Android mobile malware detection using machine learning: a systematic review.

SENANAYAKE, J., KALUTARAGE, H. and AL-KADRI, M.O.

2021

electronics

MDPI

Review

Android Mobile Malware Detection Using Machine Learning: A Systematic Review

Janaka Senanayake [1,*], Harsha Kalutarage [1] and Mhd Omar Al-Kadri [2]

[1] School of Computing, Robert Gordon University, Aberdeen AB10 7QB, UK; h.kalutarage@rgu.ac.uk
[2] School of Computing and Digital Technology, Birmingham City University, Birmingham B4 7XG, UK; omar.alkadri@bcu.ac.uk
[*] Correspondence: j.senanayake@rgu.ac.uk

Abstract: With the increasing use of mobile devices, malware attacks are rising, especially on Android phones, which account for 72.2% of the total market share. Hackers try to attack smartphones with various methods such as credential theft, surveillance, and malicious advertising. Among numerous countermeasures, machine learning (ML)-based methods have proven to be an effective means of detecting these attacks, as they are able to derive a classifier from a set of training examples, thus eliminating the need for an explicit definition of the signatures when developing malware detectors. This paper provides a systematic review of ML-based Android malware detection techniques. It critically evaluates 106 carefully selected articles and highlights their strengths and weaknesses as well as potential improvements. Finally, the ML-based methods for detecting source code vulnerabilities are discussed, because it might be more difficult to add security after the app is deployed. Therefore, this paper aims to enable researchers to acquire in-depth knowledge in the field and to identify potential future research and development directions.

Keywords: Android security; malware detection; code vulnerability; machine learning

check for updates

Citation: Senanayake, J.; Kalutarage, H.; Al-Kadri, M.O. Android Mobile Malware Detection Using Machine Learning: A Systematic Review. *Electronics* **2021**, *10*, 1606. https://doi.org/10.3390/electronics10131606

Academic Editor: Rui Pedro Lopes

Received: 29 May 2021
Accepted: 29 June 2021
Published: 5 July 2021

1. Introduction

In this technological era, smartphone usage and its associated applications are rapidly increasing [1] due to the convenience and efficiency in various applications and the growing improvement in the hardware and software on smart devices. It is predicted that there will be 4.3 billion smartphone users by 2023 [1]. Android is the most widely used mobile operating system (OS). As of May 2021, its market share was 72.2% [2]. The second highest market share of 26.99% is owned by Apple iOS, while the rest of the 0.81% is shared among Samsung, KaiOS, and other small vendors [2]. Google Play is the official app store for Android-based devices. The number of apps published on it was over 2.9 million as of May 2021. Of these, more than 2.5 million apps are classified as regular apps, while 0.4 million apps are classified as low-quality apps by AppBrain [3]. Android's worldwide popularity makes it a more attractive target for cybercriminals and is more at risk from malware and viruses. Studies have proposed various methods of detecting these attacks, and ML is one of the most prominent techniques among them [4]. This is because ML techniques are able to derive a classifier from a (limited) set of training examples. The use of examples thus avoids the need to explicitly define signatures in developing malware detectors. Defining signatures requires expertise and tedious human involvement and for some attack scenarios explicit rules (signatures) do not exist, but examples can be obtained easily. Numerous industrial and academic research has been carried out on ML-based malware detection on Android, which is the focus of this review paper.

The taxinomical classification of the review is presented in Figure 1. Android users and developers are known to make mistakes that expose them to unnecessary dangers and risks of infecting their devices with malware. Therefore, in addition to malware detection techniques, methods to identify these mistakes are important and covered in this paper (see Figure 1). Detecting malware with ML involves two main phases, which are analyzing Android Application Packages (APKs) to derive a suitable set of features and then training machine and deep learning (DL) methods on derived features to recognize malicious APKs. Hence, a review of the methods available for APK analysis is included, which consists of static, dynamic, and hybrid analysis. Similar to malware detection, vulnerability detection in software code involves two main phases, namely feature generation through code analysis and training ML on derived features to detect vulnerable code segments. Hence, these two aspects are included in the review's taxonomy.

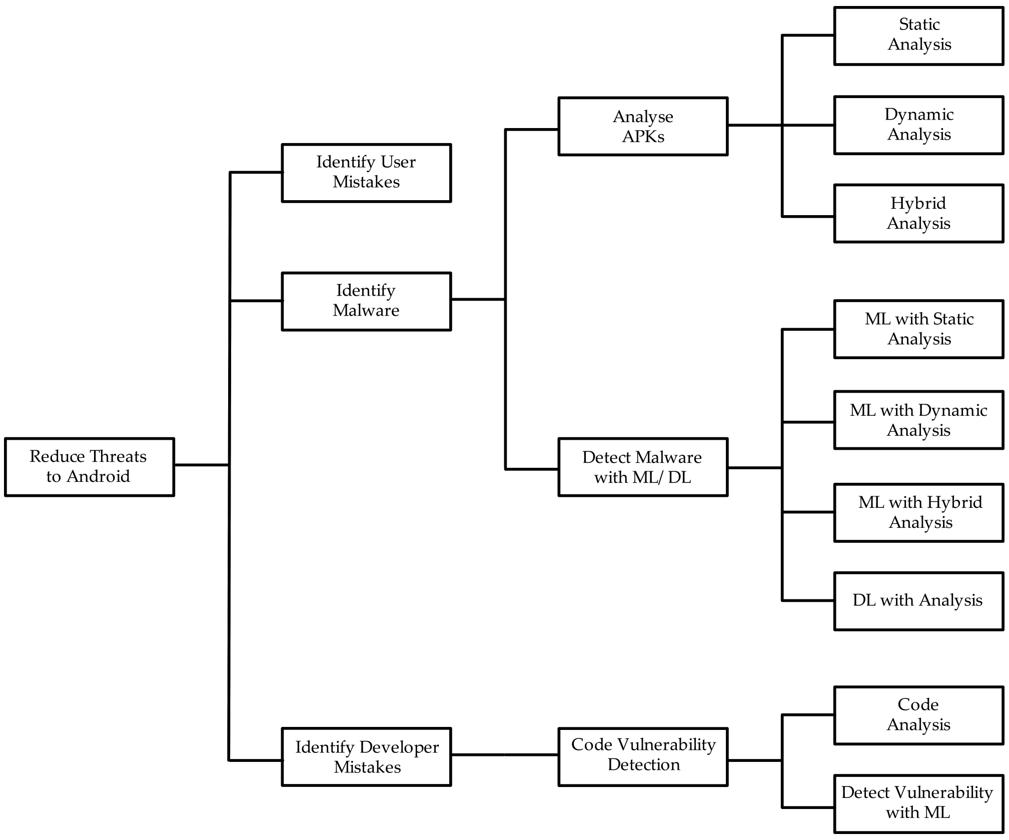

Figure 1. Taxonomy of the review.

The rest of this paper is organised as follows: Section 2 lays out the background to this study. Section 3 provides a detailed description of the review methodology, while Section 4 discusses related previous reviews on the topic. Section 5 discusses static, dynamic, and hybrid analysis techniques for Android malware detection and the application of ML and DL methods as well as a comparison of the methods used in the individual studies. Section 6 discusses ML methods to identify code vulnerabilities, with Section 7 exploring the results and discussions thereof. Finally, Section 8 concludes the paper.

2. Background

This section provides a high-level overview of the Android architecture and its built-in security as well as potential threat vectors for Android. It also provides an introduction to the ML process as it would be useful for non-ML background readers to understand the contents of this paper.

2.1. Android Architecture

Android is built on top of the Linux Kernel. Linux is chosen because it is open source, verifies the pathway evidence, provides drivers and mechanisms for networking, and manages virtual memory, device power, and security [5]. Android has a layered architecture [6]. The layers are arranged from bottom to top. On top of the Linux Kernal

Layer, the Hardware Abstraction Layer, Native C/C++ Libraries and Android Runtime, Java Application Programming Interface (API) Framework, and System Apps are stacked on top of each. Each layer is responsible for a particular task. For example, the Java API Framework provides Java libraries to perform a location awareness application-related activity such as identifying the latitude and the longitude.

Android-based applications and some system services use the Android Runtime (ART). Dalvik was the runtime environment used before the ART. Both ART and Dalvik were created for the Android applications-related projects. The ART executes the Dalvik Executable (DEX) format and the bytecode specification [7]. The other aspects are memory management and power management since the Android-based applications run on battery-powered devices with limited memory. Therefore, the Android operating system is designed in a way that any resource can be well managed [5]. For instance, the Android OS will automatically suspend the application in memory if an application is not in use at the moment. This state is known as the running state of the application life cycle. By doing this, it can preserve the power that can be utilised when the application reopens. Otherwise, the applications are kept idle until they are closed [8].

Built-In Security

Android comes with security already built in. It is a privileged separated operating system [9]. Sandboxing technique and the permission system in Android reduce some risks and bugs in the application. Sandboxing technique in Android isolates the running applications using unique identifiers which are based on the Linux environment [10]. Without having permissions granted from the user at the time of app installation or reconfiguration, apps cannot access system resources. If some of the permissions are not granted, then the application itself will not be usable. When a system update or upgrade happens, several improvements happen in terms of security and privacy. For example, Android 11, the latest stable Android version contains some changes related to security and privacy such as scoped storage enforcement, one-time permissions, permissions auto-reset, background location access, package visibility, and foreground services [11].

However, there are possibilities of malware attacks to exploit some vulnerabilities in the applications developed by various users, because the Google Play Store will not detect some vulnerabilities when publishing applications in the Play Store as in Apple App Store [12].

2.2. Threats to Android

While Android has good built-in security measures, there are several design weaknesses and security flaws that have become threats to its users. Awareness about those threats is also important to perform a proper malware detection and vulnerability analysis. Many research and technical reports have been published related to the Android threats [13] and classified Android threats based on the attack methodology. Social engineering attacks, physical access retrieving attacks, and network attacks are described under the ways of gaining access to the device. For the vulnerabilities and exploitation methods, man in the middle attacks, return to libc attacks, JIT-Spraying attacks, third-party library vulnerabilities, Dalvik vulnerabilities, network architecture vulnerabilities, virtualization vulnerabilities, and Android debug bridges and kernel vulnerabilities are considered.

The survey in [14] identified four types of attacks to Android; hardware-based attacks, kernel-based attacks, Hardware Abstraction Layer (HAL) based attacks, and application-based attacks. Hardware-based attacks such as Rowhammer, Glitch, and Drammer are related to sensors, touch screens, communication media, and DRAM. Kernel-based attacks such as Gooligan, DroidKungfu, Return-oriented Programming are related to Root Privilege, Memory, Boot Loader, and Device Driver. HAL-based attacks such as Return to User and TocTou are related to interfaces for cameras, Bluetooth, Wi-Fi, Global Positioning System (GPS), and Radio. Application-based attacks such as AdDetect, WuKong, and LibSift are related to third-party libraries, Intra-Library collusion, and privilege escalations.

Android applications are easily penetrable with proper knowledge of Android programming if suitable security mechanisms are not in place. In addition, Android marketplaces such as Google Play are not following extensive security protocols when new apps are published. For example, the Android game known as Angry Bird was hacked and the hacker managed to get into its APK file and embed a malicious code that sent text messages unknowingly by the user. The cost was 15 GPB to the user per message. More than a thousand users were affected [15].

2.2.1. Malware Attacks on Android

Malware attacks are the most common case that can be identified as a threat to Android. There are various definitions for malware given by many researchers depending on the harm they cause. The ultimate meaning of the malware is any of the malicious application with a piece of malicious code [16] which has an evil intent [17] to obtain unauthorised access and to perform neither legal nor ethical activities while violating the three main principles in security: confidentiality, integrity, and availability.

Malware related to smart devices can be classified into three perspectives as attack goals and behaviour, distribution and infection routes, and privilege acquisition modes [18]. Frauds, spam emails, data theft, and misuse of resources can be mentioned as the attack goals and behaviour perspective. Software markets, browsers, networks, and devices can be identified as the distribution and infection routes. Technical exploitation and user manipulation such as social engineering can be listed under the privilege and acquisition modes. Malware specifically related to the Android operating system is identified as Android malware [19] which harms or steals data from an Android-based mobile device. These are categorised as Trojans, Spyware, adware, ransomware, worms, botnet, and backdoors [20]. Google describes malware as potentially harmful applications. They classified malware as commercial and noncommercial spyware, backdoors, privilege escalation, phishing, types of frauds such as click fraud, toll fraud, Short Message Service (SMS) fraud, and Trojans [21].

App collusion also should be considered when studying malware. App collusion is two or more apps working together to achieve a malicious goal [22]. However, if those apps perform individually, there is no possibility of a malicious activity happening. It is a must to detect malicious inter-app communication and app permissions for app collusion detection [23,24].

2.2.2. Users and App Developers' Mistakes

The mistakes can happen knowingly or unknowingly from the developers as well as users. These mistakes may lead to threats arising to Android OS and its applications.

It has been identified that users are responsible for most security issues [25]. Some common mistakes done by the users will lead to serious threats in an Android application. At the time of installing Android applications, users will be asked to allow some permissions. However, all the users may not understand the purpose of each permission. They allow permission to run the application without considering the severity of it. Fraudulent applications might steal data and perform unintended tasks after getting the required permissions. It is possible to arise threats to the Android systems due to the mistakes performed by the app developers at the time of developing applications. In the publishing stage of the Android apps, Google Play will have only limited control over the code vulnerabilities in the applications. Sometimes developers are specifying unwanted permissions in the Android manifest file mistakenly, which encourages the user to grant the permissions if the permissions were categorised as not simple permissions [26]. Though the app development companies and some of the app stores are advising about following the security guidelines implemented at the time of development, many developers still fail to write secure codes to build secured mobile applications [27].

2.3. Machine Learning Process

ML is a branch of artificial intelligence that focuses on developing applications by learning from data without explicitly programming how the learned tasks are performed. The traditional ML methods make predictions based on past data. ML process lifecycle consists of multiple sequential steps. They are data extraction, data preprocessing, feature selection, model training, model evaluation, and model deployment [9]. Supervised learning, unsupervised learning, semisupervised learning, reinforcement learning, and deep learning are the different subcategories of ML [28]. The supervised learning approach uses a labelled dataset to train the model to solve classification and regression problems depend on the output variable type (continuous or discreet). Unsupervised learning is used to identify the internal structures (clusters), the characteristics of a dataset, and a labelled dataset is not required to train the model. A mix of both supervised and unsupervised learning techniques are applied in semisupervised learning and used in a case of limited labelled data in the used dataset [29]. The learning model and the data used for training are inferred. The model parameters are updated with the received feedback from the environment in reinforcement learning where no training data is involved. This ML method proceeds as prediction and evaluation cycles [30]. DL is defined as learning and improving by analysing algorithms on their own. It works with models such as artificial neural networks (ANN) and consists of a higher or deeper number of processing layers [31].

