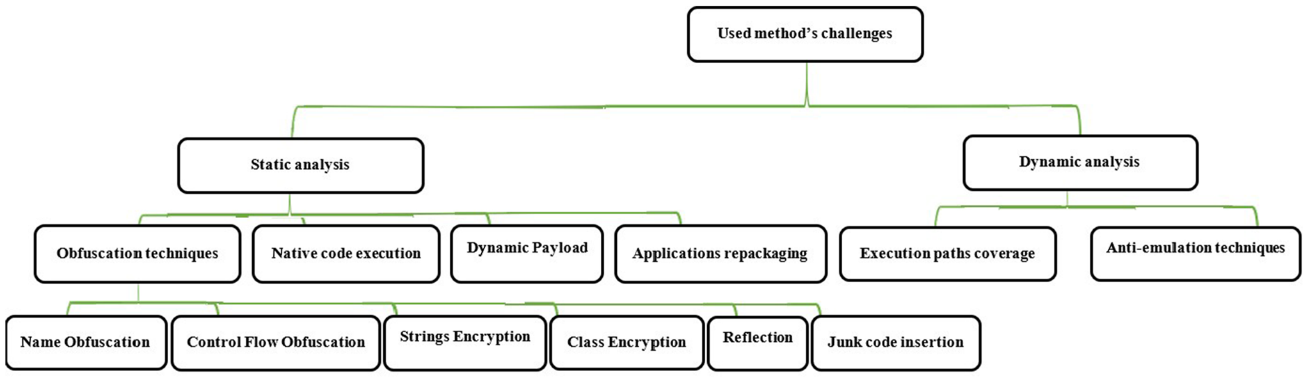

Fig. 10 The taxonomy of the used analysis methods' challenges

platform is built on a head of the standard Android emulator that comes with the Android SDK. In [148], a dynamic analysis platform called Andlantis that can handle more than 3000 Android apps per hour has been proposed. The system is able to collect valuable app's behaviour data which helps reverse-engineers and malware researchers to identify and understand anomalous applications' behaviour. The proposed framework can run the Android operating system in a virtual environment that simulates a physical device. Also, in [99] a System called STREAM has been proposed to automatically train and evaluate Android Malware classifiers. STREAM has introduced automation approaches that address APK setup, user input creation, feature vector collection, and malware classification. It also provides an effective method for quickly profiling malware and training machine learning classifiers. Furthermore, in [149], a virtual machine introspection (VMI) based dynamic analysis platform called CopperDroid has been developed to construct detailed behavioural profiles for Android malware. The proposed system constructs the malware behaviours by monitoring system calls and automatically constructs the events produced by well-known process-OS interactions as well as intra- and inter-process communications. Also, it was stated that CopperDroid can capture actions that initiated both from Java and native code execution.

4.3 Policy enforcement frameworks

It includes the works which aim to construct a set of rules to be enforced at apps installation or execution time. For example, in [150], the design and implementation of XManDroid (eXtended Monitoring on Android) which expands the Android permissions framework has been presented. The XManDroid performs runtime monitoring and analysing of communication links across applications in order to prevent potentially malicious links based on the defined policy. The proposed method aims to detect and prevent application-level privilege escalation attacks at the app's runtime. Also, in [151] the Android OS has been extended by adding a flexible privacy enforcement framework which is transparent to the applications. To achieve this goal, a part of Android framework, core libraries, and a number of services and managers outside the application VM have been modified. The developed framework is called YAASE which is an Android security extension that supports fine-grained access control policies. YAASE has used the TaintDroid taint analysis mechanism to enforce security decisions on data distribution whether inside the device (from one application to another) or outside the device (via internet connections). Also, in [152], an extended Android platform called Saint was developed to address the limitations of Android security by adding installation-time granting policies and inter-application communication (IPC) policies (i.e. run-time enforcement policies). The android installer has been extended to extract the required permissions from the manifest file and mapping the permissions to the installation time policies' database to make a decision whether the installation process will continue or not. On the other hand, Saint's run-time enforcement policies cover four critical component interactions, i.e. starting new activities, binding components to services, receiving broadcast Intents and accessing content providers. Furthermore, the run-time policy rules specify multiple conditions that should be verified for IPC proceeding.

4.4 Code packer/unpacker tools

The packing techniques are used by the applications' developers to protect their applications from tampering and reverse engineering, to this end, the tools that called packers are used. In these techniques, a combination of previously mentioned methods such as obfuscation, reflection, native code, and dynamic payload are used to hide the app's source code in order to prevent code retrieval and re-use. Unfortunately, the developers of malicious applications exploit these techniques to impede analysis of applications or to make it more difficult. Some of the studied works aim to propose and test some camouflage techniques to test the robustness of anti-malware systems, we have been classified this type of works as code packers. For example, in [131], a framework called ADAM that can generate multiple malware samples from one sample using repackaging and obfuscation techniques was developed to evaluate the robustness of anti-virus systems against malware mutation. In [153], a systematic framework called DroidChameleon has been developed for testing the robustness of some commercial anti-malware against transformation and obfuscation techniques. Also, in [134], a collection of obfuscation techniques was used in order to obfuscate malware samples that used for testing the robustness of some anti-malware engines. In [154], a framework has been developed to test the resilience of repackaging detection systems against obfuscation techniques.

Some of other studied works aim to develop tools for unpacking the applications which use the code packing techniques and retrieving the app's original source code, these works have been classified as code unpackers in this paper. For example, in [155], a packed DEX files recovering tool called PackerGrind has been proposed and developed based on a novel iterative method. PackerGrind can monitor the packed patterns effectively for extracting Dex files based on its capability of conducting cross-layer profiling in real smartphones. The tool was tested using some real packed apps and it has been stated that it is efficient in retrieving the original DEX files from the packed apps. In [156], Android code packing techniques have been described and classified to dex protection techniques, native protection techniques, memory protection techniques and code release protection techniques. Also, an automated unpacking system called AppSpear has been proposed. The proposed framework uses a new approach based on ByteCode decryption and Dex reassembly that can take the place of traditional manual analytics and memory dump-based unpacking techniques. It has been stated that the proposed method supports each of Dalvik and ART and resists the packers' techniques with low overhead.

4.5 User interface triggering tools

As mentioned previously, the problem of interaction with the application and simulating the app's normal usage is one of the most important challenge of the dynamic analysing methods. Since most of the used UI triggering tools depend on random event generation, some of the execution paths that can give important information about the actual behaviour of the application can be missed. So, the user interface trigger tools development is one of the most important research trends of the previous works. For example, in [157], AndroidRipper, a GUI ripping-based Android application's automatic user interface event trigger tool has been presented. The proposed method has been tested in term of finding real bugs and its suitability for testing processes that need to be carried out in a short amount of time. It was stated that the proposed technique is more effective in bug detection than the random testing technique implemented by MonkeyRunner tool. In [158], a system called Dynodroid that generates user interface inputs for Android applications has been presented. A new observe-select-run principle has been used to produce the sequence of such inputs efficiently. The proposed tool operates on unmodified app's binary files and generates UI inputs and system inputs with an ability to combining inputs from user and machine. The performance of the proposed tool has been compared with the performance of manual tests performed by expert users and the performance of MonkeyRunner tool. It has been stated that Dynodroid can produce much shorter input sequences than MonkeyRunner tool. In [98], An UI-identification automatic trigger tool that can interact with mobile applications in a meaningful order and monitoring apps' behaviours has been implemented. The proposed tool has been tested by building a decision model based on a variety of machine learning algorithms and the obtained results have been compared with the results of some other tools. In [116], a dynamic analysis tool named Component Traversal has been proposed in order to automatically execute the code of each given Android application as completely as possible. It has been stated that the proposed Component Traversal tool outperform the MonkeyRunner tool in term of system calls extraction.

Figure 11 shows the distribution of the studied works according to the proposed taxonomy.

Table 3 illustrates the studied works information in accordance with the proposed taxonomy.

5 The proposed Schematic Review Model

In this work, we have proposed a schematic model in light of the studied works and the proposed taxonomy. The proposed model which we called "Schematic Review Model" represents a complete description of the malware

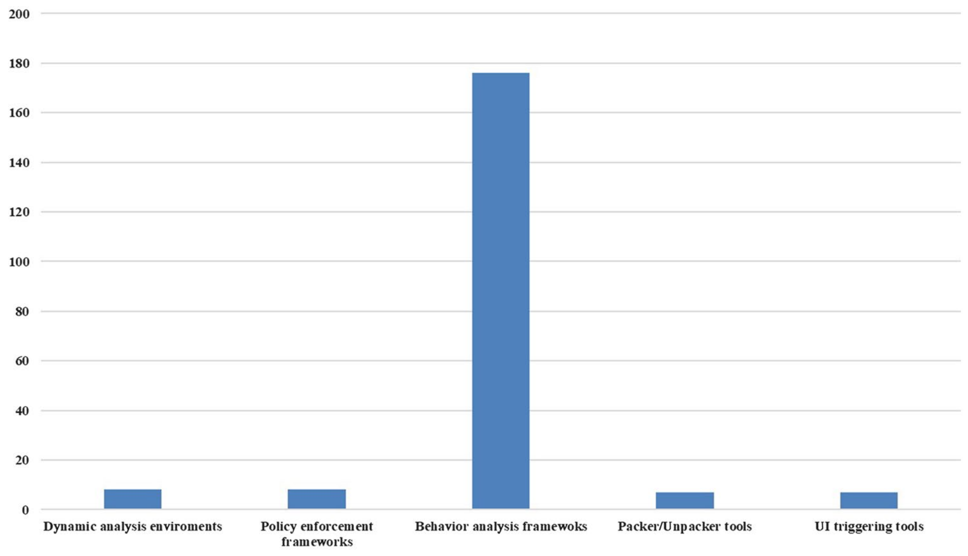

Fig. 11 The distribution of the studied works according to the proposed taxonomy

Table 3 The works that covered in this paper in accordance with the proposed taxonomy