3. Methodology

Android was first released in 2008. A few years later, the security concerns were discussed with the increasing popularity of Android applications [2]. More attention was received towards applying ML for software security in the last five years because many researchers continuously identify and propose novel ML-based methods [9]. This review was conducted according to the Preferred Reporting Items for Systematic reviews and Meta-Analysis (PRISMA) model [32]. Based on the objective of this study, first we formulated several research questions (see Section 3.1). Next, a search strategy was defined to identify the conducted studies which can be used to answer our research questions. The database usage and inclusion and exclusion criteria were also defined at this stage. The study selection criteria were defined to identify the studies aiming to answer the formulated research questions as the third stage. The fourth stage is defined as data extraction and synthesis, which describes the usage of the collected studies to analyse for providing answers to the research questions. We reviewed threats to the validity of the review and the mechanism to reduce the bias and other factors that could have influenced the outcomes of this study as the last step of the review process.

3.1. Research Questions

This systematic review aims to answer the following research questions.

RQ1: What are the existing reviews conducted in ML/DL based models to detect Android malware and source code vulnerabilities?
RQ2: What are code/APK analysing methods that can be used in malware analysis?
RQ3: What are the ML/DL based methods that can be used to detect malware in Android?
RQ4: What are the accuracy, strengths, and limitations of the proposed models related to Android malware detection?
RQ5: Which techniques can be used to analyse Android source code to detect vulnerabilities?

3.2. Search Strategy

The search strategy involves the outline of the most relevant bibliographic sources and search terms. In this review, we have used several top research repositories as main sources to identify studies. They were ACM Digital Libraries, IEEEXplore Digital Library, Science Direct, Web of Science and Springer Link. Google Scholar, and Research Gate were also used to identify research studies published in some quality venues. The search string that we used to browse through research repositories contained the following search terms: (“android malware”) OR (“malware detection”) OR (“machine learning”) OR (“deep learning”) OR (“static analysis”) OR (“dynamic analysis”) OR (“hybrid analysis”) OR (“malware analysis”) OR (“android vulnerability analysis”) OR (“ML based malware detection”) OR (“DL based malware detection”).

3.3. Study Selection Criteria

Since mobile malware detection using ML techniques related trends increased from 2016, we limit our review to study related work from 2016 to May 2021. Initially through the research database search in the top research repositories, 109 research papers and from another sources 11 research papers were identified. From these 120 papers, 5 were excluded because of duplicate entries and another 5 were excluded because they were not available in public from those 110 articles. Due to data analysis issues and experiment issues in the given context, 4 articles were excluded though the full text is available. The remaining 106 articles were reviewed in this study. We performed the snowballing process [33], considering all the references presented in the retrieved papers and evaluating all the papers referencing the retrieved ones, which resulted in two additional relevant paper. We applied the same process as for the retrieved papers. The snowballing search was conducted in March 2021. Figure 2 shows a summary of the paper selection method for this systematic review.

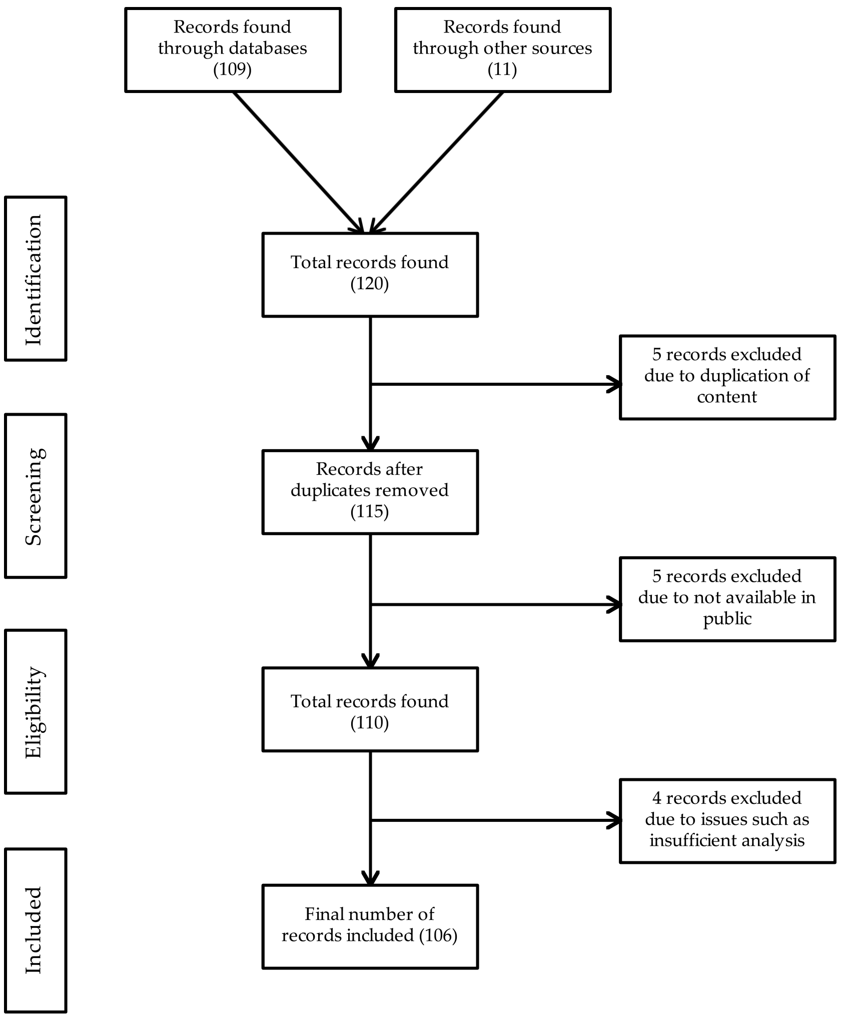
Records found through databases (109)
Records found through other sources (11)
Identification
Total records found (120)
5 records excluded due to duplication of content
Screening
Records after duplicates removed (115)
5 records excluded due to not available in public
Eligibility
Total records found (110)
4 records excluded due to issues such as insufficient analysis
Included
Final number of records included (106)

Figure 2. PRISMA method: collection of papers for the review.

3.4. Data Extraction and Synthesis

We extracted data from 9 studies to answer the RQ1, which is about the existing literature reviews related to Android malware detection using ML/DL models and Android vulnerability analysis. To map with RQ2, related studies were identified related to Android code/APK analysing techniques that can be used to analyse malware. The count for those studies was 22. To answer the RQ3 about ML/DL based techniques which can be used to detect malware, we extracted data from 18 different studies. Data from 36 research studies were extracted to find answers for the RQ4, which is about detection model accuracy, strengths, and weaknesses. The remaining 21 papers about Android source code vulnerability analysis and detection methods were used to answer the RQ5.

3.5. Threats to Validity of the Review

This review was conducted in a systematic approach explained above. We tried to minimise the bias and the other factors affecting the review study. Though we have conducted our review comprehensively, still there can be good papers which were not reviewed in this study since they are not available in the research repositories that we used. The period we were considering for the paper selection is from 2016 to May 2021, as the use of ML techniques for malware detection has increased significantly during this period due to recent advances in artificial intelligence. Therefore, if comprehensive studies were conducted before that, those studies were not captured in our work. When searching for the papers we considered the research papers written in the English language. Because of this limitation, our work may have overlooked some important works written in other languages such as Chinese, German, and Spanish.

4. Related Work

Previous reviews in [9,13,17,34–37] discussed various ML-based Android malware detection techniques and ways to improve Android security.

The review in [34] systematically reviewed the studies conducted in static analysis techniques used for Android applications from 2011 to 2015. The tools that can be used to perform Android code analysis using static analysis techniques were also summarised. Abstract representation, taint analysis, symbolic execution, program slicing, code instrumentation, and type/model checking were identified as fundamental analysis methods. Though this review correctly identified the most widely used approach to detect privacy and security related issues, the applicability of static analysis techniques for malware detection was not discussed. Apart from that, it did not take into account the recent research where novel analysis methods and malware detection methods were suggested. The study conducted in [35] provided a good systematic review mainly about static analysis techniques that can be used in Android malware detection. Four methods were identified as characteristic-based, opcode-based, program graph-based and symbolic execution-based. After that, it evaluated the capabilities of static analysis based Android malware detection methods on those four methods using the existing literature. The paper has identified ML and statistical models as possible methods by which Android malware can be identified. However, ML-based machine learning methods have not been thoroughly reviewed as the main focus is only on the static analysis techniques.

In [13], a survey was carried out using existing literature up to 2017 to identify malware detection techniques together with their advantages and disadvantages. Under static and dynamic analysis, they have grouped several approaches that can be used to identify Android malware. However, the analysis of this survey was not comprehensive as it focused on a limited number of studies. Based on the previous studies, a systematic review was conducted in [17]. According to it, there are five types of Android malware detection techniques. They are static detection, dynamic detection, hybrid detection, permission-based detection, and emulation based detection. They also summarised the reviewed work with the model accuracy of malware detection, but the approach of those studies was not discussed. The review conducted in [9] analysed several studies conducted until 2019 related to ML models which can be used to detect Android malware. The malware and APK analysis methods were not discussed in detail since the focus on identifying different ML models was the priority in this review. It is better to analyse the accuracies of the identified ML models. The novel ML/DL and other models which can be used to detect Android malware were also not in the focus of this review. The review in [36] provides a good analysis of static, dynamic, and hybrid detection techniques used in the existing research studies for Android malware detection. Along with that possibility of using machine learning models, several deep learning models are also discussed. However, this study did not comprehensively analyse the model accuracy of the machine learning methods for Android malware detection since this study focused more on discussing different malware detection approaches instead of considering the accuracy of those approaches. Hence, these works differ from our study.

In [37], a systematic review on DL-based methods for Android malware defence was discussed. Malware detection, malware family detection, repackaged/fake app detection, adversarial learning attacks and protections, and malicious behaviour analysis were identified as the malware defines objectives in this review together with the usage of DL models. Though they have identified the possible DL models, it is still better to analyse the accuracy and compare it with traditional ML methods and other hybrid approaches.

Apart from Android malware detection techniques, source code vulnerability analysis is also important to address security concerns in Android. The survey in [38] analysed several studies on ML-based and data mining approaches which can be used to identify software vulnerabilities until 2017. Though this survey provides a good analysis, they considered most of the research work in general software security. Therefore, the vulnerability analysis in Android code was not discussed. However, findings such as ML models' usage for vulnerability analysis are still beneficial for specific programming languages' related analysis.

However, several limitations have been identified in the above works, such as not covering recent proposals on ML methods to detect malware, narrow scopes, and lack of critical appraisals of suggested detection methods. The lack of a thorough analysis of ML/DL-based methods was also identified as a limitation of existing works. Android malware detection and Android code vulnerability analysis have a lot in common. ML methods used in one task can be customised for use in the other task. However, as per our understanding, there are no reviews that cover these two areas together. These shortcomings have been addressed in this work and therefore our work is unique.

5. Machine Learning to Detect Android Malware

Malware detection in Android can be performed in two ways; signature-based detection methods and behaviour-based detection methods [39]. The signature-based detection method is simple, efficient, and produces low false positives. The binary code of the application is compared with the signatures using a known malware database. However, there is no possibility to detect unknown malware using this method. Therefore, the behaviour-based/anomaly-based detection method is the most commonly used way. This method usually borrows techniques from machine learning and data science. Many research studies have been conducted to detect Android malware using traditional ML-based methods such as Decision Trees (DT) and Support Vector Machines (SVM) and novel DL-based models such as Deep Convolutional Neural Network (Deep-CNN) [40] and Generative adversarial networks [41]. These studies have shown that ML can be effectively utilised for malware detection in Android [9]. Most of these studies used datasets such as Drebin [42], Google Play [43], AndroZoo [44], AppChina [45], Tencent [46], YingYongBao [47], Contagio [48], Genome/MalGenome [49], VirusShare [50], IntelSecurity/MacAfee [51], MassVet [52], Android Malware Dataset (AMD) [53], APKPure [54], Anrdoid Permission Dataset [55], Andrototal [56], Wandoujia [57], Kaggle [58], CICMaldroid [59], AZ [60], and Github [61] to perform experiments and model training in their studies.

5.1. Static, Dynamic, and Hybrid Analysis

As mentioned earlier, analysing APKs to extract features is required to use some of the proposed ML techniques in the literature. To this end, three analysis techniques are identified as static, dynamic, and hybrid analysis method [62–64]. Static analysis can be performed by analysing the bytecode and source code (or re-engineered APK) instead of running it on a mobile device. Dynamic analysis detects malware by analysing the application while it is running in a simulated or real environment. However, there is a high chance of exposing the risks to a certain extent to the runtime environment in the dynamic analysis since malicious codes will be executed which can harm the environment. The hybrid analysis involves methods in both static and dynamic analysis.

Under the static analysis, four aspects were proposed [28] which are analysis techniques, sensitivity analysis, data structure, and code representation. Under the analysis techniques, Symbolic execution, taint analysis, program slicing, abstract interpretation, type checking, and code instrumentation were identified. For the sensitivity analysis, object, context, field, path, and flow were identified. For the data structure aspect, it is possible to list call graph (CG), Control Flow Graph (CFG), and Inter-Procedural Control Flow Graph (ICFG). Smali, Jimple, Wala-IR, Dex-Assembler, Java Byte code, or class were listed under the code representation aspect. Kernel, application, and emulator can be taken under inspection level aspect. Taint analysis and anomaly-based can be taken under the dynamic analysis approaches.

The feature extraction methods available in the static analysis consist of two types: Manifest Analysis and Code Analysis [65]. Features such as package name, permissions, intents, activities, services, and providers can be identified in Manifest Analysis. In the code analysis, features such as API calls, information flow, taint tracking, opcodes, native code, and cleartext analysis can be identified as possible features to extract. For the dynamic analysis, five feature extraction methods were identified. They were (1) Network traffic analysis for features like Uniform Resource Locators (URL), Internet Protocol (IP), Network protocols, certificates, and nonencrypted data, (2) Code instrumentation for features such as Java classes, intents, and network traffic, (3) System calls analysis, (4) System resources analysis for features such as processor, memory and battery usage, process reports, network usage, and (5) User interaction analysis for features such as buttons, icons, and actions/events. The study in [66] has explored the security of ML for Android malware detection techniques using a learning-based classifier with API calls extracted from converted smali files. Then a sophisticated secure learning method is proposed, which showed that it is possible to enhance the security of the system against a wide range of evasion attacks. This model is also applicable to anti-spam and fraud detection areas. This study can be further improved by exploring the possibilities of identifying attacks that can alter the training process.

5.2. Static Analysis with Machine Learning

Static analysis is the widely used mechanism for detecting Android malware. This is because malicious apps do not need to be installed on the device as this approach does not use the runtime environment [67].

5.2.1. Manifest Based Static Analysis with ML

Manifest based static analysis is a widely used static analysis technique. The model proposed in SigPID [68] discussed an Android permission-based malware detection mechanism. This model has identified only 22 permissions out of all the permissions listed in sample APKs that are significant by developing a three-level data purring method: permission ranking with negative rate, support based permission ranking, and permission mining with association rules. After that, the ML algorithms were employed to detect the malware. To this process, a binary format dataset of permissions, which was created using a database of malware and benign apps from Google Play was used. The support-vector machine (SVM) outperformed the other studied ML algorithms (Naïve Bayes (NB) and

(DT)) with over 90% accuracy. For the permission-based static analysis, this work was conducted comprehensively. However, it is better to check the other variables which are affecting the malware apart from permissions.

A malware detection method using Android manifest permission analysing was proposed in [69] with the use of static analyser and decompilation support of APKTool for the APK to code level extraction. AndroZoo repository was used as the dataset to train four different ML algorithms. Random Forest (RF), SVM, NB, and K-Means were used to perform the model validity process, and RF produced the highest accuracy for this model with 82.5% precision and 81.5% recall. However, the accuracy of this model is comparatively low with the other studies conducted in the same area. The close reason for that would be that this approach compares the permissions only.

The proposed work in [70] checked the possibility of using reduced dimension vector generating for malware detection. Based on that, malware detection using ML models with permission-based static analysis was performed. In the feature selection stage of this approach, the model removed the unnecessary features using a linear regression-based feature selection approach. Therefore, the classification model can run in real-time since the training time was decreased, with an accuracy of over 96%. The Multi-Layer Perception Model (MLP) algorithm outperformed NB, Linear Regression, k-nearest neighbors (KNN), C4.5, RF and Sequential Minimal Optimization (SMO). It is better to focus on hypermeter selections to also increase the performance of the classification. The model proposed in [71] performed a static analysis on Android apps. Android permissions and intents were used as the basic static features of malware classification while URLs, Emails, and IPs were used as the basic dynamic features. Initially, the APK files were decompiled using ApkTool. The extractor module of this extracted different types of information related to malware. After extracting the data through disassembling the dex files, the data were kept in a text files and they were used to create the feature vector. Then the ML algorithms RF, NB, Gradient Boosting (GB), and Ada Boosting (AB) were used to train and test the malware detection model with the usage of Drebin dataset and Google Play Store. After performing ML training and testing part for each of permission, intent, and network features individually it has identified that the above ML algorithms were performing with different accuracies. For permissions RF performed well with 0.98 precision and recall, for intents NB performed well with 0.92 precision and 0.93 recall, and for network both RF and AB performed similarly well with 0.97 precision and recall. Though this research concluded with such accuracies for malware detection it is still lacking the study of some other features like API calls, etc.

Android malware detection technique using feature weighting with join optimisation of weight mapping and classifier parameters model is proposed in JOWMDroid Framework in [72]. This model is a static analysis-based technique that selected a certain number of features out of the extracted features from the app which were related to malware detection. This process was done by decompiling the APK to manifest and class.dex files and prepared a binary feature matrix. Initial weight was calculated using Random Forest, SVM, Logistic regression (LR), and KNN ML models. Weight machine functions were designed to map the initial weight with final weights. As the last step, classifiers and weight mapping function parameters were jointly optimised by the Differential Evolutional algorithm. Drebin, AMD, Google Play, and APKPure datasets were used to train the model. Finally, it is identified that among weight unaware classifiers, RF performed better with 95.25% accuracy and for weight-aware classifiers, KNN and MLP performed better. However, with the integration of this JOWM-IO method, SVM and LR beat the RF with over 96% accuracy. If the correlation between features is also considered, the model accuracy for detecting malware will increase.

Table 1 comparatively summarises the above research studies related to manifest analysis based methods.

Table 1. Manifest based static Analysis with ML.

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML Algorithms/ Models	Selected ML Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2018	[68]	Developing 3 level data purring method and applying ML models with SigPID	Manifest Analysis for Permissions	Google Play	NB, DT, SVM	SVM	90%	High effectiveness and accuracy	Considered only the permission analysis which may lead to omit other important analysis aspects
2021	[69]	Analysing permission and training the model with identified ML algorithm	Manifest Analysis for Permissions	Google Play, AndroZoo, AppChina	RF, SVM, Gaussian NB, K-Means,	RF	81.5%	The model was trained with comparatively different datasets	Did not consider other static analysis features such as OpCode, API calls, etc.
2021	[70]	Reducing dimension vector generation and based on that perform malware detection using ML models	Manifest Analysis for permissions	AMD, APKPure	MLP, NB, Linear Regression, KNN, C.4.5, RF, SMO	MLP	96%	Efficiency, applicability and understandability are ensured	Hyper-parameter selections are not made in the use
2021	[71]	Selecting feature using dimensionality reduction algorithms and using Info Gain method	Manifest Analysis for permissions and intents	Drebin, Google Play	RF, NB, GB, AB	RF, NB, AB	RF-98%, NB-92%, AB-97%	Analysed the features as individual components and not as a whole	Did not consider about other features such as API calls, Opcode etc.
2021	[72]	Feature weighting with join optimisation of weight mapping with proposed JOWMDroid framework	Manifest Analysis for permission, Intents, Activities and Services	Drebin, AMD, Google Play APKPure	RF, SVM, LR, KNN	JOWM-IO method with SVM and LR	96%	Improved accuracy and efficiency	Correlation between features were not considered

5.2.2. Code Based Static Analysis with ML

Code based analysis is the other way of performing the static analysis to detect Android malware with ML. The model proposed in TinyDroid [39] analysed the latest malware listed in the Drebin dataset. Instruction simplification and ML are used in the model. Using the decompiled DEX files by converting APK to smali codes, the opcode sequence was abstracted. Then using that, features were extracted through N-gram and integrated with the exemplar selection method. In the exemplar selection method, for intrusion detection, a good representative of data was generated through a clustering algorithm, Affinity Propagation (AP). This is because in AP, the number of clusters determination or estimation is not required before running the application. Then the generated 2,3, and 4-gram sequences were fed into SVM, KNN, RF, and NB ML classifiers. RF algorithm was identified as the optimal algorithm for this scenario with 0.915 True Positive Rate, 0.106 False Positive Rate, 0.876 Precision, and 0.915 Recall for 2-gram sequence. High accuracy rates for the other 3 and 4-grams were also achieved compared to the studied ML algorithms. However, the proposed method still has issues such as using the malware samples taken only from few research studies and some organisation and lack of metamorphic malware samples. Therefore, some malware could remain undetected.

The approach proposed in [73] used the Drebin dataset with 5560 malware samples along with 361 malware from the Contagio dataset and 5900 benign apps from Google Play to propose another approach to detect malware by analysing API calls used in operand sequences. For the malware prediction model, the package level details were extracted from the API calls. The package n-grams were extracted from the package sequence, which represents application behaviour. Then they were combined with DT, RF, KNN, and NB ML algorithms to build a predictive model in this study and concluded that the RF algorithm performed with an accuracy of 86.89% after training the model on 2415 package n-grams. It is better to consider other information which contains in operands since it might affect the overall model. The relationship of system functions, sensitive permissions, and sensitive APIs were analysed initially in Anrdoidect [74]. A combination of system functions was used to describe the application behaviours and construct eigenvectors using the dynamic analysis technique. Based on the eigenvectors, effective methodologies of malware detection were compared along with the NB, J48 DT, and application functions decision algorithm and identified that the application functions' decision algorithm outperformed the others. There are still some improvements to be performed to this approach.

In MaMaDroid [75] model, API calls performed by apps were abstracted using static analysis techniques to classes, packages, or families. Then to determine the call graph of apps as Markov chain, the sequence of API calls was obtained. Then using ML algorithms, classification was performed using RF, KNN, and SVM and it was identified that RF had the highest accuracy among these three. However, in this method, dynamic analysis was not considered. The dynamic analysis is useful for an API calls analysis in a runtime environment to detect malicious applications.

Android malware detection approach using the method-level correlation relationship of application's abstracted API calls was discussed in [76]. Initially, the source codes of Android applications were split into methods, and abstracted API calls were kept. After that, the confidence of association rules between those calls was calculated. This approach provided behavioural semantic of the application. Then SVM, KNN, and RF algorithms were used to identify the behavioural patterns of the apps towards classifying as benign or malicious. Drebin and AMD datasets were used for this, and 96% accuracy was received with the RF algorithm. This method does not address the problems such as dynamic loading, native codes, encryptions, etc. though it has such high accuracy. If the dynamic analysis methods are also used, the accuracy of this model will increase to a further high level.

The model named SMART in [77] proposed a semantic model of Android malware based on Deterministic Symbolic Automation (DSA) to comprehend, detect, and classify malware. This approach identified 4583 malware that were not identified by leading anti-malware tools. Two main stages were included in this approach; malicious behaviour learning and malware detection and classification. In Stage 1, the model identified semantic clones among malware, and semantic models were constructed based on that. Then malicious features were extracted from DSA, and ML techniques were used to detect malware in Stage 2 after performing static analysing activities with bytecode analysis. Random Forest achieved the best classification results of 97% accuracy, and AB, C45, NB, and Linear SVM provided lower accuracy. Therefore, this work identified that DSA is possible to use for malware detection. DroidChain [78] proposed a static analysis model with behaviour chain model. The malware detection problem was transformed to a matrix model using the Wxshal algorithm to further analyse this approach. Privacy leakage, SMS financial charges, malware installation, and privilege escalation were proposed as malware models in this study using the behaviour chain model. In the static analysis part, using APKTool and DroidChain, Smali codes were extracted. Then the API call graph was generated using the Androguard [79] tool. After that, the incidence matrix was built, and the accessibility of the matrix to detect malware was calculated. The average accuracy of this model was 83%. This method can be improved to detect malware more accurately and efficiently by considering other static analysis features such as code analysis, permission analysis, etc.

The study conducted in [80] discussed testing malware detection techniques based on opcode sequence and API call sequence. The Hidden Markov Model (HMM) was trained in this and detection rates for models based on static, dynamic, and hybrid approaches were identified and it was concluded that the hybrid approaches are highly effective without performing static or dynamic analysis alone.

Tables 2 and 3 comparatively summarise the above research studies related to code analysis based methods, while Table 2 listed studies with model accuracy below 90% and Table 3 listed studies with model accuracy above 90%.

Table 2. Code based static Analysis with ML (Model Accuracy is below 90%).

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML Algorithms/ Models	Selected ML Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2016	[78]	Transforming malware detection problem to matrix model using Wxshall algo and extracting Smali codes and generated the API call graph using Androguard	Code analysis for API Calls and code instrumentation for network traffic	MalGenome	Custom build ML based Wxshall algorithm, Wxshall extended algorithm	Wxshall extended algorithm	87.75%	Few false alarms	Required to expand the behaviour model and improve the efficiency
2017	[74]	Using the combination of system functions to describe the application behaviours and constructing eigenvectors and then using Androidetect	Code analysis for API calls and Opcodes	Google Play	NB, J48 DT, Application functions decision algorithm	Application functions decision algorithm	90%	Can identify the instantaneous attacks. Can judge the source of the detected abnormal behaviour High performance in model execution	Did not consider some important static analysis features such as OpCode, API calls, etc.
2018	[39]	Using TinyDroid framework, n-Gram methods after getting the Opcode sequence from .smali after decompiling .dex	Code Analysis for Opcode	Drebin	NLP, SVM, KNN, NB, RF, AP	RF and AP with TinyDroid	87.6%	Lightweight static detection system High performance in classification and detection	Malware samples were taken only from few research studies and some organisations which lack metamorphic malware samples
2018	[73]	Analysing Package level information extracted from API calls using decompiled Smali files	Code Analysis for API calls and Information flow	Drebin, Contagio, Google Play	DT, RF, KNN, NB	RF	86.89%	Model performs well even when the length of the sequence is short	Other information contained in operands were not considered which affect to the overall model

Table 3. Code based static Analysis with ML (Model Accuracy is above 90%).

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML Algorithms/ Models	Selected ML Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2016	[77]	Using Deterministic Symbolic Automaton and Semantic Modelling of Android Attack	Code Analysis for Opcode/Byte code	Drebin	AB, C4.5, NB, LinearSVM, RF	RF	97%	Use a combined approach of ML and DSA inclusion	Unable to detect new malware patterns since this will not perform complete static analysis
2017	[80]	Training Hidden Markov Models and comparing detection rates for models based on static data, dynamic data, and hybrid approaches	Code analysis for API calls and Opcode in static analysis and System call analysis	Harebot, Security Shield, Smart HDD, Winwebsec, Zbot, ZeroAccess	HMM	HMM	90.51%	Check the difference approaches available to detect ML	Did not consider other ML algorithms or other important features
2019	[75]	Determining the apps call graphs as Markov chain Then obtaining API call sequences and using ML models with MaMaDroid	Code Analysis for API calls	Drebin, oldbenign	RF, KNN, SVM	RF	94%	the system is trained on older samples and evaluated over newer ones	Requires a high memory to perform classification
2019	[76]	Calculating confidence of association rules between abstracted API calls which provides behavioural semantic of the app	Code Analysis for API calls	Drebin, AMD	SVM, KNN, RF	RF	96%	Efficient feature extraction process Better stability of the system	Did not address the cases such as dynamic loading, native codes, encryption, etc.

5.2.3. Both Manifest and Code Based Static Analysis with ML

Some studies used both manifest and code based static analysis approaches to detect Android malware with ML. The implemented model in WaffleDetector [81], a static analysis approach to detect malware, was proposed by using a set of Android program features, sensitive permissions, and API calls with the utilization of Extreme Learning Machine (ELM). Tencent, YingYongBao, and Contagio datasets were used to train the algorithms. This method outperformed traditional binary classifiers (DT, Neural Network, SVM, and NB) with 97.06% accuracy. This approach still needs a few improvements, such as refining the combination of permissions and API calls.

The study conducted in [82] studied repackaged apps. The malware was identified from these repackaged apps with code-heterogeneity features. The codes of the apps were partitioned into subsets. Then the subsets were classified based on their behavioural features with Smalicode. Compared to the other nonpartitioning methods, this approach provides high accuracy with a False Negative Rate (FNR) of 0.35% and a False Positive Rate (FPR) of 2.97%. This method also used some Ensemble Learning mechanisms. It is better if the method improves the code heterogeneity mechanisms by using context and flow sensitivity.