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Countering dynamic challenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
1	SaaS [160]	2019	B_analysis	Vis&staticbased	Mani_F, Cod_F, Image_features	TF-IDF	ML	✓	✗	✗	✗		
2	Saint et al. [30]	2019	B_analysis	Vis-based	Cod_F, Image_Features	Deep_L	Deep_L	✗	✗	✗	✗		
3	Zhang et al. [161]	2019	B_analysis	Static	Mani_F, Cod_F	–	ML	✗	✗	✗	✗		
4	NDroid [161]	2019	B_analysis	Dynamic	System calles, API calls	–	–	✗	✓	✓	✗	✗	✗
5	CloneSpot [162]	2019	B_analysis	Static	Meta_F	–	Ptrn_M	✓	✗	✗	✗		
6	DroidDet [33]	2018	B_analysis	Static	Mani_F, cod_F	TF-IDF, Cos_Sim	ML	✗	✗	✗	✗		
7	Gurulian.et al. [32]	2016	B_analysis	Static	Meta_F	–	Sign_sim	✓	✗	✗	✗		
8	KUAFUDET [48]	2018	B_analysis	Static	Cod_F, Semtc_F	Manual pruning	ML+Sim	✓	✗	✗	✗		
9	Mad4a [61]	2018	B_analysis	Hybrid	Mani_F, Net_beh	–	Ptrn_M					✗	✗
10	SAFEDroid [77]	2018	B_analysis	Static	Cod_F, Mani_F, Meta_F	–	ML	✗	✗	✓	✓		
11	Wang et al. [86]	2018	B_analysis	Static	Mani_F, Semtc_F	–	Ptrn_M	✗	✗	✗	✗		
12	Wang et al. [49]	2018	B_analysis	Static	Cod_F, Mani_F	–	Deep_L	✗	✗	✗	✗		
13	DroidFusion [79]	2018	B_analysis	Static	Cod_F, Mani_F	IG	ML	✗	✗	✓	✗		
14	Kirubavathi et al. [50]	2017	B_analysis	Static	Mani_F	IG	ML	✗	✗	✗	✗		
15	MalPat [51]	2018	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
16	FalDroid [163]	2018	B_analysis	Static	Semtc_F	TF-IDF	ML	✗	✗	✗	✗		
17	AppSpear [156]	2018	Packer/unpacker	Dynamic	–	–	ML	✓	✓	✓	✓	✓	✗
18	Papadopoulos et al. [164]	2018	B_analysis	Dynamic	R_Con	–	ML					✗	✗
19	Jha et al. [165]	2018	B_analysis	static	Mani_F	–	–	✗	✗	✗	✗		
20	SIGPID [166]	2018	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
21	Chunlei et al. [167]	2018	B_analysis	Static	Cod_F	MI	ML	✗	✗	✗	✗		
22	Özkan et al. [168]	2018	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
23	Yangxu et al. [169]	2018	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
24	Chihiro et al. [170]	2018	B_analysis	Static	Cod_F, Mani_F	Deep_L	Deep_L	✗	✗	✗	✗		
25	RanDroid [171]	2018	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✓	✓	✓		
26	ONAMD [172]	2018	B_analysis	Static	Cod_F, Mani_F, Semtc_F		Sign, ML	✗	✗	✗	✗		
27	Jaemin et al. [173]	2018	B_analysis	Static	Cod_F	–	ML	✗	✗	✗	✗		
28	Somarriba et al. [52]	2017	B_analysis	Dynamic	Net_Beh	–	Ptrn_M					✗	✗
29	AspectDroid [57]	2018	B_analysis	Hybrid	Semtc+Dyn_Beh	–	Ptrn_M	✗	✓	✓	✗	✓	✗
30	SAMADroid [174]	2018	B_analysis	Hybrid	Mani_F, cod_F, System calles		ML	✗	✗	✗	✗	✗	✗

Table 3 (continued)

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Countering dynamic challenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
31	NTPDroid [175]	2018	B_analysis	Hybrid	Mani_F, Net_beh	–	Ptrn_M	✗	✗	✗	✗	✗	✗
32	Alzaylaee [145]	2017	UI trigger	Dynamic	API signatures	–	–					✓	✗
33	MOCDroid [74]	2016	B_analysis	Static	Cod_F	GeneticAlgorithm	ML+Sign	✗	✗	✗	✗		
34	AndroDialysis [70]	2017	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
35	Wang et al. [121]	2017	B_analysis	Static	Cod_F, Mani_F, Meta_F	MI,Chi2,ANOVA	ML	✗	✓	✓	✓		
36	Pındroid [128]	2017	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
37	DroidNative [85]	2017	B_analysis	Static	Semtc_F	–	Sign	✗	✓	✗	✗		
38	Sokolova et al. [92]	2017	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
39	Du et al. [112]	2017	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
40	Palumbo et al. [83]	2017	B_analysis	Static	Cod_F, Meta_F	Threshold based	ML	✗	✗	✗	✗		
41	Yang et al. [106]	2017	B_analysis	Static	Meta_F	IG	Sign	✗	✗	✗	✗		
42	Tong and Yan [118]	2017	B_analysis	Hybrid	Semtc, Dyn_Beh	–	Ptrn_M	✗	✓	✓	✗	✗	✗
43	Milosevic et al. [78]	2017	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
44	MalDozer [115]	2017	B_analysis	Static	Cod_F	Deep_L	Deep_L	✗	✗	✗	✗		
45	Buchanan et al. [24]	2017	B_analysis	–	–	–	–						
46	Rehman et al. [176]	2017	B_analysis	Hybrid	Mani_F, Cod_F, Dyn_Beh	–	ML	✗	✗	✗	✗	✗	✗
47	Alzaylaee et al. [55]	2017	B_analysis	Dynamic	API calls and Intents	IG	ML					✓	✓
48	Wang et al. [97]	2017	B_analysis	Dynamic	System calles	–	ML					✗	✗
49	PackerGrind [155]	2017	Packer/unpacker	Hybrid	–	–	–	✗	✓	✓	✓	✗	✗
50	Martinelli et al. [177]	2017	B_analysis	Dynamic	System calles	Deep_L	Deep_L					✗	✗
51	Liang et al. [178]	2017	B_analysis	Dynamic	System calles	Deep_L	Deep_L					✗	✗
52	Su et al. [179]	2017		Hybrid	Cod_F, Mani_F, Dyn_Beh	–	ML	✗	✗	✗	✗	✗	✗
53	CASANDRA [180]	2017	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
54	FgDetector [181]	2017	B_analysis	Static	Cod_F, Mani_F	PCA algorithm	ML	✓	✗	✗	✓		
55	Mohsen et al. [182]	2017	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
56	DeepFlow [183]	2017	B_analysis	Static	Semtc_F	Deep_L	Deep_L	✗	✗	✗	✗		
57	Nix et al. [117]	2017	B_analysis	Static	Cod_F	Deep_L	Deep_L	✗	✗	✗	✓		
58	DAPASA [184]	2017	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
59	R2-D2 [26]	2017	B_analysis	Vis-based	Cod_F, Image_Features	Deep_L	Deep_L	✗	✗	✗	✗		
60	Yang et al. [27]	2017	B_analysis	Vis-based	Image_Features	–	ML	✗	✗	✗	✗		
61	Karimi et al. [28]	2017	B_analysis	Vis-based	Semtc_F, Image_Features	–	Sign	✗	✗	✗	✗		

Table 3 (continued)

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Coun-tering dynamic chal-lenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
62	Dexteroid [89]	2016	B_analysis	Static	Cod_F	–	Ptrn_M	✗	✗	✗	✗		
63	Brown et al. [88]	2016	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
64	Ma et al. [111]	2016	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
65	ANASTASIA [110]	2016	B_analysis	Static	Cod_F, Mani_F	Trees-Classifier	Deep_L	✗	✗	✗	✗		
66	Andro-Dumpsys [64]	2016	B_analysis	Hybrid	Cod_F, Mani_F, Meta_F, Dyn_Beh	–	Ptrn_M	✗	✓	✓	✓	✗	✗
67	Alba et al. [123]	2016	B_analysis	Static	Cod_F, Mani_F	Chi2,Relief	ML	✗	✗	✗	✗		
68	SafeDroid [185]	2016	B_analysis	Static	Cod_F	–	ML	✗	✗	✗	✗		
69	ASE [186]	2016	B_analysis	Static	Mani_F, Meta_F	–	Ptrn_M	✗	✗	✗	✗		
70	Kumar et al. [100]	2016	B_analysis	Vis-based	GIST image feature	–	ML	✗	✗	✗	✗		
71	Mamadroid [129]	2016	B_analysis	Static	Semtc_F	PCA	ML	✗	✗	✗	✗		
72	Chang et al. [98]	2016	UI trigger	Hybrid	Mani_F, Dyn_Beh	IG		✗	✗	✓	✓	✗	✗
73	Verma and Muttoo [114]	2016	B_analysis	Static	Meta_F	Information gain	ML	✗	✗	✗	✗		
74	Wu et al. [126]	2016	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗	✗	✗
75	Deep4MalDroid [116]	2016	UI trigger	Dynamic	System calls	–	Deep_L	✗	✗	✗	✗		
76	DroidOL [187]	2016	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
77	DroidDetector [62]	2016	B_analysis	Hybrid	Mani_F, Code_F, Dyn_Beh	–	Deep_L	✗	✗	✓	✓	✗	✗
78	Dynalog [147]	2016	D_Env	Dynamic	API calls, other features	–	–					✗	✓
79	StormDroid [63]	2016	B_analysis	Hybrid	Mani_F, Code_F, Dyn_Beh	–	ML	✗	✗	✓	✗	✗	✗
80	Chen et al. [188]	2016	B_analysis	Static	Code_F, Semtc_F	TF-IDF	ML	✗	✗	✓	✗		
81	Ju et al. [189]	2016	B_analysis	Static	Mani_F	–	Ptrn_M	✗	✗	✗	✗		
82	Mmda [190]	2016	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
83	DroidDeepLearner [191]	2016	B_analysis	Static	Cod_F, Mani_F	Deep_L	Deep_L	✗	✗	✗	✗		
84	Xiaotian et al. [192]	2016	B_analysis	Static	Cod_F, Mani_F	Markov blanket discovery algorithms	ML	✗	✗	✗	✗		
85	Manzhi et al. [193]	2016	B_analysis	Static	Cod_F, Mani_F		ML	✗	✗	✗	✓		
86	Alejandro et al. [194]	2016	B_analysis	Static	Mani_F, Semtc_F	–	ML	✗	✗	✗	✗		
87	DroidDeep [195]	2016	B_analysis	Static	Cod_F, Mani_F	Deep Belief Network (DBN)	Deep_L	✗	✗	✗	✗		
88	DroidChain [196]	2016	B_analysis	Static	Semtc_F	–	Sing	✗	✗	✗	✗		
89	ROAR [197]	2016	B_analysis	Static	Mani_F, Semtc_F	–	Sing	✗	✓	✗	✗		
90	Xiaotian et al. [198]	2016	B_analysis	Static	Mani_F, Semtc_F, Cod_F	–	ML	✗	✗	✗	✓		
91	Salvador et al. [199]	2016	B_analysis	Static	Mani_F	IG, Relief, Chi^2	ML	✗	✗	✗	✗		

Table 3 (continued)