Using the Drebin dataset, a method to detect Android malware using static analysis is discussed in [83]. Using this method with high accuracy of 98.7%, it was possible to detect malware using a sample of 10,865 applications. In this method, initially, the APK file was downloaded using the extracted download link from the APKPure website by using web mining techniques. Then the APK content was extracted using Apktool and generated the AndroidManifest.xml and classes.dex files. The application features were extracted from AndroidManifest.xml using the AAPT utility while decompiling classes.dex into a jar file using the dex2jar tool. Then the number of lines of code feature was extracted after extracting the java source files from the jar file using the jd-cmd tool. This static analysis approach was evaluated using ten different ML algorithms; KNN, SVM, Bayes Net, NB, LR, J48, RT, RF, AB, and BA. Out of them RF with 1000 decision trees outperformed the others with 0.987 precision, recall, and F-measure [83]. Though the model has high accuracy, it is better to study behavioural analysis of app behaviour by performing dynamic analysis.

In RanDroid [84] model, already classified malicious and benign apps were used to train the SVM, DT, RF, and NB ML algorithms. Initially, the APK files were decompiled using Androguard (a python-ased tool) [79]. Then the required features of permission, API calls, is_crypto_code, is_dynamic_code, is_native_code, is_reflection_code, is_database were extracted and transformed into binary vectors. Then it was trained using ML algorithm and identified that the DT was the most suitable algorithm for this static analysis approach with 97.7% accuracy. However, in this study, broadcast receivers, filtered intend, Control Flow Graph analysis, deep native code analysis, and dynamic analysis are not considered; they are identified as drawbacks.

In [85] a model named TFDroid has been proposed, which is a ML based malware detection by topics and sensitive data flow analysis using SVM with an accuracy of 93.7%. FlowDroid is a static analysis tool that was used in this approach to extract data flow in benign and malicious apps. The permission granularity was transformed using the data flow features. After that, a classifier was implemented for each category and performed the validation process. Google Play and Drebin datasets were used to train the model in this study. It is better to check the other possible ML algorithms' performance also. Since this study is related to data flow, it is better to perform dynamic analysis and introduce a hybrid model to increase the accuracy of detecting Android malware.

The DroidEnsemble [86] analyses the static behaviours of Android apps and builds a model to detect Android malware. In this approach, static features such as permissions, hardware features, filter intents, API calls, code patterns, and structural features of function call graphs of the application were extracted. Then after creating the binary vector, SVM, KNN, RF, and ML algorithms were performed to evaluate the performance of the features and their ensemble. The proposed methodology achieved detection accuracy of 95.8% and 90.68%, respectively, for static features and structural features. For ensemble of both types, the accuracy was increased to 98.4% with SVM. Sting features like API calls and structural features like function call graphs can be checked with dynamic analysis. Therefore, in this model, the malware detection accuracy would be increased when both static and dynamic analysis were integrated.

Table 4 comparatively summarised the above research studies related to both manifest and code based static analysis methods with ML.

Table 4. Both Manifest and Code based Static Analysis with ML.

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML Algorithms/ Models	Selected ML Algorithms /Models	Model Accuracy	Strengths	Limitations/Drawbacks
2017	[81]	Using customized method named Waffle Director	Manifest Analysis for Sensitive permissions and API calls	Tencent, YingYongBao, Contagio	DT, Neural Network, SVM, NB, ELM	ELM	97.06%	Fast Learning speed and Minimal human intervention	Combination of permissions and API calls are not refined
2017	[82]	Using a code-heterogeneity-analysis framework to classify Android repackaged malware by Smali code intermediate representation	Manifest Analysis for Intents, Permissions and API calls	Genome, VirusShare, Benign App	RF, KNN, DT, SVM	RF with custom model proposed	FNR-0.35%, FPR-2.96%	Provide in-depth and fine-grained behavioural analysis and classification on programs	Detection issues can happen when the malware use coding techniques like reflection and cannot handle if the encryption techniques used in DEX
2018	[84]	Extracting features and transforming into binary vectors and training using ML with RanDroid Framework	Manifest Analysis for Permissions Code Analysis for API calls, opcode and native calls	Drebin	SVM, DT, RF NBs	DT	97.7%	Highly accurate to analyse permission, API calls, opcode an native calls toward malware detection	Broadcast receivers, filtered intend, Control Flow Graph analysis, deep native code analysis were not considered
2018	[86]	Creating the binary vector, apply ML models, evaluate performance of the features and their ensemble using DroidEnsemble	Manifest analysis for permissions, code analysis for API calls and system calls analysis	Google Play, AnZhi, LenovoMM, Wandoujia	SVM, KNN, RF	SVM	98.4%	Characterises the static behaviours of apps with ensemble of string and structural features.	Mechanism will fail if the malware contains encryption, anti-disassembly, or kernel-level features to evade the detection
2019	[83]	Extracting applications features from manifest while decompiling classes.dex into jar file and applying ML models	Manifest Analysis for permissions, activities and Code Analysis for Opcode	Drebin, playstore, Genome	KNN, SVM, BayesNet, NB, LR, J48, RT, RF, AB	RF with 1000 decision trees	98.7%	High efficiency, Lightweight analysis and fully automated approach	Did not consider about the API calls and other important features when analysing the DEX.
2019	[85]	Using FlowDroid for static analysis and proposing TFDroid framework to detect malware using sensitive data flow analysis	Manifest Analysis for permission and Code Analysis for information flow	Drebin, Google Play	SVM	SVM	93.7%	Analysed the functions of applications by their descriptions to check the data flow.	Did not consider the improving clustering techniques and applicability of other ML models

5.3. *Dynamic Analysis with Machine Learning*

The second analysis approach is dynamic analysis. Using this approach it is possible to detect malware with ML after running the application in a runtime environment. Android Malware detection using a network-based approach was introduced in [87]. In this approach, a detection application was developed. It contained three modules: network traces collection, network feature extraction, and detection. In the traces collection module, network activities of running applications were monitored and recorded the network traces periodically. The features extraction module extracted features of the network used by the applications. Those features were Domain Name System (DNS) based features, HyperText Transfer Protocol (HTTP) based features, Origin destination based features, and Transmission Control Protocol (TCP) based features. DT, LR, KNN, Bayes Network, and RF algorithm were used in the detection module. The RF algorithm provided the highest accuracy (98.7%) among them. However, this approach used network-based analysis. If the malware apps were using encrypted transfers, the malware detection accuracy would decrease. Therefore, the model also should consider such factors.

The proposed model in 6th Sense [88], using Markov Chain, NB, Logistic Model Tree (LMT) to detect malware using dynamic analysis is based on sensors available in a mobile device. A context-aware intrusion detection system is studied in this approach by collecting and observing changes in sensor data. This step happened when the applications were performing activities that enhanced security. This model distinguishes malware and benign applications. Three types of malware activities (triggering, leaking information, and stealing data) were identified using this approach via sensors available in the device. The collected data was divided as 75% for training and 25% for testing. For the Markov Chain-based detection technique, a training dataset was used to compute the state transitions and build a transition matrix. A training dataset was used with NB to determine the sensor condition changing frequency. For the other ML algorithms, all the data were defined as benign and malware. In this study, LMT outperformed others with 99.3% precision and 99.98% recall. Though this study is a comprehensive one, it is better if the tradeoffs such as frequency accuracy, battery frequency, etc. are considered.

The proposed method in [89] discussed dynamic analysis-based techniques which extract a set of dynamic permissions from APKs in different sources and run them in an emulator. Then it evaluates the model using NB, RF, Simple Logistic, DT, and K-Star ML models. After that, it is identified that Simple Logistic performs well with 0.997 precision and 0.996 recall. Some issues were in the dataset used in this model. For example, some benign and malicious apps were using the same permissions, and some apps crashed when running the application in an emulator. Therefore, if the dataset is fine-tuned more before use, this model provides even more accuracy.

In [90], a framework called Service Monitor was proposed, which is a lightweight host-based detection system that can detect malware on devices. This framework was built using dynamic analysis. Service Monitor monitored the way of requesting system services to create the Markov Chain Model. The Markov Chain is used as a feature vector to perform the classification tasks with ML algorithms: RF, KNN, and SVM. The RF method performed well with an accuracy of 96.7% after training the model with AndroZoo, Drebin, and Malware Genome datasets. Some benign apps also requested the system services in a similar way to malware. Therefore, this could lead to some misclassification of this model. To avoid that and enhance the classification accuracy, signature-based verification to the Service Monitor can be applied.

A mechanism named DATDroid was proposed in [91] which is a dynamic analysis based malware detection technique with an overall accuracy of 91.7% with 0.931 precision and 0.9 recall values with RF ML algorithm. As the initial stage, feature extraction was performed by collecting system calls, recording CPU and memory usage, and recording network packet transferring. Then in the feature selection stage, Gain Ratio Attribute Evaluator was applied. After that, the model training and validation were performed as the next stage to identify malicious and benign applications using APKPure and Genome Project datasets. In addition to the features studied in this, there can be an impact from features like HTTP, DNS, TCP/IP, and memory usage patterns towards identifying malware which should be discussed.

In [92], a framework which is named as MEGDroid, using the dynamic analysis was proposed to improve the event generation process in Android malware detection. In this method, it automatically extracted and represented information related to malware as a domain-specific model. Decompilation, model discovery, integration and transformation, analysis and transformation, and event production were the steps included in this model. The model was then used to analyse malware after training with the AMD dataset. This model extracted every possible event source from malware code and was developed as an Eclipse plugin. Based on the results, MEGDroid provides better coverage in malware detection through generating UI, whereas system events and monitoring the system calls are lacking in this approach.

Table 5 comparatively summarises the above research studies related to dynamic analysis based methods.

Table 5. Dynamic analysis based malware detection approaches.

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML Algorithms/ Models	Selected ML Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2017	[87]	Extracting the DNS, HTTP, TCP, Origin based features of the network used by apps	Network traffic analysis for network protocols	Genome	DT,LR, KNN, Bayes Network, RF	RF	98.7%	Work with different OS versions, Detect unknown malware, and infected apps	If the malware apps using encrypted, not possible to detect malware properly
2017	[88]	Using Markov Chain-based detection technique, to compute the state transitions and to build transition matrix with 6thSense	System resources analysis for process reports and sensors	Google Play	Markov Chain, NB, LMT	LMT	95%	Highly effective and efficient at detecting sensor-based attacks while yielding minimal overhead	Tradeoffs such as frequency accuracy, battery frequency are not discussed which can affect the malware detection accuracy
2017	[89]	Using Dynamic based permission analysis using a run-time and detect malware using ML calculate the accuracy	Code instrumentation analysis Java classes and dynamic permissions	Pvsingh, Android Botnet, DroidKin	NB, RF, Simple Logistic, DT K-Star	Simple Logistic	99.7%	High Accuracy	Need to address the app crashing issue in the selected emulators in dynamic analysis
2019	[90]	Using dynamically tracks execution behaviours of applications and using ServiceMonitor framework	System call analysis	AndroZoo, Drebin and Malware Genome	RF, KNN, SVM	RF	96.7%	High accuracy and high efficiency	Not detecting difference in some system calls of malware and benign apps since signature based verification was not applied
2020	[91]	Extracting the features and permissions from Android app. Performing feature selection and proceed to classification with DATDroid	System call analysis, Code instrumentation for network traffic analysis and System resources analysis	APKPure, Genome	RF, SVM	RF	91.7%	High efficiency	Impact from features like HTTP, DNS, TCP/IP patterns are not considered
2021	[92]	Using decompilation, model discovery, integration and transformation, analysis and transformation, event production	Code instrumentation for java classes, intents	AMD	ML algorithms used in MEGDroid, Monkey, Droidbot	MEGDroid	91.6%	Considerably increases the number of triggered malicious payloads and execution code coverage	System calls are not monitored

5.4. Hybrid Analysis with Machine Learning

Hybrid analysis is the third approach which can be used in ML-based Android malware detection. The review in [93] identified three approaches of malware detection, which are the signature-based, anomaly-based, and topic modelling based approaches. ML algorithms such as DT, J48, RF, KNN, KMeans, and SVM can be applied to all these approaches. Signature-based malware was detected using ML algorithms after the feature extraction process. After the feature extraction, sensitive API calls were also analysed before applying ML algorithms. Documents were collected such as reviews, user documents, and app descriptions before following a similar approach as the signature-based method, initially in the topic modelling approach. It was identified that the behavioural based approach is better than the signature-based approach. If the topic modelling is combined with that approach, it was possible to achieve good results. The hybrid analysis method is created when the dynamic analysis method is integrated with the static analysis method. According to this study, the SVM classifier with the hybrid analysis method performed better than the other ML algorithms.

The model proposed in [94] discussed a methodology of using ML algorithms with static analysis and dynamic analysis. In the static analysis approach, malicious and benign applications' manifest data were taken as JSON files from MalGenome and Kaggale datasets to train the ML model. The trending apps were taken from well-known app stores. Androguard [79] was used to extract information from the APK files. After reverse engineering, decompiling, testing, and training with SVM, LR, KNN based ML models, a JSON file was prepared. According to this model, LR was identified as the most suitable ML algorithm, which has 81.03% accuracy. Many improvements are required to the proposed static analysis model since comparatively this has a low accuracy. However, the proposed dynamic analysis approach outperformed the static analysis approach with high accuracy of 93% of both precision and recall over the RF. In this approach, Droidbox was used to run APKs obtained from MalGenome and Android Wave Lock in a sandbox environment. Then a CSV file is obtained after converting the JSON file obtained by analysing the APK and after that the key features are extracted. As the last step, DT, RF, SVM, KNN, and LR

ML algorithms were used with extracted key features. Then accuracy and results were checked and the particular app was labelled as malware or benign. It would be better if this study explored the possibilities of using other ML algorithms also.

In [95], authors conducted an experiment using various ML technologies to analyse the relative effectiveness of the static and dynamic analysis method towards detecting malware. This study used the Drebin dataset and a custom dataset to train the ML algorithm to classify malware and benign apps. Altogether the whole dataset contains 103 malware and 97 benign apps. For the static analysis, the APK files were reverse-engineered by a tool available in Virustotal and extracted the permissions using a custom XML parser. Then binary feature vectors and permission vectors were created, and ML algorithms were applied. For dynamic analysis, applications were executed on separated Android Virtual Devices (AVDs). System calls and their frequencies were traced using the MonkeyRunner tool since the frequency representation of system calls contained behavioural information on apps. Usually, malware has higher frequencies compared to benign apps. After that, a feature vector of system calls was created, and ML algorithms were applied. The RF, J.48, Naïve Bayes, Simple Logistic, BayesNet Augmented Naïve Bayes (TAN), BayesNet K2, Instance Based Learner (IBk), SMO PolyKernel, and SMO NPolyKernel algorithms were used for both static and dynamic analysis. The best results of 0.96 for static analysis and 0.88 for dynamic analysis were achieved when RF with 100 trees was used. Permissions extracted from the AndroidManifest.xml file were considered for static analysis, and system calls extracted from the runtime were considered in the dynamic analysis.

The model proposed in [96] explained a hybrid analysis process to detect malware using ML algorithms with the accuracy of 80% when using the permissions analysis in static analysis approach and 60% accuracy when analysing by system calls. Malware samples were collected using a honeypot and search repositories such as Androditotal to train the model. However, this study lacks the consideration of other features' which affect malware detection that should also be considered to achieve a high accuracy model.

In [97], the model proposed a hybrid analysis-based efficient mechanism for Android malware detection, which used the malware genome dataset and the Drebin dataset to train the ML and DL models in the static analysis approach. CICMalDroid dataset for the dynamic analysis approach and 261 combined features were extracted for the hybrid analysis. To increase the performance, this model used dimension reduction using Principal Component Analysis (PCA). SVM, KNN, RF, DT, NB, MLP, and GB were used to train and test the model. Out of these ML/DL algorithms, GB outperformed the others in terms of accuracy (96.35%), but it took a comparatively long training time. Forty-six features from dynamic analysis results were also analysed. After performing combined hybrid analysis, GB again performed well with an accuracy of 99.36% and efficiency compared to the Random Forest and MLP. It is better to study the runtime environment and configuration more because this does not cover some areas.

The model described in [98] proposed a Tree TAN based hybrid malware detection mechanism by considering both static and dynamic features such as API calls, permissions, and system calls. LR algorithms were trained for these three features. Drebin, AMD, AZ, Github, and GP datasets were used in this and modelled the output relationships as a TAN to detect if the given app is malicious or benign with an accuracy of 0.97. There is a possibility of some malware remaining undetected from the model, which can be reduced using Reinforcement Learning techniques.

Tables 6 and 7 comparatively summarise the above research studies related to hybrid analysis based methods, where Table 6 listed studies with model accuracy below 90% and Table 7 listed studies with model accuracy above 90%.

Table 6. Hybrid analysis based malware detection approaches (model accuracy is below 90% or overall accuracy is not available).

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML algorithms/ Models	Selected ML algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2017	[96]	Using a set of Python and Bash scripts which automated the analysis of the Android data.	Manifest analysis for permissions and System call analysis for dynamic analysis	Andrototal	NB, DT	DT	80%	Model execution is efficient	Consider system call appearance rather than frequency and Lower number of samples used to train
2018	[95]	Using Binary feature vector and permission vector datasets were created using the analysis techniques and was used with the ML algorithms	Manifest analysis for permissions and system call analysis	Drebin	RF, J.48, NB, Simple Logistic, BayesNet TAN, BayesNet K2, SMO PolyKernel, IBK, SMO NPolyKernel	RF	Static-96%, Dynamic-88%	Compared with several ML algorithms	Accuracy depends on the 3rd party tool (Monkey runner) used to collect features.
2019	[94]	Preparing a JSON file after reverse engineering, decompiling, and analysing the APK by running in a sandbox environment and then extracting the key features and applied ML	Manifest analysis for permissions, code analysis for API calls and System call analysis	MalGenome, Kaggle, Androguard [79]	SVM, LR, KNN, RF	LR for static analysis and RF for dynamic analysis	Static-81.03%, Dynamic-93%	Dynamic analysis performed was better than the static analysis approach in terms of detection accuracy	Did not perform a proper hybrid analysis approach to increase the overall accuracy

Table 7. Hybrid analysis based malware detection approaches (model accuracy is above 90%).

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML Algorithms/ Models	Selected ML Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2017	[99]	Using import term extraction, clustering and applying genetic algorithm with MOCODroid	Code analysis for API calls and information flow and system call analysis	Virus-total, Google Play	Genatic algorithm, Multiobjective evolutionary algorithm	Multiobjective evolutionary classifier	95.15%	Possible to avoid the effects of the concealment strategies	Did not consider about other clustering methods.
2020	[97]	Extracted 261 combined features of the hybrid analysis with using the support of datasets and performed the ML/DL models	Manifest analysis for permissions and system call analysis	MalGenome, Drebin, CICMal-Droid	SVM, KNN, RF, DT, NB, MLP, GB	GB	99.36%	Hybrid analysis is having higher accuracy comparing to static analysis and dynamic analysis individually	Runtime environment and configuration is not considered
2020	[98]	Using Conditional dependencies among relevant static and dynamic features. Then trained ridge regularised LR classifiers and modelled their output relationships as a TAN	Manifest analysis for permissions, code analysis for API calls and system call analysis	Drebin, AMD, AZ, Github, GP	TAN	TAN	97%	Highly accurate	Possibility of some malwares remain undetected
2021	[100]	Using exploit static, dynamic, and visual features of apps to predict the malicious apps using information fusion and applied Case Based Reasoning (CBR)	Manifest analysis for permissions and System call analysis	Drebin	CBR, SVM, DT	CBR	95%	Require limited memory and processing capabilities	Require to present the knowledge representation to address some limitations

5.5. Use of Deep Learning Based Methods

It is possible to use deep learning techniques also for detecting Android malware. In MLDroid, a web-based Android malware detection framework [101] was proposed by performing dynamic analysis. In this work, ML and DL methods were used with an overall 98.8% malware detection rate.

The model proposed in [102] disused a method to detect malware using a semantic-based DL approach and implemented a tool called DeepRefiner. This approach used the Long Short Term Memory (LSTM) on the semantic structure of Android bytecode with two layers of detection and validation. This method used the LSTM over Recurrent Neural Network (RNN) since RNN contains gradient vanish problem. Using this approach with an accuracy of 97.4% and a false positive rate of 2.54%, it was possible to detect malware. It was efficient and accurate compared with the traditional approaches. Since this approach uses the static analysis approach, some limitations can arise based on the runtime environment, which can be identified if this model uses the hybrid analysis approach.

MOCDroid [99] model discussed a multiobjective evolutionary classifier to detect malware in Android. It combined multiobjective optimisation with clustering to generate a classifier using third-party call group behaviours. This method produced an accuracy of 95.15%. Import term extraction, clustering, and applying a genetic algorithm were the three steps included in this process. Initially, the DEX files were uncompressed from the APK after using the decompression tool, and Java codes were obtained using the JADX tool [103]. Then the document term matrix was transformed. As the next step, K-Means clustering was applied since it was identified as the highest accuracy model for this, and the genetic algorithm was also applied. The results were compared with a random set of 10,000 benign and malicious apps with different antivirus engines. It is possible to consider other clustering methods to improve the accuracy of this method.

The work proposed in [104] discussed a method to detect Android malware using a deep convolutional neural network (CNN). Raw opcode sequence from disassembled Smali program was analysed using static analysers to classify the malware. The advantage of this method is automatically learning the feature indicative of malware. This work was inspired by n-gram based methods. To train the models Android Malware Genome project dataset [49] and Intel Security/MacAfee Lab dataset were used. The classification system of this provides 0.87 precision and recall accuracies. The accuracy of the malware detection can be increased when the dynamic analysis is also performed.

A deep learning-based static analysis approach was experimented with an accuracy of 99.9% and with an F1-score of 0.996 in [105]. This approach used a dataset of over 1.8 million Android apps. The attributes of malware were detected through vectorised opcode extracted from the bytecode of the APKs with one-hot encoding. After performing experiments on Recurrent Neural Networks, Long Short Term Memory Networks, Neural Networks, Deep Convents, and Diabolo Network models, it was identified that Bidirectional Long Short-Term Memory (BiLSTMs) is the best model for this approach. It is better to analyse the complete byte code using static analysis and check the app behaviour with dynamic analysis to build a more comprehensive malware detection tool based on deep learning techniques.

The DL-Droid framework based on deep learning techniques [106] proposed a new way of detecting Android malware with dynamic analysis techniques. This approach was having a detection rate of 97.8% by only including dynamic features. When the static features were also included in that, the detection rate would increase to 99.6%. The experiments were performed on real devices in which the application can run exactly the way the user experiences it. Further to this, some comparisons of detection performance and code coverage were also included in this work. Traditional ML classifier performances were also compared. This novel method outperformed the ML-based methods such as NB, SL, SVM, J48, Pruning Rule-Based Classification Tree (PART), RF, and DL. In addition to this work, seeking the possibilities to include intrusion detection mechanism in the DL-Droid would be a valuable addition.

The AdMat model proposed in [107] discussed a CNN on Matrix-based approach to detect Android malware. This model characterised apps and treated them as images. Then the adjacency matrix was constructed for apps, and it was simplified with the size of 219 × 219 to enhance the efficiency in data processing after transferring decompiled source code into call-graph of Graph Modelling Language (GML) format. Those matrices were the input images to the CNN, and the model was trained to identify and classify malware and benign apps. This model has an accuracy of 98.2%. Even though the model is highly accurate, there are limitations to this work, such as performing static analysis only, and the performance depends on the number of used features.

The model proposed in [108] discussed a DL-based method that uses CNN approach to analyse API sequence call, opcode, and permissions to detect Android malware in a zero-day scenario. The model achieved a weighted average detection rate of 91% and 81% on two datasets Drebin and AMD after the model was trained. The model can further improve if the dynamic analysis techniques are also considered.

With an accuracy of 95%, a multimodal analysis of malware apps using information fusion was presented in [100] which used hybrid analysis techniques. The study used CBR for training and validation purposes. SVM and DT were compared with the proposed model validation, but the classic ML algorithms were outperformed by the CBR-based method. If the work can represent the knowledge representation, some of the limitations can be addressed.

Tables 8 and 9 comparatively summarise the above research studies related to deep learning based malware detection methods, where Table 8 listed studies with model accuracy below 90% and Table 9 listed studies with model accuracy above 90%.

Table 8. Deep learning based Malware Detection Approaches (Model Accuracy is below 90% or overall accuracy is not available).

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML/DL Algorithms/ Models	Selected DL Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2017	[104]	Using n-Gram methods after getting the Opcode sequence from .smali after dissembling .apk	Code Analysis for Opcodes	Genome, IntelSecurity, MacAfee, Google Play	CNN, NLP	Deep CNN	87%	Automatically learn the feature indicative of malware without hand engineering	Assumption of all APKs are benign in Google Play dataset while all are malicious in malware dataset
2021	[108]	Using DL based method which uses Convolution Neural Network based approach to analyse features	Code Analysis for API calls, Opcode and Manifest Analysis for Permission	Drebin, AMD	CNN	CNN	91% and 81% on two datasets	Reduce over fitting and possible to train to detect new malware just by collecting more sample apps	Did not compared with other ML/DL methods

Table 9. Deep learning based malware detection approaches (model accuracy is above 90%).

Year	Study	Detection Approach	Feature Extraction Method	Used Datasets	ML/DL Algorithms/ Models	Selected DL Algorithms/ Models	Model Accuracy	Strengths	Limitations/Drawbacks
2018	[102]	Applying LSTM on semantic structure of bytecode with 2 layers of detection and validating with DeepRefiner	Code Analysis for Opcode/bytecode	Google Play, VirusShare, MassVet	RNN, LSTM	LSTM	97.4%	High efficiency with average of 0.22 s to the 1st layer and 2.42 s to the 2nd layer detection	Need to train the model regularly to update the training model on new malware
2020	[105]	Detecting Malware attributes by vectorised opcode extracted from the bytecode of the APKs with one-hot encoding before apply DL Techniques	Code Analysis for Opcode	Drebin, AMD, VirusShare	BiLSTM, RNN, LSTM, Neural Networks, Deep Convents, Diabolo Network model	BiLSTMs	99.9%	Very high accuracy, Able to achieve zero day malware family without overhead of previous training	Did not analyse complete byte code
2020	[106]	Using DynaLog to select and extract features from Log files and using DL-Droid to perform feature ranking and apply DL	Code instrumentation analysis for java classes, intents, and systems calls	Intel Security	NB, SL, SVM, J48, PART, RF, DL	DL	99.6%	Experiments were performed on real devices High accuracy	Could have implemented the intrusion detection part also to make it more comprehensive malware detection tool
2021	[101]	Selecting features gained by feature selection approaches. Applying ML/DL models to detect malware	Code instrumentation for java classes, permissions, and API calls at the runtime	Android Permissions Dataset, Computer and security dataset	farthest first clustering, Y-MLP, nonlinear ensemble decision tree forest, DL	DL with methods in MLDroid	98.8%	High accuracy and easy to retrain the model to identify new malware	Human interaction would be required in some cases. Can contain issues in the datasets
2021	[107]	Characterising apps and treating as images. Then constructing the adjacency matrix. Then applying CNN to identify malware with AdMat framework	Code Analysis for API calls, Information flow, and Opcode	Drebin AMD	CNN	CNN	98.2%	High Accuracy and efficiency	Performance is depending on number of used features

6. Machine Learning Methods to Detect Code Vulnerabilities

Hackers do not just create malware. They also try to find loopholes in existing applications and perform malicious activities. Therefore, it is necessary to find vulnerabilities in Android source code. A code vulnerability of a program can happen due to a mistake at the designing, development, or configuration time which can be misused to infringe on the security [38]. Detection of code vulnerability can be performed in two ways. The first method is reverse-engineering the APK files using a similar approach discussed in Section 3. The second method is identifying the security flaws at the time of designing and developing the application [109]. The study conducted in [110] has identified five main categories of security approaches. They were secure requirements modelling, extended Unified Modeling Language (UML) based secure modelling profiles, non-UML-based secure modelling notations, vulnerability identification, adaption and mitigation, and software security-focused process. Under these categories, 52 security approaches were identified. All these approaches are used to identify software vulnerabilities at the time of designing and developing the applications. Based on the findings of the surveys and interviews conducted in [111] related to intervention for long-term software security, the importance of having an automated code analysis tool to identify vulnerabilities of the written codes has been identified. The empirical analysis conducted in [112] identified the static software metrics' correlation and the most informative metrics which can be used to find code vulnerability related to Android source codes.

6.1. Static, Dynamic, and Hybrid Source Code Analysis

Similar to analysing APKs for malware detection, there are three ways of analysing source codes. They are static analysis, dynamic analysis, and hybrid analysis. In static analysis, without executing the source code, a program is analysed to identify properties by converting the source to a generalised abstraction such as Abstract Syntax Tree (AST) [113]. The number of reported false vulnerabilities depends on the accuracy of the generalisation mechanism. The runtime behaviour of the application is monitored while using specific input parameters in dynamic analysis. The behaviour depends on the selection of input parameters. However, there are possibilities of undetected vulnerabilities [114].

In hybrid analysis, it provides the characteristics of both static analysis and dynamic analysis, which can analyse the source code and run the application to identify vulnerabilities while employing detection techniques [115].

The study conducted in [116] performed an online experiment where Android developers were the participants. Vulnerable code samples containing hard-coded credentials, encryptions, Structured Query Language (SQL) injections, and logging with sensitive data were given to the participants together with the guidance of static analysis tools and asked to indicate the appropriate fix. After analysing the experiment results, it has been identified that automated code vulnerability detection support is required for the developers to perform better when developing secure applications.