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Countering dynamic challenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
92	Wei et al. [53]	2015	B_analysis	Dynamic	Net_Beh	–	ML					✗	✗
93	Elish et al. [87]	2015	B_analysis	Static	Semtc_F	–	Ptrn_M	✗	✗	✗	✗		
94	APK Auditor [67]	2015	B_analysis	Static	Mani_F	–	Sign	✗	✗	✗	✗		
95	Kurniawan et al. [54]	2015	B_analysis	Dynamic	R_Con, Net_Beh	–	ML					✗	✗
96	Yerima et al. [119]	2015	B_analysis	Static	Cod_F, Mani_F	MI	ML	✗	✗	✗	✗		
97	Canfora et al. [91]	2015	B_analysis	Static	Semtc_F	MI	ML	✗	✗	✗	✗		
98	Lantz et al. [138]	2015	B_analysis	Static	Cod_F	–	Ptrn_M	✗	✓	✗	✗		
99	Mobile-Sandbox [127]	2015	B_analysis	Hybrid	Mani_F, Dyn_Beh	–	ML	✗	✓	✓	✓	✗	✗
100	Li et al. [200]	2015	B_analysis	Static	Cod_F	–	Ptrn_M	✗	✗	✗	✗		
101	Rosmansyah and Dabarsyah [201]	2015	B_analysis	Static	Cod_F	–	ML	✗	✗	✗	✗		
102	Li et al. [202]	2015	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
103	DroidSafe [203]	2015	B_analysis	Static	Cod_F	–	–	✗	✓	✗	✗		
104	Sheen et al. [109]	2015	B_analysis	Static	Cod_F	IG, Relief, Chi^2	ML	✗	✗	✗	✗		
105	Lee et al. [82]	2015	B_analysis	Static	Semtc_F, Cod_F	–	Sign	✗	✗	✗	✗		
106	M0Droid [204]	2015	B_analysis	Dynamic	System calles	–	Sign					✗	✗
107	Bushra et al. [205]	2015	B_analysis	Dynamic	Dyn_Beh	–	ML					✗	✗
108	MARVIN [206]	2015	B_analysis	Hybrid	Cod_F, Mani_F, Net_Beh, File_Acc	Fisher Score	ML	✗	✓	✓	✓	✗	✗
109	Ashutosh et al. [29]	2015	Packer/unpacker	Vis-based	–	–	–	✗	✗	✗	✓		
110	Maiorca et al. [134]	2015	Packer/unpacker	Static	–	–	–	✓	✓	✓	✓		
111	Andlantis [207]	2014	D_Env	Dynamic	Net_Beh	–	–	✗	✗	✓	✗	✗	✗
112	Adebayo et al. [208]	2014	B_analysis	Static	Cod_F	Apriori-PSO	ML	✗	✗	✗	✗		
113	TaintDroid [59]	2014	B_analysis	Dynamic	Semtc_F	–	–					✗	✗
114	DroidSentinel [209]	2014	B_analysis	Dynamic	–	–	Sign					✗	✗
115	Li et al. [148]	2014	B_analysis	Dynamic	Net_Beh	–	ML					✗	✗
116	Mdoctor [210]	2014	B_analysis	Static	Mani_F, Meta_F		Sign	✗	✗	✗	✗		
117	Merlo et al. [211]	2014	B_analysis	Dynamic	R_Con	–	–					✗	✗
118	DroidTrace [142]	2014	B_analysis	Hybrid	Mani_F, Dyn_beh	–	Ptrn_M	✗	✗	✓	✓	✗	✗
119	Dendroıd [212]	2014	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✗		
120	Moonsamy et al. [94]	2014	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
121	PasDroid [213]	2014	B_analysis	Dynamic	Semtc_F	–	–					✗	✗

Table 3 (continued)

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Coun-tering dynamic chal-lenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
122	DroidSIFT [71]	2014	B_analysis	Static	Semtc_F, Cod_F	–	ML	✗	✗	✗	✗		
123	Drebin [69]	2014	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
125	Yerima et al. [214]	2014	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
126	Ng et al. [125]	2014	B_analysis	Dynamic	System calls	–	Ptrn_M					✗	✗
127	Manilyzer [108]	2014	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
128	AMDetector [65]	2014	B_analysis	Hybrid	Mani_F, Cod_F, Dyn_Beh	–	Ptrn_M	✗	✗	✓	✗	✗	✗
129	Apposcopy [215]	2014	B_analysis	Static	Semtc_F	–	Sign	✗	✗	✗	✗		
130	Xiaoyan et al. [216]	2014	B_analysis	Static	Mani_F	PCA algorithm	ML	✗	✗	✗	✗		
131	Xiangyu et al. [217]	2014	B_analysis	Static	Mani_F	IG, Fisher Score, Chi^2	ML	✗	✗	✗	✗		
132	Droid Detective [218]	2014	B_analysis	Static	Mani_F	–	Ptrn_M	✗	✗	✗	✗		
133	MIGDroid [143]	2014	B_analysis	Static	Cod_F	–	G_Sim	✓	✗	✗	✗		
134	Idrees et al. [219]	2014	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
135	Raphael et al. [220]	2014	B_analysis	Static	Cod_F, Mani_F	X–ANOVA, X–Utest	G_Sim	✗	✗	✗	✗		
136	Park et al. [81]	2014	B_analysis	Static	Cod_F, Mani_F	–	G_Sim	✗	✗	✗	✗		
137	DroidGraph [84]	2014	B_analysis	Static	Cod_F	–	Sign	✗	✗	✗	✗		
138	Moghaddam et al. [72]	2014	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
139	Shen et al. [102]	2014	B_analysis	Static	Cod_F, Mani_F	Heuristic based	G_Sim	✗	✗	✗	✗		
140	Britton et al. [221]	2014	B_analysis	Static	Mani_F, Semtc_F	–	ML	✗	✗	✗	✗		
141	DroidAnalyzer [222]	2014	B_analysis	Static	Cod_F, Mani_F	–	Sign	✗	✗	✗	✗		
142	Deepa et al. [223]	2014	B_analysis	Static	Cod_F	GK, IG &CFS	ML	✗	✗	✗	✗		
143	Shabtai et al. [224]	2014	B_analysis	Dynamic	Net_Beh	–	ML	✓	✗	✓	✗		
144	Yerima et al. [225]	2013	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✓	✓		
145	AppsPlayground [58]	2013	D_Env	Dynamic	–	–	–					✗	✗
146	Mobile-Sandbox [139]	2013	B_analysis	Hybrid	Mani_F, Dyn_Beh	–	–	✗	✓	✗	✓	✗	✗
147	DroidChameleon [137]	2013	Packer/unpacker	Static	–	–	–					✗	✗
148	Dynodroid [158]	2013	UI trigger	Dynamic	–	–	–					✗	✗
149	Chekina et al. [226]	2013	B_analysis	Dynamic	Net_Beh	–	–					✗	✗
150	Karami et al. [227]	2013	UI trigger	Dynamic	Net_Beh, File_Acc	–	–					✓	✗
151	Stream [99]	2013	D_Env	Dynamic	R_Con, Net_Beh		ML					✗	✗
152	Mas'ud et al. [66]	2013	B_analysis	Hybrid	Mani_F, Cod_F, Sys_call, Net_Beh	–	–	✗	✗	✗	✗	✗	✗
153	AEMF [228]	2013	B_analysis	Dynamic	Semtc_F	–	Ptrn_M					✗	✗

Table 3 (continued)

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Countering dynamic challenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
154	DroidAPIMiner [75]	2013	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✓	✓	✗		
155	Samra et al. [68]	2013	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
156	AppGuard [229]	2013	Policy_Enf	Static	–	–	Policy	✗	✗	✗	✗		
157	A3 [80]	2013	B_analysis	Static	Semtc_F	–	Ptrn_M	✗	✗	✗	✗		
158	Aung et al. [113]	2013	B_analysis	Static	Mani_F	IG	ML	✗	✗	✗	✗		
159	WHYPER [90]	2013	B_analysis	Static	Mani_F	–	–	✗	✗	✗	✗		
160	DroidOLytics [103]	2015	B_analysis	Static	Semtc_F	Similarity digest hashing	Sign	✓	✗	✗	✗		
161	Peiravian et al. [230]	2013	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
162	Glodek et al. [140]	2013	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✓	✗	✗		
163	Lu et al. [231]	2013	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
164	AnDarwin [105]	2013	B_analysis	Static	Semtc_F	–	Ptrn_M	✓	✗	✗	✗		
165	Yu et al. [232]	2013	B_analysis	Dynamic	System calles	–	ML					✗	✗
166	Alam et al. [233]	2013	B_analysis	Dynamic	Combination	–	ML					✗	✗
167	Ham et al. [234]	2013	B_analysis	Dynamic	R_Con	–	ML					✗	✗
168	VetDroid [235]	2013	B_analysis	Dynamic	–	–	–					✗	✗
169	CopperDroid [149]	2013	B_analysis	Dynamic	System calls	–	–	✗	✓	✗	✗	✓	✗
170	DroidAnalytics [236]	2013	B_analysis	Hybrid	Mani_F, API calls	–	Sign	✓	✗	✓	✓	✗	✗
171	ANANAS [237]	2013	B_analysis	Dynamic	Net_Beh, System calles	–	–	✗	✓	✓	✗		
172	Puma [238]	2013	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
173	Huang et al. [154]	2013	Packer/unpacker	Static	–	–	–						
174	DroidLogger [56]	2012	B_analysis	Hybrid	Cod_F, Dyn_Beh	–	–	✗	✗	✗	✓	✗	✗
175	SmartDroid [239]	2012	UI trigger	Hybrid	Semtc_F, Dyn_Beh	–	–	✗	✗	✗	✗	✗	✗
176	DroidRanger [141]	2012	UI trigger	Hybrid	Mani_F, Dyn_B	–	Sign	✗	✓	✓	✗	✗	✗
177	ProfileDroid [240]	2012	D_Env	Hybrid	Mani_F, Dyn_B	–	–	✗	✗	✗	✗	✗	✗
178	DroidScope [60]	2012	D_Env	Dynamic	Semtc_F	–	–					✗	✗
179	AndroidRipper [157]	2012	UI trigger	Dynamic	–	–	–					✓	✗
180	AndroidLeaks [93]	2012	B_analysis	Static	Cod_F, Mani_F	–	Sign	✗	✗	✗	✗		
181	Sanz et al. [95]	2012	B_analysis	Static	Cod_F, Mani_F, Meta_F	IG	ML	✗	✗	✗	✗		
182	DroidMat [93]	2012	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
183	DroidMOSS [144]	2012	B_analysis	Static	Cod_F, Meta_F	Fuzzy hashing	Sign	✓	✗	✗	✗		
184	RiskRanker [104]	2012	B_analysis	Static	Semtc_F	–	Sign	✗	✓	✗	✓		

Table 3 (continued)

#	Tool name	Year	Goal	Method	Used features	Feature selection	Detection method	Countering static challenges				Countering dynamic challenges	
								R	*N*	*D*	*O*	*X_P*	*Em*
185	Wei et al. [241]	2012	B_analysis	Static	Mani_F	–	–	✗	✗	✗	✗		
186	MADAM [242]	2012	B_analysis	Dynamic	System calles & other Dyn_Beh	–	ML					✗	✗
187	Sahs et al. [243]	2012	B_analysis	Static	Mani_F, Semtc_F	–	ML	✗	✗	✗	✗		
188	Sanz et al. [95]	2012	B_analysis	Static	Cod_F, Mani_F, Meta_F	IG	ML	✗	✗	✗	✗		
189	Yang et al. [244]	2012	B_analysis	Static	Semtc_F	–	–	✗	✗	✗	✗		
190	Gascon et al. [245]	2012	B_analysis	Static	Semtc_F	–	ML	✗	✗	✗	✓		
191	ADAM [131]	2012	Packer/unpacker	Static	–	–	–						
192	YAASE [151]	2011	Policy_Enf	Dynamic	–	–	Policy					✗	✗
193	XmanDroid [150]	2011	Policy_Enf	Dynamic	–	–	Policy					✗	✗
194	Saint [152]	2012	Policy_Enf	Dynamic	–	–	Policy					✗	✗
195	Su et al. [246]	2011	B_analysis	Dynamic	Net_Beh, System calles	–	ML					✗	✗
196	QUIRE (2011 dynamic 4)	2011	D_Env	Dynamic	–	–	Policy					✗	✗
197	Stowaway [76]	2011	B_analysis	Static	Cod_F, Mani_F	–	Ptrn_M	✗	✗	✗	✗		
198	Crowdroid [247]	2011	B_analysis	Dynamic	System calls	–	ML					✗	✗
199	ComDroid [248]	2011	B_analysis	Static	Cod_F, Mani_F	–	–	✗	✗	✗	✗		
200	Isohara [249]	2011	B_analysis	Dynamic	System calles	–	Sign					✗	✗
201	Apex [250]	2010	Policy_Enf	Dynamic	–	–	Policy					✗	✗
202	Porscha [251]	2010	Policy_Enf	Dynamic	–	–	Policy					✗	✗
203	CrePE [252]	2010	Policy_Enf	Dynamic	–	–	Policy					✗	✗
204	Paranoid [253]	2010	B_analysis	Dynamic	Multiple Dyn_Beh features	–	–					✗	✗
205	Barrera et al. [254]	2010	B_analysis	Static	Mani_F	–	ML	✗	✗	✗	✗		
206	AASandbox [255]	2010	D_Env	Hybrid	Cod_F, Mani_F, System calles	–	–	✗	✓	✗	✓	✗	✗
207	Shabtai et al. [256]	2010	B_analysis	Static	Cod_F, Mani_F	–	ML	✗	✗	✗	✗		
208	Kirin [257]	2009	Policy_Enf	Static	Mani_F	–	Policy	✗	✗	✗	✗		

Cod_F, code based features; Semtc_F, semantic features; Mani_F, manifest features; Meta_F, app metadata features; R_Con, resource consumption; Net_Beh, network behavior; Dyn_Beh, dynmic Behavior; R, repackaging; N, native code; D, dynamic payload; O, obfuscation techniques; X_P, coverage of all execution paths; Em, anti-emulator techniques; ML, machine learning; Sign, signature based; Ptrn_M, pattern's matching; Polc&Rul, policy & rules based; G_Sim, graph similarity; Vis-based, visualisation-based; Rules, rules-based; Policy, policy-based; Deep_L, deep learning; MI, mutual information; IG, information gains; Cos_Sim, cosine similarity; CFS, correlation feature selection; GK, Goodman Kruskals analysing process in a way that enables the researcher to take a comprehensive look to the domain without much effort. We believe that the existence of such models is very important in reviews papers in all domains to give an abstracted description of the most research trends in the domain using one model. The process of malware analysing has been described in multiple phases, these phases were described detailly in a schematic manner, as showed in Fig. 12. To the best of our knowledge, it is the first time that this process is described in this way with these much of details. The most techniques and methods that used in the studied works have been overviewed and organized under multiple phases. These phases have been discussed in detail in the 'Used techniques phases-based taxonomy' section.

6 Decision and future works areas

Although there are many reviews that have been conducted in order to highlight the works that achieved in Android malware analysing domain, there is no comprehensive taxonomy for all research trends in this domain. Furthermore, none of the existing review papers contains a schematic model that makes it easy for the reader to know the methods and methodologies used in a particular field of research without much effort. This paper aims at proposing a comprehensive taxonomy and suggesting a detailed schematic review approach. To this end, a large number of works that published within a period of time that almost starting from the date of the emergence of the first malicious applications targeting the Android system

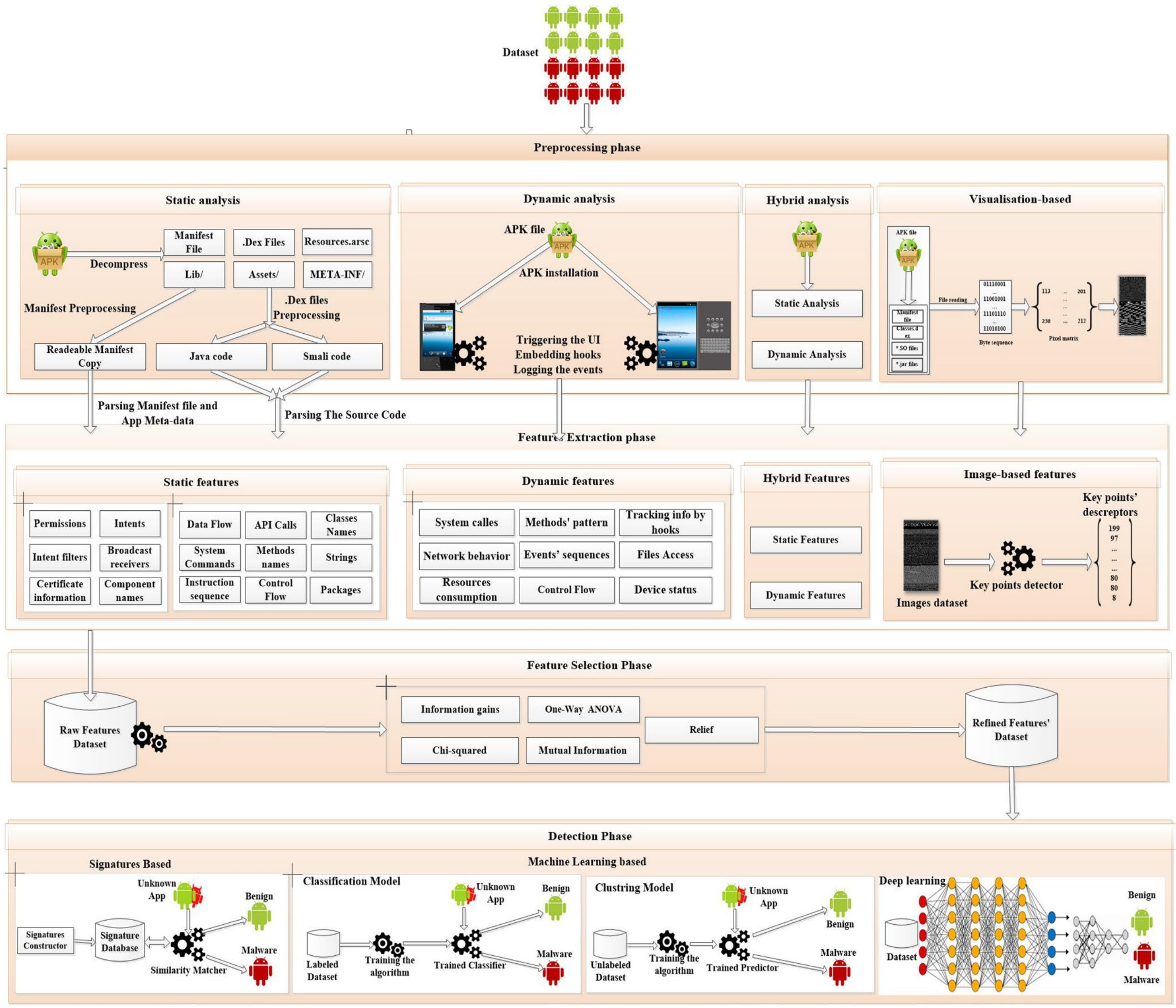

Fig. 12 The proposed Android malware detection methodologies schematic model

until the present time have been studied. Also, a comprehensive taxonomy has been suggested so that including as most as possible research trends in this domain. Moreover, a novel detailed schematic model called Schematic Review has been developed. It has been observed that most of the studied works use static analysis method. Also, it has been noted that most of the previously conducted works do not address the analysis evasions techniques such as Native code usage, Dynamic code loading, repackaging, or code encryption. In addition, we have observed that most of the works that looked comprehensive to some extent face problems such as increasing complexity and computational time or it is un-automated frameworks. We also noted that the malware-visualization based analysis method has been used in a very small number of the studied works although of its success in desktop malware detection domain. Moreover, almost all static works have been done based on the bytecode level, only in two studies, the application was analysed based on the Native code level, and no study has analysed the two level of code. Thus, the common weak point of all designed static analysis-based approaches is the analysis of the Native code. It has been noted that semantic features were not used extensively in the studied works. In addition, the features' engineering and selection methods were used only in 39 studies. Furthermore, deep learning techniques have been tested only in 14 studies. In terms of the used dataset, we noted that most of the developed frameworks have been evaluated using a benign dataset that downloaded from the official market, and the well-known malicious datasets like Drebin, Malgenom as a malware dataset. In a small part of works, a mix of apps that downloaded from the official market and the third-party markets have been used as a benign dataset. And a mix of the well-known malicious datasets and some samples that collected from the internet have been used as a malicious dataset a in small number of works. In addition, the dynamic analysis drawbacks have not been addressed in most of the works which use this analysing method (whether dynamic or hybrid analysis frameworks). Also, most of the proposed dynamic frameworks suffer from increased overhead and computational complexity. Furthermore, most of the dynamic analysis frameworks use random-based events generation tools for interacting with the tested program (such as MonkeyRunner), so some app's execution paths can be missed. Thus, there is an urgent need for deeper and more comprehensive analysis methods such that all malware developers' camouflage technologies such as obfuscation, dynamic loading, native code…etc. can be addressed. Moreover, the proposed tools should maintain the performance at an acceptable level and the needed user intervention should be as low as possible. Therefore, we suggest constructing the app markets' future security tools based on multi-levels analysis frameworks. In other words, the applications should be filtered according to their severity level so that a small number of applications reach the stages of analysis that need a great analysing cost. Thus, a signature-based or heuristic-based method can be used in the first level, to this end, a lightweight signatures database should be constructed, and the apps are matched with it. In case the application is not matched any signature, it will be transferred to the second level in which a lightweight static analysis-based method can be used. If the application can be judged to be benign or malicious with no doubt the analysis will be ended. On the other hand, if the app has a suspicious behaviour or in case that the app uses obfuscation, dynamic content loading techniques…etc. it will be transferred to the third analysing level, which we propose to be a dynamic analysis method. In the third level, the app will be executed in an analysing environment and its behaviour will be studied by extracting as many as possible dynamic features. If a decision cannot be taken the extracted static and dynamic features can be analysed in a hybridized manner. Moreover, it is possible to add another level to this model, so that if the system is unable to judge on the app's behaviour clearly, the reports can be studied by the analyst manually to give a final decision. Also, the signature database should be updated according to the decisions that taken in the analysing levels. Using this method, the applications are filtered so that only a few applications will reach to the final stage. Consequently, just a few applications will take a lot of analysis time, thus it will significantly reduce the overall overhead comparing with performing hybrid or dynamic analysis on all applications which will be analysed. More importantly, the App Store will have a high level of protection.

Through the extensive study carried out in this paper, some points which should be focused in future works were identified.

6.1 Static analysis

As noted previously, the static analysis methods face many challenges. Although some of the static analysis's drawbacks have been addressed to some extent in a number of previous works, there is still a need for more focus on finding stronger solutions that avoid previous solutions' weaknesses. Therefore, it is necessary to focus on the following areas in the future works.

6.1.1 Native code and bytecode analysis

The most of previous works are limited to analysis at bytecode level and a few of studies have addressed the analysis at the native code level, and almost there is no framework that works at both bytecode and native code. For example, Alam et al. [85] focus on analysing apps at the native code level and when needed to analyse a bytecode, it must be converted into native code and extracting native code-based features. However, this system suffers from some weaknesses such as the inability to detect zero-day malware and it is limited to detect previously known malware. Therefore, it should be focused on building frameworks that extract features from the two code levels, in order to obtain a deeper analysis for the applications and closing the door in front of embedding the malicious code within the native code libraries.