To analyse Android source code, Android Linters can be applied. Linters have been proposed to detect and fix these bad practices and they perform a static analysis based on AST or Universal AST (UAST) generation through written source codes [117]. The study in [118] discussed several Linters such as PMD, CheckStyle, Infer, and FindBugs, Detekt, Ktlint, and Android Lint discussed the usage of them. Android studio adopts the Android Lint, which identifies 339 issues related to correctness, security, performance, usability, accessibility, and internationalisation. In the proposed model in FixDroid [27], security-oriented suggestions along with their fixes were provided to the developer once the Android Lint identified security flaws. The FixDroid method can further be improved by employing ML techniques to produce highly accurate security suggestions.

However, just warning the developer about security issues in the code is not sufficient. There should be a mechanism to inform the developer about the severity level of the security issue also. By using app user reviews, OASSIS [119] proposed a method to prioritise static analysis warnings generated from Android Lint. Based on the review analysis using sentiment analysis, it was possible to identify the issues in Android apps. After receiving prioritised lint warnings, developers will able to take prompt actions. The study in [120] proposed a mechanism named as MagpieBridge to integrate static analysis into Integrated Development Environments (IDEs) and code editors such as Eclipse, IntelliJ, Jupyter, Sublime Text, and PyCharm. However, the possibility of extending this to the Android platform should be discussed further.

In [121], using static and dynamic analysis, a vulnerability identification of Secure Sockets Layer (SSL)/Transport Layer Security (TLS) certificate verification in Android application was described. This experiment found that out of the analysed 2213 Android apps, 360 apps contain vulnerable codes using the proposed framework of DCDroid. Therefore, through SSL/TLS certificates, it is possible to identify some vulnerabilities.

6.2. Applying ML to Detect Source Code Vulnerabilities

It has been proven that ML methods can be applied on a generalised architecture such as AST to detect Android code vulnerabilities [38]. Most of the research was conducted using static analysis techniques to analyse the source code.

With the use of ML, vulnerability detection rules were extracted with static metrics as discussed in [122]. Thirty-two supervised ML algorithms were considered for most common vulnerabilities and identified that when the model used the J48 ML algorithm, 96% accuracy could be obtained in vulnerability detection. The model proposed in [123] discussed an automated mechanism to classify well-written and malicious code using a portable executable (PE) structure through static analysis and ML with an accuracy of 98.77%. The proposed methodology used RF, GB, DT, and CNN as ML models.

The study in [124] built a model to predict software vulnerabilities of codes using ML before releasing the code. After developing a source code representation using AST and intelligently analysing it, the ML models were applied. Popular datasets such as NIST SAMATE, Draper VDISC, and SATE IV Juliet Test Suite, which contain C, C++, Java, and Python source codes, were used to train the model. However, using this model, it was not possible to locate a specific place of vulnerability. It is identified as a drawback, and it has not proven that the same approach is possible to apply to other programming languages and frameworks. However, there is a possibility of using this approach for Android applications, which were developed using Java.

In [125], using C and C++ source codes, a vulnerability detection system was proposed using ML and deep feature representation learning. Apart from using the existing datasets, the Drapper dataset was compiled using Drebin and Github repositories with millions of open-source functions and labelled with carefully selected findings. The findings of the research were compared with Bag of Words (BOW), RF, RNN, and CNN models.

The study conducted in [126] developed a mechanism to classify subroutines as vulnerable or not vulnerable in C language using ML methods. The National Vulnerability Dataset (NVD) was used to collect C programming code blocks and their known vulnerabilities. After preparing the AST and preprocessing the data, feature extraction, feature selection, and classification tasks were performed and ML algorithms were applied.

The applicability of deep learning to detect code vulnerabilities was discussed in [127]. Comparison of using three DL algorithms CNN, LSTM, and CNN-LSTM were discussed in this study. The proposed model has an accuracy of 83.6% when applying the DL models. Using Deep Neural Networks, it was possible to predict vulnerable code components. The model in [128] evaluated it using some Java-based Android applications. In this mechanism, N-gram analysis and statistical feature selection for constructing features were performed. This model can classify vulnerable classes with high precision, accuracy, and recall.

In [129], a model was proposed to detect zero-day Android malware using a distinctive parallel classifier and a mechanism to identify oncoming highly elusive vulnerabilities in the source code with an accuracy of 98.27% with the use of Ml algorithms; PART, Ripple Down Rule Learner (RIDOR), SVM, and MLP.

ML-Based Vulnerability Detection Specifically for Android

There is less research conducted relating to Android vulnerability detection with ML. The methodology of the studies, which were conducted on general programming languages, could apply to the Android code vulnerability detection after training the model using specific code datasets and adjusting the generalisation mechanism.

The work conducted in [130] prepared a manually curated dataset that can be used to fix vulnerabilities of open-source software. The possibility of automatically identifying security-related commits in the relevant code repository has been proven since it has been successfully used to train classifiers.

In [131] repository of Android security vulnerabilities was created named AndroVul, which includes dangerous permissions, security code smells, and high-risk shell command vulnerabilities. In [132], a study was conducted to predicatively analyse the vulnerabilities in Internet of Things (IoT) related Android applications using statistical codes and applying ML. In this study, 1406 Android apps were taken with various risk levels, and six ML models (KNN, LR, RF, DT, SVM, and GB) were administered to examine security risk prediction. It is identified that RF performs well in the intermediate risk level. GB performs well at a very high-risk level compared to the other ML model-based approaches. The study conducted in [133] proposed an ML-based vulnerabilities detection mechanism to identify security flaws of Android Intents using hybrid analysis. Adaboost algorithm was used to perform the ML based analysis.

Tables 10 and 11 summarise selected studies from above which are related to Android vulnerability analysis. Table 10 lists the studies which have model accuracy below 90% and Table 11 lists the studies which have model accuracy above 90%.

Table 10. Android vulnerability detection mechanisms (Model accuracy is below 90%).

Year	Study	Code Analysis Method	Approach	Used ML/DL Methods/ Frameworks	Accuracy of the Model
2017	[127]	Dynamic Analysis	Collected 9872 sequences of function calls as features. Performed dynamic analysis with DL methods	CNN-LSTM	83.6%
2017	[133]	Hybrid Analysis	Decompiled the apk file. Performed static analysis of the manifest file to obtain the components/permissions. Dynamic analysis and fuzzy testing were conducted and obtained system status.	AB and DT	77%
2019	[115]	Hybrid Analysis	Reverse engineered the APK, Decoded the manifest files & codes and extracted meta data from it. Performed dynamic analysis to identify intent crashing and insecure network connections for API calls. Generated the report.	AndroShield	84%
2020	[124]	Hybrid Analysis	Performed intelligent analysis of generated AST. Checked ML can differentiate vulnerable and nonvulnerable.	MLP and a customised model	70.1%

Table 11. Android vulnerability detection mechanisms (model accuracy is above 90%).

Year	Study	Code Analysis Method	Approach	Used ML/DL Methods/ Frameworks	Accuracy of the Model
2017	[113]	Static Analysis	Generated the AST, navigated it, and computed detection rules. Identified smells when training with manually created dataset.	ADOCTOR framework	98%
2017	[128]	Static Analysis	Combined N-gram analysis and statistical feature selection for constructing features. Evaluated the performance of the proposed technique based on a number of Java Android programs.	Deep Neural Network	92.87%
2019	[129]	Hybrid Analysis	Decompiled the APK and selected the features and executed the APK and generated log files with system calls. Generated the vector space and trained with ML algorithms as parallel classifiers.	MLP, SVM, PART, RIDOR, MaxProb, ProdProb	98.37%
2020	[121]	Hybrid Analysis	In static analysis, vulnerabilities of SSL/TLS certification were identified. Results from static analysis about user interfaces were analysed to confirm SSL/TLS misuse in dynamic analysis.	DCDroid	99.39%
2021	[122]	Static Analysis	32 supervised ML algorithms were considered for 3 common vulnerabilities: Lawofdemeter, BeanMemberShouldSerialize, and LocalVariablecouldBeFinal	J48	96%
2021	[123]	Static Analysis	Classified malicious code using a PE structure and a method for classifying it using a PE structure	CNN	98.77%

7. Results and Discussion

Based on the reviewed studies in ML/DL based methods to detect malware, it is identified that 65% of studies related to malware detection techniques used static analysis, 15% used dynamic analysis, and the remaining 20% followed the hybrid analysis technique. This is illustrated in Figure 3. This high attractiveness of static analysis may be due to the various advantages associated with it over dynamic analysis, such as ability to detect more vulnerabilities, localising vulnerabilities, and offering cost benefits.

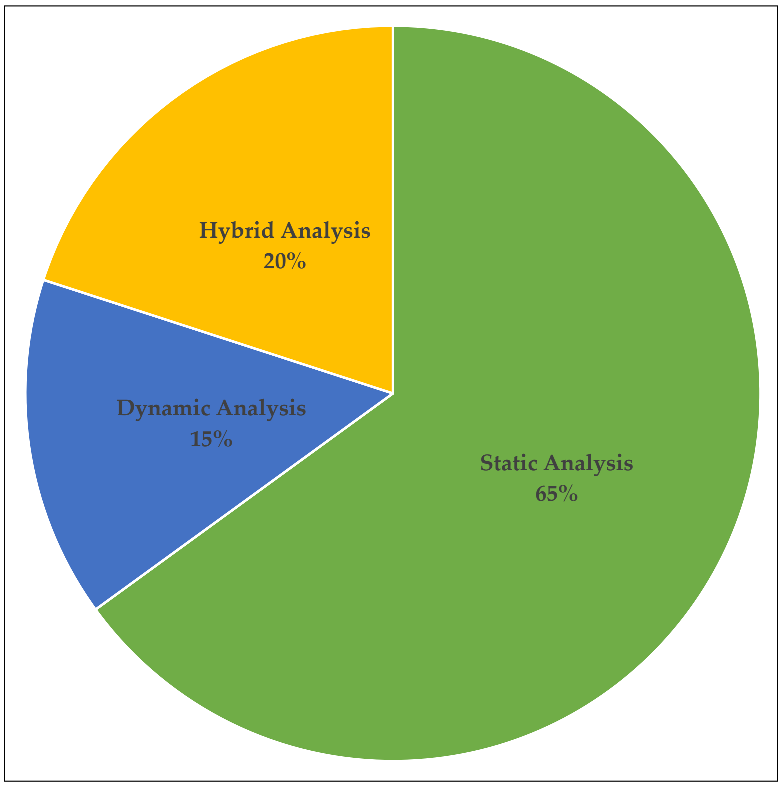

Figure 3. Malware analysis techniques used in the reviewed studies.

Many ML/DL based malware detection studies used the code analysis method as the feature extraction method. Apart from that, manifest analysis and system call analysis methods are the other widely used methods. Figure 4 illustrates those feature extraction methods used in the reviewed studies. It is possible to detect a substantial amount of malware after analysing decompiled source codes rather than analysing permissions or other features. That may be the reason for the high usage of code analysis in malware detection.

By using the feature extraction methods, permissions, API calls, system calls, and opcodes are the most widely extracted features. This is illustrated in Figure 5 along with the other extracted features in the reviewed studies. Many hybrid analysis methods extracted permissions as the feature to perform static analysis. It is easy to analyse permissions when comparing with the other features too. These could be reasons for the high usage of permissions as the extracted feature. Services and network protocols have low usage in feature extractions. The reason for this may be it is comparatively not easy to analyse those features.

The datasets used in ML/DL based Android malware detection studies to train the algorithms are illustrated in Figure 6. Drebin was the most widely used dataset in Android Malware Detection, and it was used in 18 reviewed studies. Google Play, MalGenome, and AMD datasets are the other widely used datasets. The reason for the highest usage of the Drebin dataset may be because it provides a comprehensive labelled dataset. Since Google Play is the official app store of Android, it may be a reason to have high usage for the dataset from Google.

It is identified that the RF, SVM, and NB are at the top of widely studied ML models to detect Android malware. The reason may be that the resource cost to run RF, SVM, or NB based models is low. Models like CNN, LSTM, and AB have less usage because to run such advanced models, good computing power is required, and the trend for DL-based models was also boosted in recent years. Table 12 summarises widely used ML/DL algorithms with their advantages and disadvantages. Figure 7 illustrates all of the studied ML/DL models with their usage in the reviewed studies.

The majority of the studies used hybrid analysis and static analysis as the source code analysis techniques in vulnerability detection in Android, as illustrated in Figure 8. To perform a highly accurate vulnerability analysis, the source code should be analysed and executed too. Therefore, this may be the reason to have hybrid analysis and static analysis as the widely used source code analysis methods to detect vulnerabilities.

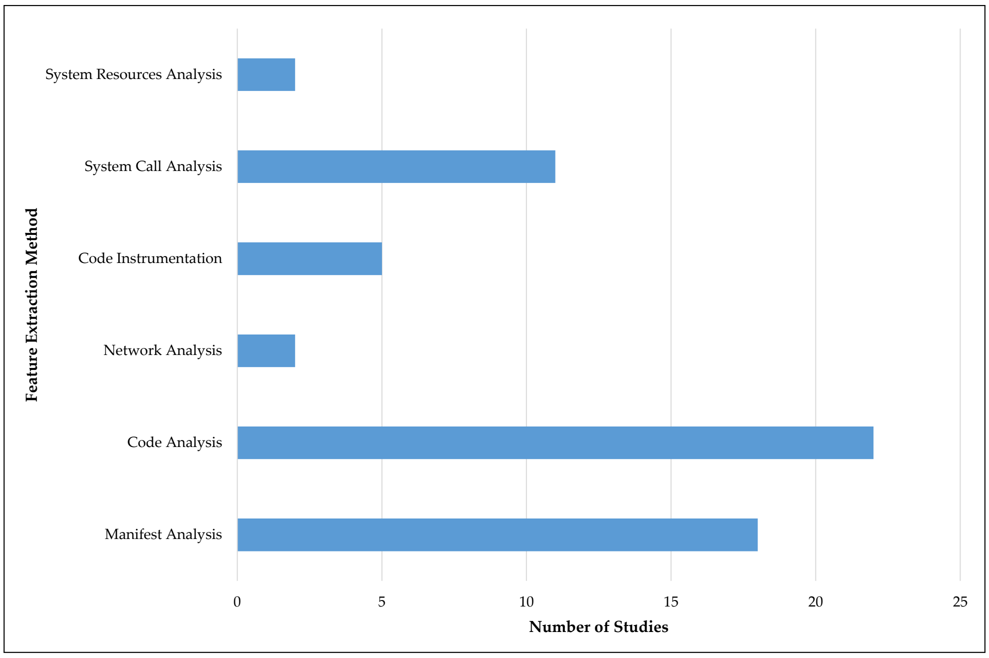

Figure 4. Feature extraction methods used in the reviewed studies.