6.1.2 Tackle obfuscation techniques

As mentioned before, obfuscation techniques are one of the biggest static analysis techniques' challenges, and it has been dealt with very limited in previous studies. Thus, there is an urgent need to find more robust solutions based on more semantic features to counter obfuscation techniques such as reflection, control flow obfuscation, Junk code insertion…etc.

6.2 Dynamic analysis

Although the dynamic analysis is the solution used to address static analysis method's drawbacks, there are some effective techniques used by developers of malicious applications to defeat this analysing method. For example, in 2017, a number of malicious applications that exceeded the protection of the android official market have been discovered by McAfee's staff and has been downloaded by a large number of users [9]. So, we have suggested some trends that can be focused on it in future works.

6.2.1 User interface triggers

The problem of covering the entire app's code and scanning all possible app's execution paths is one of the most important problems of dynamic analysing methods [33], because of that the most of the used UI trigger tools generates random events, so the app's source code will not be fully covered. Also, the malware's developers can depend on the generated UI events' frequency to discover whether the application is executed in an analysing environment in order to stop the malicious content's execution and executing a fake benign code. Therefore, semantic-based robust tools should be developed to detect events that should be triggered so that all the app's source code can be covered, and the user interfaces can be triggered using well-defined events.

6.2.2 Counter anti-emulator technologies

These techniques are used to detect whether the application is being executed in an analysing environment and thereby hide its malicious behaviour [121, 146]. In some previous works such as [159], some steps have been taken to mitigate the effects of these techniques, but there is an urgent need to develop countermeasure techniques that can overcome these technologies and detect malicious applications that using it.

6.2.3 Time complexity

one of important challenge that facing dynamic analysis methods is how to reduce the time that required for executing the application, collecting the features and making the right decision. In addition to the difficulty of implementing this procedure in a fully automated manner. Therefore, it is useful to focus on this aspect in future works.

Furthermore, comprehensive datasets should be used so that it should include a collection of benign applications from both official and third-party markets and a mix of well-known malicious datasets' samples and as much as possible of new daily published malware samples.

7 Conclusions

After the proliferation of malicious applications targeting devices which use the Android operating system, it became necessary to find solutions to address these threats. Therefore, the detection of malicious applications targeting Android becomes one of the most important scientific research's trends. Thus, a large number of frameworks have been proposed and developed from 2009 till these days. In this paper, an extensive study that includes more than 200 papers published between 2009 and 2019 has been conducted. In addition to analysing the studied works according to multiple criteria. Also, we have been proposed a robust and comprehensive taxonomy in the light of the studied works such that most of the conducted works in this domain can be classified under it. Also, we proposed a detailed schematic model called Schematic Review Model illustrating the process of Android malware detection. To our knowledge, this is the first time that this process is explained in this way with this amount of details. Furthermore, the features that used in the studied works have been discussed in detail and has been classified into multiple classes. Also, we examined in detail the most important challenges that facing the commonly used analysing methods. Moreover, we have made a comprehensive summary of all the works that covered in the paper with indicating the challenges facing the used analysing methods that have been addressed in each of which. We concluded that there is a shortage in most of the works that have been accomplished in this field, and therefore some points have been suggested to be covered in the future works.

Compliance with ethical standards

Conflict of interest None of the authors of this paper has a financial or personal relationship with other people or organizations that could inappropriately influence or bias the content of the paper.

References

1. Gartner_Q2 (2017) Gartner says demand for 4G smartphones in emerging markets spurred growth in second quarter of 2017. https://www.gartner.com/newsroom/id/3788963. Accessed 14 July 2018
2. Gartner_Q4 (2017) Gartner says worldwide sales of smartphones recorded first ever decline during the fourth quarter of 2017. https://www.gartner.com/newsroom/id/3859963. Accessed 11 July 2018
3. Statista_a (2018) Number of available applications in the Google Play Store from December 2009 to June 2018. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed 13 July 2018
4. Statista_b (2018) Growth of available mobile apps at Google Play worldwide from 2nd quarter 2015 to 1st quarter 2018. https://www.statista.com/statistics/185729/google-play-quarterly-growth-of-available-apps/. Accessed 13 July 2018
5. Statista_c (2018) Cumulative number of apps downloaded from the Google Play as of May 2016 (in billions). https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/. Accessed 14 July 2018
6. Pulse_Secure (2015) Mobile threat report. 2015: Pulse Secure Mobile Threat Center (MTC)
7. Symantec (2016) Internet security threat report. Internet report
8. G-Data (2017) 8,400 new android malware samples every day. https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day. Accessed 14 July 2018
9. McAfee (2017) New android malware found in 144 Google Play Apps. https://securingtomorrow.mcafee.com/mcafee-labs/android-malware-grabos-exposed-millions-to-pay-per-install-scam-on-google-play/. Accessed 14 July 2018
10. Faruki P et al (2015) Android security: a survey of issues, malware penetration, and defenses. IEEE Commun Surv Tutor 17(2):998–1022. https://doi.org/10.1109/comst.2014.2386139
11. Tan DJ, Chua T-W, Thing VL (2015) Securing android: a survey, taxonomy, and challenges. ACM Comput Surv (CSUR) 47(4):58
12. Tam K et al (2017) The evolution of android malware and android analysis techniques. ACM Comput Surv (CSUR) 49(4):76
13. Rashidi B, Fung CJ (2015) A survey of android security threats and defenses. JoWUA 6(3):3–35
14. Sadeghi A et al (2017) A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Trans Softw Eng 43(6):492–530. https://doi.org/10.1109/tse.2016.2615307
15. Feizollah A et al (2015) A review on feature selection in mobile malware detection. Digit Investig 13:22–37. https://doi.org/10.1016/j.diin.2015.02.001
16. Li L et al (2017) Static analysis of android apps: a systematic literature review. Inf Softw Technol 88:67–95. https://doi.org/10.1016/j.infsof.2017.04.001
17. Bakour K, Ünver HM, Ghanem R (2018) The android malware static analysis: techniques, limitations, and open challenges. In: 2018 3rd international conference on computer science and engineering (UBMK). IEEE
18. Android_PlayProtect (2018) Play protect. https://www.android.com/play-protect/. Accessed 14 July 2018
19. Xie L et al (2010) pBMDS: a behavior-based malware detection system for cellphone devices. In: Proceedings of the third ACM conference on wireless network security. ACM
20. Vidas T, Christin N, Cranor L (2011) Curbing android permission creep. In: Proceedings of the web
21. Bartel A et al (2012) Automatically securing permission-based software by reducing the attack surface: an application to android. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering. ACM
22. Stirparo P et al (2013) In-memory credentials robbery on android phones. In: 2013 world congress on internet security (WorldCIS). IEEE
23. Grace MC et al (2012) Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the fifth ACM conference on security and privacy in wireless and mobile networks. ACM
24. Buchanan WJ, Chiale S, Macfarlane R (2017) A methodology for the security evaluation within third-party android marketplaces. Digit Investig 23:88–98. https://doi.org/10.1016/j.diin.2017.10.002
25. Felt AP et al (2012) Android permissions: user attention, comprehension, and behavior. In: Proceedings of the eighth symposium on usable privacy and security. ACM
26. Huang TH-D, Kao H-Y (2017) R2-D2: color-inspired convolutional neural network (CNN)-based android malware detections. arXiv preprint arXiv:1705.04448
27. Yang M, Wen Q (2017) Detecting android malware by applying classification techniques on images patterns. In: 2017 IEEE 2nd international conference on cloud computing and big data analysis (ICCCBDA). IEEE
28. Karimi A, Moattar MH (2017) Android ransomware detection using reduced opcode sequence and image similarity. In: 2017 7th international conference on computer and knowledge engineering (ICCKE). IEEE
29. Jain A, Gonzalez H, Stakhanova N (2015) Enriching reverse engineering through visual exploration of android binaries. In: Proceedings of the 5th program protection and reverse engineering workshop. ACM
30. Yen Y-S, Sun H-M (2019) An android mutation malware detection based on deep learning using visualization of importance from codes. Microelectron Reliab 93:109–114
31. APKTool (2018) A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/. Accessed 14 July 2018
32. Gurulian I et al (2016) You can't touch this: consumer-centric android application repackaging detection. Future Gener Comput Syst 65:1–9. https://doi.org/10.1016/j.future.2016.05.021
33. Zhu H-J et al (2018) DroidDet: effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing 272:638–646. https://doi.org/10.1016/j.neucom.2017.07.030
34. JD-Project (2018) Java Decompiler project. http://jd.benow.ca/. Accessed 14 July 2018

35. pxb1988 (2018) Tools to work with android.dex and java.class files. https://github.com/pxb1988/dex2jar. Accessed 14 July 2018
36. Mike-Strobel (2018) Procyon: a suite of Java metaprogramming tools. https://bitbucket.org/mstrobel/procyon. Accessed 14 July 2018
37. SIIS (2018) ded: decompiling android applications. http://siis.cse.psu.edu/ded/. Accessed 14 July 2018
38. Androguard (2018) Reverse engineering, malware and goodware analysis of android applications. https://code.google.com/archive/p/androguard/. Accessed 15 July 2018
39. Skylot-jadx (2018) Dex to Java decompiler. https://github.com/skylot/jadx. Accessed 15 July 2018
40. Monkeyrunner (2018) https://developer.android.com/studio/test/monkeyrunner/. Accessed 15 July 2018
41. honeynet-droidbot (2018) A lightweight test input generator for android. https://github.com/honeynet/droidbot. Accessed 15 July 2018
42. Adb (2018) Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb. Accessed 15 July 2018
43. Logcat (2018) Logcat command-line tool. https://developer.android.com/studio/command-line/logcat. Accessed 15 July 2018
44. droidbox (2018) Dynamic analysis of android apps. https://github.com/pjlantz/droidbox. Accessed 15 July 2018
45. Robotium (2018) Android UI testing. https://github.com/RobotiumTech/robotium. Accessed 15 July 2018
46. Strace (2018) Linux syscall tracer. https://strace.io/. Accessed 15 July 2018
47. TcpDump (2018) tcpdump, a powerful command-line packet analyzer. http://www.tcpdump.org/. Accessed 15 July 2018
48. Chen S et al (2018) Automated poisoning attacks and defenses in malware detection systems: an adversarial machine learning approach. Comput Secur 73:326–344. https://doi.org/10.1016/j.cose.2017.11.007
49. Wang W, Zhao M, Wang J (2018) Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J Ambient Intell Humaniz Comput. https://doi.org/10.1007/s12652-018-0803-6
50. Kirubavathi G, Anitha R (2017) Structural analysis and detection of android botnets using machine learning techniques. Int J Inf Secur 17(2):153–167. https://doi.org/10.1007/s10207-017-0363-3
51. Tao G et al (2018) MalPat: mining patterns of malicious and benign android apps via permission-related APIs. IEEE Trans Reliab 67(1):355–369. https://doi.org/10.1109/tr.2017.2778147
52. Somarriba O, Zurutuza U (2017) A collaborative framework for android malware detection using DNS & dynamic analysis. In: 2017 IEEE 37th Central America and Panama convention (CONCAPAN XXXVII)
53. Wei S et al (2015) Mining network traffic for application category recognition on android platform. In: 2015 IEEE international conference on progress in informatics and computing (PIC). IEEE
54. Kurniawan H, Rosmansyah Y, Dabarsyah B (2015) Android anomaly detection system using machine learning classification. In: 2015 international conference on electrical engineering and informatics (ICEEI). IEEE
55. Alzaylaee MK, Yerima SY, Sezer S (2017) Emulator vs real phone: android malware detection using machine learning. In: Proceedings of the 3rd ACM on international workshop on security and privacy analytics. ACM
56. Shuaifu D, Tao W, Wei Z (2012) DroidLogger: reveal suspicious behavior of android applications via instrumentation. In: 2012 7th international conference on computing and convergence technology (ICCCT)
57. Ali-Gombe AI et al (2018) Toward a more dependable hybrid analysis of android malware using aspect-oriented programming. Comput Secur 73:235–248. https://doi.org/10.1016/j.cose.2017.11.006
58. Rastogi V, Chen Y, Enck W (2013) AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the third ACM conference on data and application security and privacy. ACM
59. Enck W et al (2014) TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst (TOCS) 32(2):5. https://doi.org/10.1145/2619091
60. Yan L-K, Yin H (2012) DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: USENIX security symposium
61. Kabakus AT, Dogru IA (2018) An in-depth analysis of android malware using hybrid techniques. Digit Investig 24:25–33. https://doi.org/10.1016/j.diin.2018.01.001
62. Yuan Z, Lu Y, Xue Y (2016) Droiddetector: android malware characterization and detection using deep learning. Tsinghua Sci Technol 21(1):114–123. https://doi.org/10.1109/TST.2016.7399288
63. Chen S et al (2016) StormDroid: a streaminglized machine learning-based system for detecting android malware, pp 377–388. https://doi.org/10.1145/2897845.2897860
64. Jang J-W et al (2016) Andro-Dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. Comput Secur 58:125–138. https://doi.org/10.1016/j.cose.2015.12.005
65. Zhao S et al (2014) Attack tree based android malware detection with hybrid analysis. In: Trust, security and privacy in computing and communications (TrustCom), pp 380–387. https://doi.org/10.1109/trustcom.2014.49
66. Mas'ud MZ et al (2013) Profiling mobile malware behaviour through hybrid malware analysis approach. In: 2013 9th international conference on information assurance and security (IAS). https://doi.org/10.1109/ISIAS.2013.6947737
67. Talha KA, Alper DI, Aydin C (2015) APK auditor: permission-based android malware detection system. Digit Investig 13:1–14. https://doi.org/10.1016/j.diin.2015.01.001
68. Samra AAA, Ghanem OA (2013) Analysis of clustering technique in android malware detection. In: 2013 seventh international conference on innovative mobile and internet services in ubiquitous computing. IEEE, pp 729–733. https://doi.org/10.1109/imis.2013.111
69. Arp D et al (2014) DREBIN: effective and explainable detection of android malware in your pocket. In: Ndss
70. Feizollah A et al (2017) AndroDialysis: analysis of android intent effectiveness in malware detection. Comput Secur 65:121–134. https://doi.org/10.1016/j.cose.2016.11.007
71. Zhang M et al (2014) Semantics-aware android malware classification using weighted contextual API dependency graphs. In: 2014 ACM SIGSAC conference on computer and communications security, pp 1105–1116. https://doi.org/10.1145/2660267.2660359
72. Moghaddam SH, Abbaspour M (2014) Sensitivity analysis of static features for android malware detection. In: 2014 22nd Iranian conference on electrical engineering (ICEE). IEEE
73. Wu D-J et al (2012) Droidmat: android malware detection through manifest and API calls tracing. In: 2012 seventh Asia joint conference on information security (Asia JCIS). IEEE
74. Martín A, Menéndez HD, Camacho D (2016) MOCDroid: multi-objective evolutionary classifier for android malware detection.

Soft Comput 21(24):7405–7415. https://doi.org/10.1007/s00500-016-2283-y
75. 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. Springer, Cham
76. Felt AP et al (2011) Android permissions demystified. In: Proceedings of the 18th ACM conference on computer and communications security. ACM, Chicago, Illinois, USA, pp 627–638
77. Sen S, Aysan AI, Clark JA (2018) SAFEDroid: using structural features for detecting android malwares. In: Security and privacy in communication networks. Springer, Cham
78. Milosevic N, Dehghantanha A, Choo K-KR (2017) Machine learning aided android malware classification. Comput Electr Eng 61:266–274. https://doi.org/10.1016/j.compeleceng.2017.02.013
79. Yerima SY, Sezer S (2018) DroidFusion: a novel multilevel classifier fusion approach for android malware detection. IEEE Trans Cybern. https://doi.org/10.1109/tcyb.2017.2777960
80. Zhang L, Niu Y, Wu X, Wang Z, Yibo X (2013) A3: automatic analysis of android malware. In: International workshop on cloud computing and information security
81. Park W et al (2014) Analyzing and detecting method of android malware via disassembling and visualization. In: 2014 international conference on information and communication technology convergence (ICTC). IEEE
82. Lee J, Lee S, Lee H (2015) Screening smartphone applications using malware family signatures. Comput Secur 52:234–249. https://doi.org/10.1016/j.cose.2015.02.003
83. Palumbo P et al (2017) A pragmatic android malware detection procedure. Comput Secur 70:689–701. https://doi.org/10.1016/j.cose.2017.07.013
84. Kwon J et al (2014) Droidgraph: discovering android malware by analyzing semantic behavior. In: 2014 IEEE conference on communications and network security (CNS). IEEE
85. Alam S et al (2017) DroidNative: automating and optimizing detection of android native code malware variants. Comput Secur 65:230–246. https://doi.org/10.1016/j.cose.2016.11.011
86. Wang C et al (2018) Research on data mining of permissions mode for android malware detection. Clust Comput. https://doi.org/10.1007/s10586-018-1904-x
87. Elish KO et al (2015) Profiling user-trigger dependence for android malware detection. Comput Secur 49:255–273. https://doi.org/10.1016/j.cose.2014.11.001
88. Brown J, Anwar M, Dozier G (2016) Detection of mobile malware: an artificial immunity approach, pp 74–80. https://doi.org/10.1109/spw.2016.32
89. Junaid M, Liu D, Kung D (2016) Dexteroid: detecting malicious behaviors in android apps using reverse-engineered life cycle models. Comput Secur 59:92–117. https://doi.org/10.1016/j.cose.2016.01.008
90. Pandita R, Xiao X, Yang W, Enck W, Xie T (2013) WHYPER: towards automating risk assessment of mobile applications. In: USENIX security symposium
91. Canfora G et al (2015) Effectiveness of opcode ngrams for detection of multi family android malware, pp 333–340. https://doi.org/10.1109/ares.2015.57
92. Sokolova K, Perez C, Lemercier M (2017) Android application classification and anomaly detection with graph-based permission patterns. Decis Support Syst 93:62–76. https://doi.org/10.1016/j.dss.2016.09.006
93. Gibler C et al (2012) AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: International conference on trust and trustworthy computing. Springer, Berlin
94. Moonsamy V, Rong J, Liu S (2014) Mining permission patterns for contrasting clean and malicious android applications. Future Gener Comput Syst 36:122–132. https://doi.org/10.1016/j.future.2013.09.014
95. Sanz B et al (2012) On the automatic categorisation of android applications. In: 2012 IEEE consumer communications and networking conference (CCNC). IEEE
96. Tan M et al (2017) Android malware detection combining feature correlation and Bayes classification model. In: 2017 IEEE 9th international conference on communication software and networks (ICCSN). IEEE
97. Wang C et al (2017) An android malware dynamic detection method based on service call co-occurrence matrices. Ann Telecommun 72(9–10):607–615. https://doi.org/10.1007/s12243-017-0580-9
98. Chang W-L, Sun H-M, Wu W (2016) An android behavior-based malware detection method using machine learning. In: 2016 IEEE international conference on signal processing, communications and computing (ICSPCC). IEEE
99. Amos B, Turner H, White J (2013) Applying machine learning classifiers to dynamic android malware detection at scale. In: 2013 9th international wireless communications and mobile computing conference (IWCMC). IEEE
100. Kumar A et al (2016) Machine learning based malware classification for android applications using multimodal image representations. In: 2016 10th international conference on intelligent systems and control (ISCO)
101. Bakour K, Daş GS, Ünver HM (2017) An intrusion detection system based on a hybrid Tabu-genetic algorithm. In: 2017 international conference on computer science and engineering (UBMK). IEEE
102. Shen T et al (2014) Detect android malware variants using component based topology graph. In: 2014 IEEE 13th international conference on trust, security and privacy in computing and communications (TrustCom), pp 406–413. https://doi.org/10.1109/trustcom.2014.52
103. Faruki P et al (2015) AndroSimilar: robust signature for detecting variants of android malware. J Inf Secur Appl 22:66–80. https://doi.org/10.1016/j.jisa.2014.10.011
104. Grace M et al (2012) RiskRanker: scalable and accurate zero-day android. In: Proceedings of the 10th international conference on mobile systems, applications, and services. ACM
105. Crussell J, Gibler C, Chen H (2015) AnDarwin: scalable detection of android application clones based on semantics. IEEE Trans Mob Comput 14(10):2007–2019. https://doi.org/10.1109/TMC.2014.2381212
106. Yang X et al (2017) Characterizing malicious android apps by mining topic-specific data flow signatures. Inf Softw Technol 90:27–39. https://doi.org/10.1016/j.infsof.2017.04.007
107. Chumachenko K (2017) Machine learning methods for malware detection and classification. http://urn.fi/URN:NBN:fi:amk-201703103155. Accessed 13 Mar 2019
108. Feldman S, Stadther D, Wang B (2014) Manilyzer: automated android malware detection through manifest analysis, pp 767–772. https://doi.org/10.1109/mass.2014.65
109. Sheen S, Anitha R, Natarajan V (2015) Android based malware detection using a multifeature collaborative decision fusion approach. Neurocomputing 151:905–912. https://doi.org/10.1016/j.neucom.2014.10.004
110. Fereidooni H et al (2016) ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In: 2016 8th IFIP international conference on new technologies, mobility and security (NTMS). IEEE
111. Ma L et al (2016) Ultra-lightweight malware detection of android using 2-level machine learning, pp 729–733. https://doi.org/10.1109/ICISCE.2016.161
112. Du Y, Wang J, Li Q (2017) An android malware detection approach using community structures of weighted