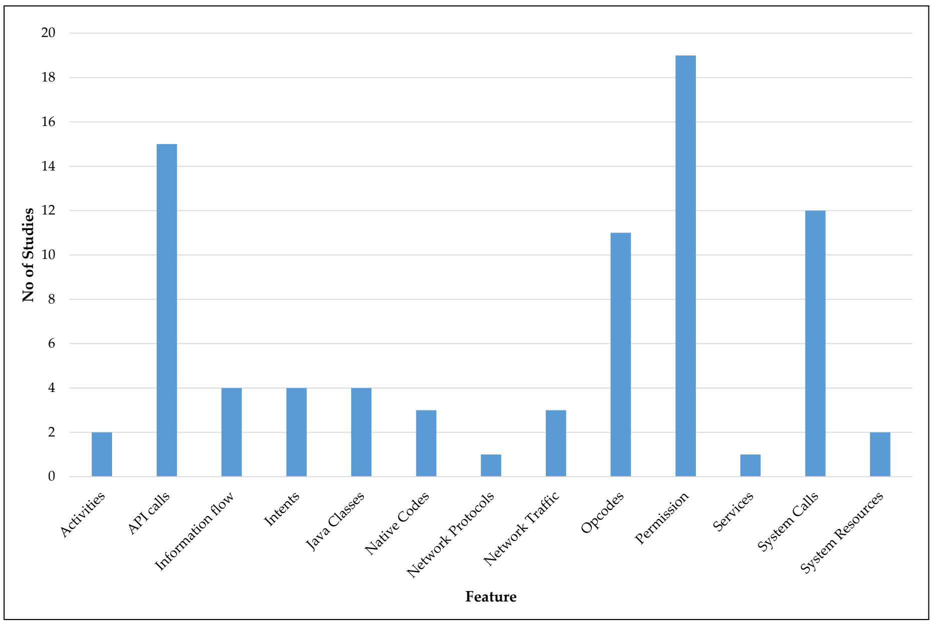

Figure 5. Extracted features in the reviewed studies.

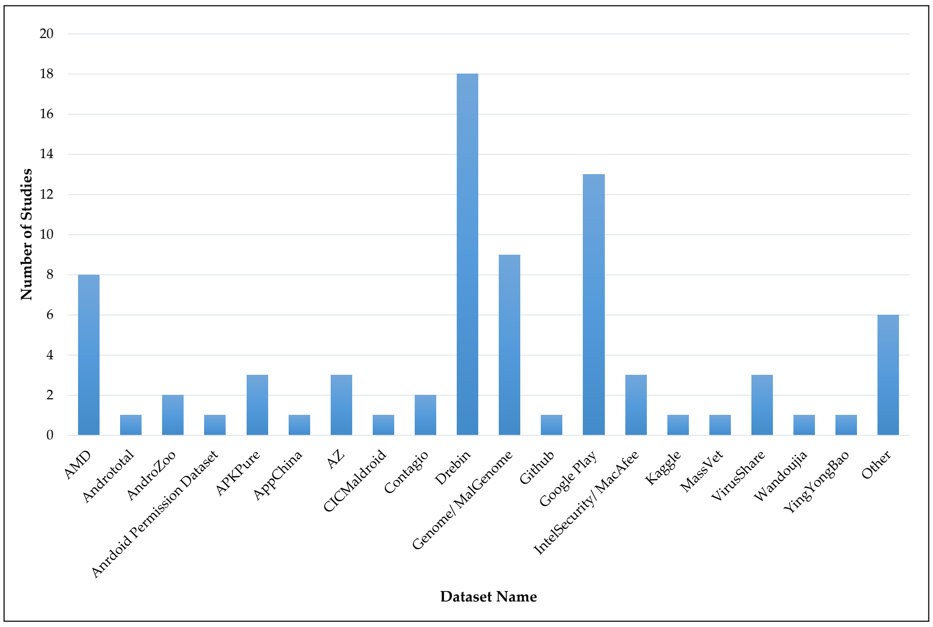

Figure 6. Usage of datasets.

Table 12. Commonly used ML/DL algorithms for Android malware detection.

Algorithm	**Advantages**	**Disadvantages**
DT	• Possible handle samples with missing values • Easy to understand	• Might cause the overfitting problem
NB	• Easily and quickly trainable	• Need to calculate prior probability • Not applicable if the feature variables are corelated
Regression Models	• Widely used in statistics based studies • Direct and Fast	• Not possible to deal well with high dimensional features
KNN	• Suitable to solve multiclassification problems	• Computation overhead is relatively high • Issues with the skewness of data
SVM	• Possible to solve high dimensional nonlinear small scale problems	• High overhead in data processing • Might face some issues when there are missing values in the sample
K-Means	• Easy to implement • Fast and simple	• Sensitive to outliers
RF	• Reduces overfitting • Normalising of data is not required	• Requires much time to train • Requires high computational power
Neural Networks	• Highly accurate • Strong fault tolerance	• Requires much time to train • Require a large number of data to train the model
LSTM	• Capable to remember facts for lengthy interval	• Requires high computational resources
CNN	• Reduce unimportant parameters by weight sharing and downsampling	• High computational cost
Ensemble Learning	• Accuracy is high	• Overhead on model training and maintenance

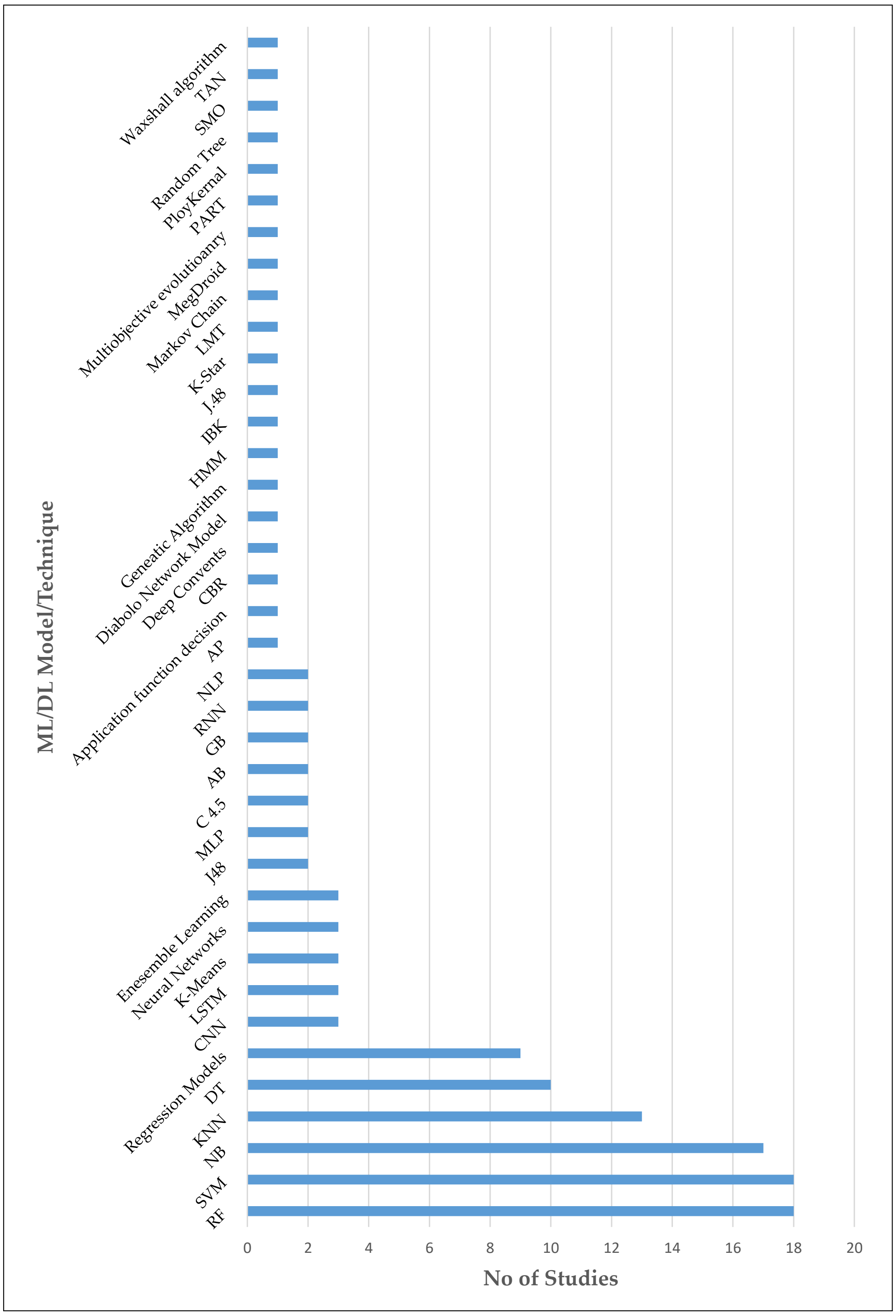

Figure 7. ML/DL models used in the reviewed studies.

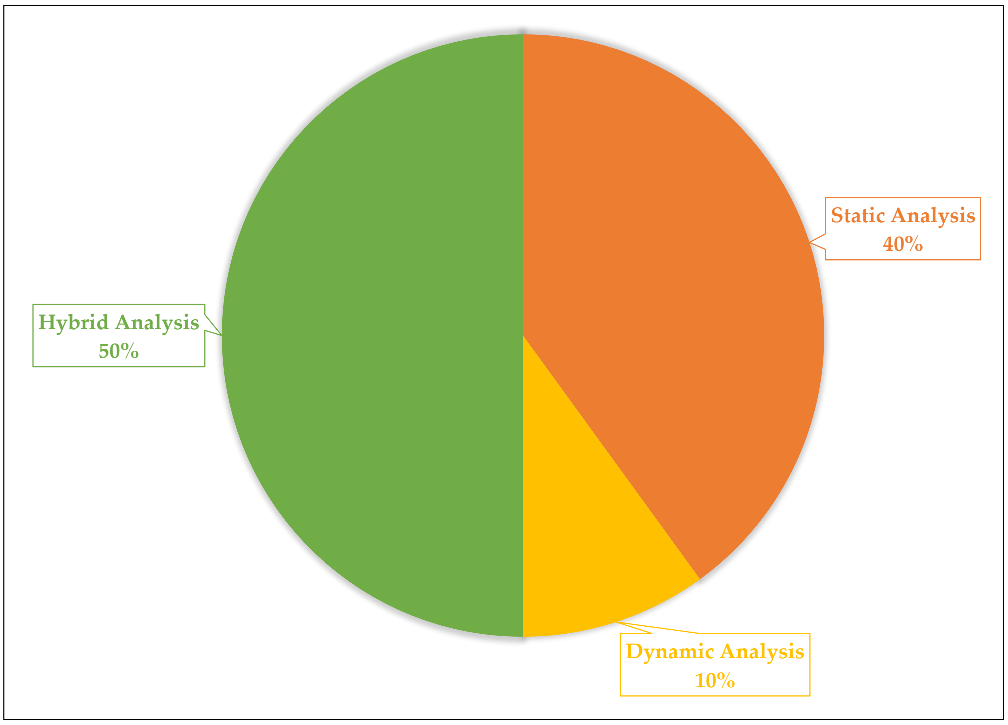

Figure 8. Android source code vulnerability analysis methods.

8. Conclusions and Future Work

Any smartphone is potentially vulnerable to security breaches, but Android devices are more lucrative for attackers. This is due to its open-source nature and the larger market share compared to other operating systems for mobile devices. This paper discussed the Android architecture and its security model, as well as potential threat vectors for the Android operating system. Based upon the available literature, a systematic review of the state-of-the-art ML-based Android malware detection techniques was carried out, covering the latest research from 2016 to 2021. It discussed the available ML and DL models and their performance in Android malware detection, code and APK analysis methods, feature analysis and extraction methods, and strengths and limitations of the proposed methods. Malware aside, if a developer makes a mistake, it is easier for a hacker to find and exploit these vulnerabilities. Therefore, methods for the detection of source code vulnerabilities using ML were discussed. The work identified the potential gaps in previous research and possible future research directions to enhance the security of Android OS.

Both Android malware and its detection techniques are evolving. Therefore, we believe that similar future reviews are necessary to cover these emerging threats and their detection methods. As per our findings in this paper, since DL methods have proven to be more accurate than traditional ML models, it will be beneficial to the research community if more comprehensive systematic reviews can be performed by focusing only on DL-based malware detection on Android. The possibility of using reinforcement learning to identify source code vulnerabilities is another area of interest in which systematic reviews and studies can be carried out.