function call graphs. IEEE Access 5:17478–17486. https://doi.org/10.1109/access.2017.2720160
113. Aung Z, Zaw W (2013) Permission-based android malware detection. Int J Sci Technol Res 2(3):228–234
114. Verma S, Muttoo SK (2016) An android malware detection framework-based on permissions and intents. Def Sci J 66(6):618. https://doi.org/10.14429/dsj.66.10803
115. Karbab EB et al (2017) Android malware detection using deep learning on API method sequences. arXiv preprint arXiv:1712.08996. https://arxiv.org/abs/1712.08996v1
116. Hou S et al (2016) Deep4maldroid: A deep learning framework for android malware detection based on Linux kernel system call graphs. In: IEEE/WIC/ACM international conference on web intelligence workshops (WIW). IEEE
117. Nix R, Zhang J (2017) Classification of android apps and malware using deep neural networks. In: 2017 international joint conference on neural networks (IJCNN). IEEE
118. Tong F, Yan Z (2017) A hybrid approach of mobile malware detection in android. J Parallel Distrib Comput 103:22–31. https://doi.org/10.1016/j.jpdc.2016.10.012
119. Yerima SY, Muttik I, Sezer S (2015) High accuracy android malware detection using ensemble learning. IET Inf Secur 9(6):313–320. https://doi.org/10.1049/iet-ifs.2014.0099
120. Vergara JR, Estévez PA (2013) A review of feature selection methods based on mutual information. Neural Comput Appl 24(1):175–186. https://doi.org/10.1007/s00521-013-1368-0
121. Wang X et al (2017) Characterizing android apps' behavior for effective detection of malapps at large scale. Future Gener Comput Syst 75:30–45. https://doi.org/10.1016/j.future.2017.04.041
122. Omer Fadl Elssied N, Ibrahim O, Hamza Osman A (2014) A novel feature selection based on one-way ANOVA F-test for e-mail spam classification. Res J Appl Sci Eng Technol 7(3):625–638. https://doi.org/10.19026/rjaset.7.299
123. Coronado-De-Alba LD, Rodríguez-Mota A, Escamilla-Ambrosio PJ (2016) Feature selection and ensemble of classifiers for android malware detection. In: 2016 8th IEEE Latin-American conference on communications (LATINCOM). IEEE
124. Rosario SF, Thangadurai K (2015) RELIEF: feature selection approach. Int J Innov Res Dev 4(11):219
125. Ng DV, Hwang J-IG (2014) Android malware detection using the dendritic cell algorithm. In: 2014 international conference on machine learning and cybernetics (ICMLC). IEEE
126. Wu S et al (2016) Effective detection of android malware based on the usage of data flow APIs and machine learning. Inf Softw Technol 75:17–25. https://doi.org/10.1016/j.infsof.2016.03.004
127. Spreitzenbarth M et al (2014) Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques. Int J Inf Secur. https://doi.org/10.1007/s10207-014-0250-0
128. Idrees F et al (2017) PIndroid: a novel android malware detection system using ensemble learning methods. Comput Secur 68:36–46. https://doi.org/10.1016/j.cose.2017.03.011
129. Mariconti E et al (2016) Mamadroid: detecting android malware by building markov chains of behavioral models. arXiv preprint arXiv:1612.04433. https://arxiv.org/abs/1612.04433v3
130. Zhou Y, Jiang X (2012) Dissecting android malware: characterization and evolution, pp 95–109. https://doi.org/10.1109/sp.2012.16
131. Zheng M, Lee PP, Lui JC (2012) ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, Berlin
132. Yerima SY, McWilliams G, Sezer S (2014) Analysis of Bayesian classification-based approaches for android malware detection. IET Inf Secur 8(1):25–36. https://doi.org/10.1049/iet-ifs.2013.0095
133. Faruki P et al (2016) Android code protection via obfuscation techniques: past, present and future directions. arXiv preprint arXiv:1611.10231. https://arxiv.org/abs/1611.10231v1
134. Maiorca D et al (2015) Stealth attacks: an extended insight into the obfuscation effects on android malware. Comput Secur 51:16–31. https://doi.org/10.1016/j.cose.2015.02.007
135. Karlo-Mravunac (2017). https://sgros-students.blogspot.com/search/label/obfuscation. Accessed 18 July 2018
136. Mavrogiannopoulos N, Kisserli N, Preneel B (2011) A taxonomy of self-modifying code for obfuscation. Comput Secur 30(8):679–691. https://doi.org/10.1016/j.cose.2011.08.007
137. Rastogi V, Chen Y, Jiang X (2013) DroidChameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security. ACM, Hangzhou, pp 329–334
138. Lantz P, Johansson B (2015) Towards bridging the gap between Dalvik bytecode and native code during static analysis of android applications. In: 2015 international wireless communications and mobile computing conference (IWCMC). IEEE
139. Spreitzenbarth M et al (2013) Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th annual ACM symposium on applied computing. ACM, Coimbra, pp 1808–1815
140. Glodek W, Harang R (2013) Rapid permissions-based detection and analysis of mobile malware using random decision forests, pp 980–985. https://doi.org/10.1109/milcom.2013.170
141. Zhou Y et al (2012) Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS, vol 25
142. Zheng M, Sun M, Lui JCS (2014) DroidTrace: a ptrace based android dynamic analysis system with forward execution capability. In: 2014 international wireless communications and mobile computing conference (IWCMC)
143. Hu W et al (2014) Migdroid: detecting app-repackaging android malware via method invocation graph. In: 2014 23rd international conference on computer communication and networks (ICCCN). IEEE
144. Zhou W et al (2012) Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the second ACM conference on data and application security and privacy. ACM
145. Alzaylaee MK, Yerima SY, Sezer S (2017) Improving dynamic analysis of android apps using hybrid test input generation. In: 2017 international conference on cyber security and protection of digital services (cyber security). IEEE
146. Vidas T, Christin N (2014) Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on information, computer and communications security-ASIA CCS '14, pp 447–458
147. Alzaylaee MK, Yerima SY, Sezer S (2016) 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
148. Li J et al (2014) Research of android malware detection based on network traffic monitoring. In: 2014 IEEE 9th conference on industrial electronics and applications (ICIEA). IEEE
149. Tam K et al (2015) CopperDroid: automatic reconstruction of android malware behaviors. In: NDSS
150. Bugiel S et al (2011) XManDroid: a new android evolution to mitigate privilege escalation attacks. Technical report TR-2011-04, Technische Universit, Darmstadt
151. Russello G et al (2011) Yaase: yet another android security extension. In: 2011 IEEE third international conference on privacy, security, risk and trust (PASSAT) and 2011 IEEE third international conference on social computing (SocialCom). IEEE

152. Ongtang M et al (2012) Semantically rich application-centric security in android. Secur Commun Netw 5(6):658–673. https://doi.org/10.1002/sec.360
153. Rastogi V, Chen Y, Jiang X (2013) Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security. ACM
154. Huang H et al (2013) A framework for evaluating mobile app repackaging detection algorithms. In: International conference on trust and trustworthy computing. Springer, Berlin
155. Xue L et al (2017) Adaptive unpacking of android apps. In: IEEE/ACM 39th international conference, pp 358–369. https://doi.org/10.1109/icse.2017.40
156. Li B et al (2018) AppSpear: automating the hidden-code extraction and reassembling of packed android malware. J Syst Softw 140:3–16. https://doi.org/10.1016/j.jss.2018.02.040
157. Amalfitano D et al (2012) Using GUI ripping for automated testing of android applications. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering. ACM
158. Machiry A, Tahiliani R, Naik M (2013) Dynodroid: an input generation system for android apps. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM
159. Spreitzenbarth M et al (2013) Mobile-sandbox: having a deeper look into android applications, pp 1808–1815
160. Zhang Y et al (2019) SaaS: a situational awareness and analysis system for massive android malware detection. Future Gener Comput Syst 95:548–559
161. Zhang L, Thing VL, Cheng Y (2019) A scalable and extensible framework for android malware detection and family attribution. Comput Secur 80:120–133
162. Martín I, Hernández JA (2019) CloneSpot: fast detection of android repackages. Future Gener Comput Syst 94:740–748
163. Fan M et al (2018) Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Trans Inf Forensics Secur 13(8):1890–1905. https://doi.org/10.1109/tifs.2018.2806891
164. Papadopoulos H et al (2018) Android malware detection with unbiased confidence guarantees. Neurocomputing 280:3–12. https://doi.org/10.1016/j.neucom.2017.08.072
165. Jha AK, Lee WJ (2018) An empirical study of collaborative model and its security risk in android. J Syst Softw 137:550–562. https://doi.org/10.1016/j.jss.2017.07.042
166. Li J et al (2018) Significant permission identification for machine-learning-based android malware detection. IEEE Trans Ind Inform 14(7):3216–3225
167. Zhao C et al (2018) Quick and accurate android malware detection based on sensitive APIs. In: 2018 IEEE international conference on smart internet of things (SmartIoT). IEEE
168. Şahın DÖ et al (2018) New results on permission based static analysis for android malware. In: 2018 6th international symposium on digital forensic and security (ISDFS). IEEE
169. Jin Y et al (2018) Android malware detector exploiting convolutional neural network and adaptive classifier selection. In: 2018 IEEE 42nd annual computer software and applications conference (COMPSAC). IEEE
170. Hasegawa C, Iyatomi H (2018) One-dimensional convolutional neural networks for android malware detection. In:2018 IEEE 14th international colloquium on signal processing & its applications (CSPA). IEEE
171. Koli J (2018) RanDroid: android malware detection using random machine learning classifiers. In: 2018 technologies for smart-city energy security and power (ICSESP). IEEE
172. Riasat R et al (2018) Onamd: an online android malware detection approach. In: 2018 international conference on machine learning and cybernetics (ICMLC). IEEE
173. Jung J et al (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). IEEE
174. Arshad S et al (2018) SAMADroid: a novel 3-level hybrid malware detection model for android operating system. IEEE Access 6:4321–4339
175. Arora A, Peddoju SK (2018) NTPDroid: a hybrid android malware detector using network traffic and system permissions. In: 2018 17th IEEE international conference on trust, security and privacy in computing and communications/12th IEEE international conference on big data science and engineering (TrustCom/BigDataSE). IEEE
176. Rehman Z-U et al (2017) Machine learning-assisted signature and heuristic-based detection of malwares in android devices. Comput Electr Eng. https://doi.org/10.1016/j.compeleceng.2017.11.028
177. Martinelli F, Marulli F, Mercaldo F (2017) Evaluating convolutional neural network for effective mobile malware detection. Procedia Comput Sci 112:2372–2381
178. Liang H, Song Y, Xiao D (2017) An end-to-end model for android malware detection. In: 2017 IEEE international conference on intelligence and security informatics (ISI). IEEE
179. Su M-Y, Chang J-Y, Fung K-T (2017) Machine learning on merging static and dynamic features to identify malicious mobile apps. In: 2017 ninth international conference on ubiquitous and future networks (ICUFN). IEEE
180. Narayanan A et al (2017) Context-aware, adaptive, and scalable android malware detection through online learning. IEEE Trans Emerg Top Comput Intell 1(3):157–175
181. Li D et al (2017) FgDetector: fine-grained android malware detection. In: 2017 IEEE second international conference on data science in cyberspace (DSC). IEEE
182. Mohsen F et al (2017) Detecting android malwares by mining statically registered broadcast receivers. In: 2017 IEEE 3rd international conference on collaboration and internet computing (CIC). IEEE
183. Zhu D et al (2017) DeepFlow: deep learning-based malware detection by mining Android application for abnormal usage of sensitive data. In: 2017 IEEE symposium on computers and communications (ISCC). IEEE
184. Fan M et al (2017) Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. IEEE Trans Inf Forensics Secur 12(8):1772–1785
185. Goyal R et al (2016) SafeDroid: a distributed malware detection service for android. , pp 59–66. https://doi.org/10.1109/SOCA.2016.14
186. Song J et al (2016) An integrated static detection and analysis framework for android. Pervasive Mob Comput 32:15–25. https://doi.org/10.1016/j.pmcj.2016.03.003
187. Narayanan A et al (2016) Adaptive and scalable android malware detection through online learning. In: 2016 international joint conference on neural networks (IJCNN). IEEE
188. Chen W et al (2016) More semantics more robust, pp 147–158. https://doi.org/10.1145/2939918.2939931
189. Ju S-H, Seo H-S, Kwak J (2016) Research on android malware permission pattern using permission monitoring system. Multimed Tools Appl 75(22):14807–14817. https://doi.org/10.1007/s11042-016-3273-x
190. Wang K, Song T, Liang A (2016) Mmda: metadata based malware detection on android. In: 2016 12th international conference on computational intelligence and security (CIS). IEEE

191. Wang Z et al (2016) DroidDeepLearner: identifying android malware using deep learning. In: 2016 IEEE 37th Sarnoff symposium. IEEE
192. Zhang X et al (2016) A novel android malware detection method based on markov blanket. In: IEEE international conference on data science in cyberspace (DSC). IEEE
193. Yang M, Wen Q (2016) Detecting android malware with intensive feature engineering. In: 2016 7th IEEE international conference on software engineering and service science (ICSESS). IEEE
194. Martín A, Menéndez HD, Camacho D (2016) String-based malware detection for android environments. In: International symposium on intelligent and distributed computing. Springer, Berlin
195. Su X et al (2016) A deep learning approach to android malware feature learning and detection. In: 2016 IEEE Trustcom/BigDataSE/I SPA. IEEE
196. Wang Z et al (2016) DroidChain: a novel android malware detection method based on behavior chains. Pervasive Mob Comput 32:3–14
197. Karbab EB, Debbabi M, Mouheb D (2016) Fingerprinting android packaging: generating DNAs for malware detection. Digit Investig 18:S33–S45
198. Zhang X, Jin Z (2016) A new semantics-based android malware detection. In:2016 2nd IEEE international conference on computer and communications (ICCC). IEEE
199. Morales-Ortega S et al (2016) Native malware detection in smartphones with android os using static analysis, feature selection and ensemble classifiers. In: 2016 11th international conference on malicious and unwanted software (MALWARE). IEEE
200. Li Q, Li X (2015) Android malware detection based on static analysis of characteristic tree, pp 84–91. https://doi.org/10.1109/cyberc.2015.88
201. Rosmansyah Y, Dabarsyah B (2015) Malware detection on android smartphones using API class and machine learning. In: 2015 International Conference on Electrical Engineering and Informatics (ICEEI). IEEE
202. Li W, Ge J, Dai G (2015) Detecting malware for android platform: an SVM-based approach. In: 2015 IEEE 2nd international conference on cyber security and cloud computing (CSCloud). IEEE
203. Gordon MI et al (2015) Information-flow analysis of android applications in DroidSafe. https://doi.org/10.14722/ndss.2015.23089
204. Damshenas M et al (2015) M0Droid: an android behavioral-based malware detection model. J Inf Priv Secur 11(3):141–157. https://doi.org/10.1080/15536548.2015.1073510
205. Almin SB, Chatterjee M (2015) A novel approach to detect android malware. Procedia Comput Sci 45:407–417. https://doi.org/10.1016/j.procs.2015.03.170
206. Lindorfer M, Neugschwandtner M, Platzer C (2015) Marvin: efficient and comprehensive mobile app classification through static and dynamic analysis. In: 2015 IEEE 39th annual computer software and applications conference (COMPSAC). IEEE
207. Bierma M et al (2014) Andlantis: large-scale android dynamic analysis. arXiv preprint arXiv:1410.7751. https://arxiv.org/abs/1410.7751v1
208. Adebayo OS, AbdulAziz N (2014) Android malware classification using static code analysis and a priori algorithm improved with particle swarm optimization. In: 2014 fourth world congress on information and communication technologies (WICT). IEEE
209. Liang S et al (2014) An effective online scheme for detecting android malware. In: 2014 23rd international conference on computer communication and networks (ICCCN). IEEE
210. Lagerspetz E et al (2014) MDoctor: a mobile malware prognosis application, pp 201–206. https://doi.org/10.1109/icdcsw.2014.36
211. Merlo A, Migliardi M, Fontanelli P (2014) On energy-based profiling of malware in android. In: 2014 international conference on high performance computing & simulation (HPCS)
212. Suarez-Tangil G et al (2014) Dendroid: a text mining approach to analyzing and classifying code structures in android malware families. Expert Syst Appl 41(4):1104–1117. https://doi.org/10.1016/j.eswa.2013.07.106
213. Hsiao SW et al (2014) PasDroid: real-time security enhancement for android, pp 229–235. https://doi.org/10.1109/imis.2014.28
214. Yerima SY, Sezer S, Muttik I (2014) Android malware detection using parallel machine learning classifiers. In: 2014 eighth international conference on next generation mobile apps, services and technologies (NGMAST). IEEE
215. Feng Y et al (2014) 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
216. Xiaoyan Z, Juan F, Xiujuan W (2014) Android malware detection based on permissions. https://doi.org/10.1049/cp.2014.0605
217. Xiangyu J (2014) Android malware detection through permission and package. In: 2014 international conference on wavelet analysis and pattern recognition
218. Liang S, Du X (2014) Permission-combination-based scheme for android mobile malware detection. In: 2014 IEEE international conference on communications (ICC). IEEE
219. Idrees F, Rajarajan M (2014) Investigating the android intents and permissions for malware detection. In: 2014 IEEE 10th international conference on wireless and mobile computing, networking and communications (WiMob). IEEE
220. Raphael R, Vinod P, Omman B (2014) X-ANOVA and X-Utest features for android malware analysis. In: 2014 international conference on advances in computing, communications and informatics (ICACCI). IEEE
221. Wolfe B, Elish KO, Yao DD (2014) Comprehensive behavior profiling for proactive android malware detection. In: international conference on information security. Springer, Berlin
222. Seo S-H et al (2014) Detecting mobile malware threats to homeland security through static analysis. J Netw Comput Appl 38:43–53
223. Deepa K, Radhamani G, Vinod P (2015) Investigation of feature selection methods for android malware analysis. Procedia Comput Sci 46:841–848
224. Shabtai A et al (2014) Mobile malware detection through analysis of deviations in application network behavior. Comput Secur 43:1–18. https://doi.org/10.1016/j.cose.2014.02.009
225. Yerima SY et al (2013) A new android malware detection approach using Bayesian classification. In: 2013 IEEE 27th international conference on advanced information networking and applications (AINA)
226. Tenenboim-Chekina L et al (2013) Detecting application update attack on mobile devices through network features. In: 2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS). IEEE
227. Karami M et al (2013) Behavioral analysis of android applications using automated instrumentation, pp 182–187. https://doi.org/10.1109/sere-c.2013.35
228. Vasquez S, Simmonds J (2013) Mobile application monitoring. In: 2013 32nd international conference of the Chilean computer science society, pp 30–32. https://doi.org/10.1109/sccc.2013.16
229. Backes M et al (2014) AppGuard—fine-grained policy enforcement for untrusted android applications. In: Revised selected papers of the 8th international workshop on data privacy

management and autonomous spontaneous security, vol 8247. Springer, Berlin, pp 213–231
230. Peiravian N, Zhu X (2013) Machine learning for android malware detection using permission and API calls, pp 300–305. https://doi.org/10.1109/ictai.2013.53
231. Lu Y et al (2013) Android malware detection technology based on improved Bayesian classification, pp 1338–1341. https://doi.org/10.1109/imccc.2013.297
232. Wei Y et al (2013) On behavior-based detection of malware on android platform. In: 2013 IEEE global communications conference (GLOBECOM). IEEE
233. Alam MS, Vuong ST (2013) Random forest classification for detecting android malware. In: IEEE international conference on green computing and communications (GreenCom), 2013 IEEE internet of things (iThings/CPSCom), and IEEE cyber, physical and social computing. IEEE
234. Ham H-S, Choi M-J (2013) Analysis of android malware detection performance using machine learning classifiers. In: 2013 international conference on ICT convergence (ICTC). IEEE
235. Zhang Y et al (2013) Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security. ACM
236. Zheng M, Sun M, Lui J (2013) Droidanalytics: a signature based analytic system to collect, extract, analyze and associate android malware. arXiv preprint arXiv:1302.7212
237. Eder T et al (2013) ANANAS—a framework for analyzing android applications. In: 2013 international conference on availability, reliability and security, pp 711–719
238. Sanz B et al (2013) Puma: permission usage to detect malware in android. In: International joint conference CISIS'12-ICEUTE 12-SOCO 12 special sessions. Springer, Berlin
239. Zheng C et al (2012) SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications. In: Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices. ACM, Raleigh, pp 93–104
240. Wei X et al (2012) ProfileDroid: multi-layer profiling of android applications. In: 18th annual international conference on mobile computing and networking. ACM
241. Wei X et al (2012) Permission evolution in the android ecosystem. In: Proceedings of the 28th annual computer security applications conference. ACM
242. Dini G et al (2012) MADAM: a multi-level anomaly detector for android malware. In: International conference on mathematical methods, models, and architectures for computer network security. Springer, Berlin
243. Sahs J, Khan L (2012) A machine learning approach to android malware detection. In: 2012 European intelligence and security informatics conference (EISIC). IEEE
244. Yang Z, Yang M (2012) Leakminer: detect information leakage on android with static taint analysis. In: 2012 third world congress on software engineering (WCSE). IEEE
245. Gascon H et al (2013) Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on artificial intelligence and security. ACM
246. Su X, Chuah M, Tan G (2012) Smartphone dual defense protection framework: detecting malicious applications in android markets. In: 2012 8th international conference on mobile ad hoc and sensor networks (MSN), pp 153–160
247. Burguera I, Zurutuza U, Nadjm-Tehrani S (2011) 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
248. Chin E et al (2011) Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on mobile systems, applications, and services. ACM, Bethesda, Maryland, USA, pp 239–252
249. Isohara T, Takemori K, Kubota A (2011) Kernel-based behavior analysis for android malware detection. In: 2011 seventh international conference on computational intelligence and security, pp 1011–1015
250. Nauman M, Khan S, Zhang X (2010) Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM symposium on information, computer and communications security. ACM
251. Ongtang M, Butler K, McDaniel P (2010) Porscha: policy oriented secure content handling in android. In: Proceedings of the 26th annual computer security applications conference. ACM
252. Conti M, Nguyen VTN, Crispo B (2011) CRePE: context-related policy enforcement for android, vol 6531, pp 331–345. https://doi.org/10.1007/978-3-642-18178-8_29
253. Portokalidis G et al (2010) Paranoid android: versatile protection for smartphones. In: Proceedings of the 26th annual computer security applications conference. ACM
254. Barrera D et al (2010) A methodology for empirical analysis of permission-based security models and its application to android, pp 73–84
255. Blasing T et al (2010) An android application sandbox system for suspicious software detection. In: 2010 5th international conference on malicious and unwanted software (MALWARE 2010). IEEE
256. Shabtai A, Fledel Y, Elovici Y (2010) Automated static code analysis for classifying android applications using machine learning. In: 2010 international conference on computational intelligence and security, pp 329–333
257. Enck W, Ongtang M, McDaniel P (2009) On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on computer And communications security. ACM, Chicago, Illinois, USA, pp 235–245