Author Contributions: Conceptualization, J.S., H.K. and M.O.A.-K.; methodology, J.S., H.K. and M.O.A.-K.; validation, J.S., H.K. and M.O.A.-K.; investigation, J.S.; Project administration, H.K.; writing—original draft preparation, J.S.; writing—review and editing, J.S., H.K. and M.O.A.-K.; visualization, J.S.; supervision, H.K. and M.O.A.-K.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: We thank the Accelerating Higher Education Expansion and Development (AHEAD) grant of Sri Lanka, University of Kelaniya—Sri Lanka and Robert Gordon University—United Kingdom for their support.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Number of Mobile Phone Users Worldwide from 2016 to 2023 (In Billions). Available online: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (accessed on 19 May 2021).
2. Mobile Operating System Market Share Worldwide. Available online: https://gs.statcounter.com/os-market-share/mobile/worldwide/ (accessed on 19 May 2021).
3. Number of Android Applications on the Google Play Store. Available online: https://www.appbrain.com/stats/number-of-android-apps/ (accessed on 19 May 2021).
4. Gibert, D.; Mateu, C.; Planes, J. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *J. Netw. Comput. Appl.* **2020**, *153*, 102526. [CrossRef]
5. Khan, J.; Shahzad, S. Android Architecture and Related Security Risks. *Asian J. Technol. Manag. Res. [ISSN: 2249–0892]* **2015**, *5*, 14–18. Available online: http://www.ajtmr.com/papers/Vol5Issue2/Vol5Iss2_P4.pdf (accessed on 19 May 2021).
6. Platform Architecture. Available online: https://developer.android.com/guide/platform (accessed on 19 May 2021).
7. Android Runtime (ART) and Dalvik. Available online: https://source.android.com/devices/tech/dalvik (accessed on 19 May 2021).
8. Cai, H.; Ryder, B.G. Understanding Android application programming and security: A dynamic study. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 364–375. [CrossRef]
9. Liu, K.; Xu, S.; Xu, G.; Zhang, M.; Sun, D.; Liu, H. A Review of Android Malware Detection Approaches Based on Machine Learning. *IEEE Access* **2020**, *8*, 124579–124607. [CrossRef]
10. Gilski, P.; Stefanski, J. Android os: A review. *Tem J.* **2015**, *4*, 116. Available online: https://www.temjournal.com/content/41/14/temjournal4114.pdf (accessed on 19 May 2021).
11. Privacy in Android 11 | Android Developers. Available online: https://developer.android.com/about/versions/11/privacy (accessed on 19 May 2021).
12. Garg, S.; Baliyan, N. Comparative analysis of Android and iOS from security viewpoint. *Comput. Sci. Rev.* **2021**, *40*, 100372. [CrossRef]
13. Odusami, M.; Abayomi-Alli, O.; Misra, S.; Shobayo, O.; Damasevicius, R.; Maskeliunas, R. Android malware detection: A survey. In *International Conference on Applied Informatics*; Springer: Cham, Switzerland, 2018; pp. 255–266. [CrossRef]
14. Bhat, P.; Dutta, K. A survey on various threats and current state of security in android platform. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–35. [CrossRef]
15. Tam, K.; Feizollah, A.; Anuar, N.B.; Salleh, R.; Cavallaro, L. The evolution of android malware and android analysis techniques. *ACM Comput. Surv. (CSUR)* **2017**, *49*, 1–41. [CrossRef]
16. Li, L.; Li, D.; Bissyandé, T.F.; Klein, J.; Le Traon, Y.; Lo, D.; Cavallaro, L. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Trans. Inf. Forensics Secur.* **2017**, *12*, 1269–1284. [CrossRef]
17. Ashawa, M.A.; Morris, S. Analysis of Android malware detection techniques: A systematic review. *Int. J. Cyber-Secur. Digit. Forensics* **2019**, *8*, 177–187. [CrossRef]
18. Suarez-Tangil, G.; Tapiador, J.E.; Peris-Lopez, P.; Ribagorda, A. Evolution, detection and analysis of malware for smart devices. *IEEE Commun. Surv. Tutor.* **2013**, *16*, 961–987. [CrossRef]
19. Mos, A.; Chowdhury, M.M. Mobile Security: A Look into Android. In Proceedings of the 2020 IEEE International Conference on Electro Information Technology (EIT), Chicago, IL, USA, 31 July–1 August 2020; pp. 638–642. [CrossRef]
20. Faruki, P.; Bharmal, A.; Laxmi, V.; Ganmoor, V.; Gaur, M.S.; Conti, M.; Rajarajan, M. Android security: A survey of issues, malware penetration, and defenses. *IEEE Commun. Surv. Tutor.* **2014**, *17*, 998–1022. [CrossRef]
21. Android Security & Privacy 2018 Year in Review. Available online: https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf (accessed on 19 May 2021).
22. Kalutarage, H.K.; Nguyen, H.N.; Shaikh, S.A. Towards a threat assessment framework for apps collusion. *Telecommun. Syst.* **2017**, *66*, 417–430. [CrossRef]
23. Asavoae, I.M.; Blasco, J.; Chen, T.M.; Kalutarage, H.K.; Muttik, I.; Nguyen, H.N.; Roggenbach, M.; Shaikh, S.A. Towards automated android app collusion detection. *arXiv* **2016**, arXiv:1603.02308.
24. Asăvoae, I.M.; Blasco, J.; Chen, T.M.; Kalutarage, H.K.; Muttik, I.; Nguyen, H.N.; Roggenbach, M.; Shaikh, S.A. Detecting malicious collusion between mobile software applications: The Android case. In *Data Analytics and Decision Support for Cybersecurity*; Springer: Cham, Switzerland, 2017; pp. 55–97. [CrossRef]
25. Malik, J. Making sense of human threats and errors. *Comput. Fraud Secur.* **2020**, *2020*, 6–10. [CrossRef]
26. Calciati, P.; Kuznetsov, K.; Gorla, A.; Zeller, A. Automatically Granted Permissions in Android apps: An Empirical Study on their Prevalence and on the Potential Threats for Privacy. In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Korea, 29–30 June 2020; pp. 114–124. [CrossRef]

27. Nguyen, D.C.; Wermke, D.; Acar, Y.; Backes, M.; Weir, C.; Fahl, S. A stitch in time: Supporting android developers in writing secure code. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 1065–1077. [CrossRef]
28. Garg, S.; Baliyan, N. Android Security Assessment: A Review, Taxonomy and Research Gap Study. *Comput. Secur.* **2020**, *100*, 102087. [CrossRef]
29. Van Engelen, J.E.; Hoos, H.H. A survey on semi-supervised learning. *Mach. Learn.* **2020**, *109*, 373–440. [CrossRef]
30. Alauthman, M.; Aslam, N.; Al-Kasassbeh, M.; Khan, S.; Al-Qerem, A.; Choo, K.K.R. An efficient reinforcement learning-based Botnet detection approach. *J. Netw. Comput. Appl.* **2020**, *150*, 102479. [CrossRef]
31. Shrestha, A.; Mahmood, A. Review of deep learning algorithms and architectures. *IEEE Access* **2019**, *7*, 53040–53065. [CrossRef]
32. Page, M.; McKenzie, J.; Bossuyt, P.; Boutron, I.; Hoffmann, T.; Mulrow, C.; Shamseer, L.; Tetzlaff, J.M.; Akl, E.A.; Brennan, S.E.; et al. The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *BMJ* **2020**, *372*. [CrossRef]
33. Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, London, UK, 13–14 May 2014; pp. 1–10. [CrossRef]
34. Li, L.; Bissyandé, T.F.; Papadakis, M.; Rasthofer, S.; Bartel, A.; Octeau, D.; Klein, J.; Traon, L. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* **2017**, *88*, 67–95. [CrossRef]
35. Pan, Y.; Ge, X.; Fang, C.; Fan, Y. A Systematic Literature Review of Android Malware Detection Using Static Analysis. *IEEE Access* **2020**, *8*, 116363–116379. [CrossRef]
36. Sharma, T.; Rattan, D. Malicious application detection in android—A systematic literature review. *Comput. Sci. Rev.* **2021**, *40*, 100373. [CrossRef]
37. Liu, Y.; Tantithamthavorn, C.; Li, L.; Liu, Y. Deep Learning for Android Malware Defenses: A Systematic Literature Review. *arXiv* **2021**, arXiv:2103.05292.
38. Ghaffarian, S.M.; Shahriari, H.R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–36. [CrossRef]
39. Chen, T.; Mao, Q.; Yang, Y.; Lv, M.; Zhu, J. TinyDroid: A lightweight and efficient model for Android malware detection and classification. *Mob. Inf. Syst.* **2018**, *2018*. [CrossRef]
40. Nisa, M.; Shah, J.H.; Kanwal, S.; Raza, M.; Khan, M.A.; Damaševičius, R.; Blažauskas, T. Hybrid malware classification method using segmentation-based fractal texture analysis and deep convolution neural network features. *Appl. Sci.* **2020**, *10*, 4966. [CrossRef]
41. Amin, M.; Shah, B.; Sharif, A.; Ali, T.; Kim, K.l.; Anwar, S. Android malware detection through generative adversarial networks. *Trans. Emerg. Telecommun. Technol.* **2019**, e3675. [CrossRef]
42. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of the 2014 Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2014; doi:10.14722/ndss.2014.23247 [CrossRef]
43. Google Play. Available online: https://play.google.com/ (accessed on 19 May 2021).
44. AndroZoo. Available online: https://androzoo.uni.lu/ (accessed on 19 May 2021).
45. AppChina. Available online: https://tracxn.com/d/companies/appchina.com (accessed on 19 May 2021).
46. Tencent. Available online: https://www.pcmgr-global.com/ (accessed on 19 May 2021).
47. YingYongBao. Available online: https://android.myapp.com/ (accessed on 19 May 2021).
48. Contagio. Available online: https://www.impactcybertrust.org/dataset_view?idDataset=1273/ (accessed on 19 May 2021).
49. Zhou, Y.; Jiang, X. Dissecting android malware: Characterization and evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 95–109. [CrossRef]
50. VirusShare. Available online: https://virusshare.com/ (accessed on 19 May 2021).
51. Intel Security/MacAfee. Available online: https://steppa.ca/portfolio-view/malware-threat-intel-datasets/ (accessed on 19 May 2021).
52. Chen, K.; Wang, P.; Lee, Y.; Wang, X.; Zhang, N.; Huang, H.; Zou, W.; Liu, P. Finding unknown malice in 10 s: Mass vetting for new threats at the google-play scale. In Proceedings of the 24th USENIXSecurity Symposium (USENIX Security 15), Redmond, WA, USA, 7–8 May 2015; pp. 659–674.
53. Android Malware Dataset. Available online: http://amd.arguslab.org/ (accessed on 19 May 2021).
54. APKPure. Available online: https://m.apkpure.com/ (accessed on 19 May 2021).
55. Anrdoid Permission Dataset. Available online: https://data.mendeley.com/datasets/b4mxg7ydb7/3 (accessed on 19 May 2021).
56. Maggi, F.; Valdi, A.; Zanero, S. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, Berlin, Germany, 8 November 2013; pp. 49–54. [CrossRef]
57. Wandoujia App Market. Available online: https://www.wandoujia.com/apps (accessed on 19 May 2021).
58. Google Playstore Appsin Kaggle. Available online: https://www.kaggle.com/gauthamp10/google-playstore-apps (accessed on 19 May 2021).
59. CICMaldroid Dataset. Available online: https://www.unb.ca/cic/datasets/maldroid-2020.html (accessed on 19 May 2021).
60. AZ Dataset. Available online: https://www.azsecure-data.org/other-data.html/ (accessed on 19 May 2021).

61. Github Malware Dataset. Available online: https://github.com/topics/malware-dataset (accessed on 19 May 2021).
62. Alqahtani, E.J.; Zagrouba, R.; Almuhaideb, A. A Survey on Android Malware Detection Techniques Using Machine Learning Algorithms. In Proceedings of the 2019 Sixth International Conference on Software Defined Systems (SDS), Rome, Italy, 10–13 June 2019; pp. 110–117. [CrossRef]
63. Lopes, J.; Serrão, C.; Nunes, L.; Almeida, A.; Oliveira, J. Overview of machine learning methods for Android malware identification. In Proceedings of the 2019 7th International Symposium on Digital Forensics and Security (ISDFS), Barcelos, Portugal, 10–12 June 2019; pp. 1–6. [CrossRef]
64. Choudhary, M.; Kishore, B. HAAMD: Hybrid analysis for Android malware detection. In Proceedings of the 2018 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 4–6 January 2018; pp. 1–4. [CrossRef]
65. Kouliaridis, V.; Kambourakis, G. A Comprehensive Survey on Machine Learning Techniques for Android Malware Detection. *Information* **2021**, *12*, 185. [CrossRef]
66. Chen, L.; Hou, S.; Ye, Y.; Chen, L. An adversarial machine learning model against android malware evasion attacks. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*; Springer: Cham, Switzerland, 2017; pp. 43–55. [CrossRef]
67. Lubuva, H.; Huang, Q.; Msonde, G.C. A review of static malware detection for Android apps permission based on deep learning. *Int. J. Comput. Netw. Appl.* **2019**, *6*, 80–91. [CrossRef]
68. Li, J.; Sun, L.; Yan, Q.; Li, Z.; Srisa-An, W.; Ye, H. Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3216–3225. [CrossRef]
69. Mcdonald, J.; Herron, N.; Glisson, W.; Benton, R. Machine Learning-Based Android Malware Detection Using Manifest Permissions. In Proceedings of the 54th Hawaii International Conference on System Sciences, Maui, HI, USA, 5–8 January 2021; p. 6976. [CrossRef]
70. Şahin, D.Ö.; Kural, O.E.; Akleylek, S.; Kılıç, E. A novel permission-based Android malware detection system using feature selection based on linear regression. *Neural Comput. Appl.* **2021**, 1–16. [CrossRef]
71. Nawaz, A. Feature Engineering based on Hybrid Features for Malware Detection over Android Framework. *Turk. J. Comput. Math. Educ. (TURCOMAT)* **2021**, *12*, 2856–2864.
72. Cai, L.; Li, Y.; Xiong, Z. JOWMDroid: Android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. *Comput. Secur.* **2021**, *100*, 102086. [CrossRef]
73. Zhang, P.; Cheng, S.; Lou, S.; Jiang, F. A novel Android malware detection approach using operand sequences. In Proceedings of the 2018 Third International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC), Shanghai, China, 18–19 October 2018; pp. 1–5. [CrossRef]
74. Wei, L.; Luo, W.; Weng, J.; Zhong, Y.; Zhang, X.; Yan, Z. Machine learning-based malicious application detection of android. *IEEE Access* **2017**, *5*, 25591–25601. [CrossRef]
75. Onwuzurike, L.; Mariconti, E.; Andriotis, P.; Cristofaro, E.D.; Ross, G.; Stringhini, G. MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur. (TOPS)* **2019**, *22*, 1–34. [CrossRef]
76. Zhang, H.; Luo, S.; Zhang, Y.; Pan, L. An efficient Android malware detection system based on method-level behavioral semantic analysis. *IEEE Access* **2019**, *7*, 69246–69256. [CrossRef]
77. Meng, G.; Xue, Y.; Xu, Z.; Liu, Y.; Zhang, J.; Narayanan, A. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 306–317. [CrossRef]
78. Wang, Z.; Li, C.; Yuan, Z.; Guan, Y.; Xue, Y. DroidChain: A novel Android malware detection method based on behavior chains. *Pervasive Mob. Comput.* **2016**, *32*, 3–14. [CrossRef]
79. Androguard. Available online: https://pypi.org/project/androguard/ (accessed on 19 May 2021).
80. Damodaran, A.; Di Troia, F.; Visaggio, C.A.; Austin, T.H.; Stamp, M. A comparison of static, dynamic, and hybrid analysis for malware detection. *J. Comput. Virol. Hacking Tech.* **2017**, *13*, 1–12. [CrossRef]
81. Sun, Y.; Xie, Y.; Qiu, Z.; Pan, Y.; Weng, J.; Guo, S. Detecting Android malware based on extreme learning machine. In Proceedings of the 2017 IEEE 15th International Conference on Dependable, Autonomic and Secure Computing, 15th International Conference on Pervasive Intelligence and Computing, 3rd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), Orlando, FL, USA, 6–10 November 2017; pp. 47–53. [CrossRef]
82. Tian, K.; Yao, D.; Ryder, B.G.; Tan, G.; Peng, G. Detection of repackaged android malware with code-heterogeneity features. *IEEE Trans. Dependable Secur. Comput.* **2017**, *17*, 64–77. [CrossRef]
83. Kabakus, A.T. What static analysis can utmost offer for Android malware detection. *Inf. Technol. Control* **2019**, *48*, 235–249. [CrossRef]
84. Koli, J. RanDroid: Android malware detection using random machine learning classifiers. In Proceedings of the 2018 Technologies for Smart-City Energy Security and Power (ICSESP), Bhubaneswar, India, 28–30 March 2018; pp. 1–6. [CrossRef]
85. Lou, S.; Cheng, S.; Huang, J.; Jiang, F. TFDroid: Android malware detection by topics and sensitive data flows using machine learning techniques. In Proceedings of the 2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT), Kahului, HI, USA, 14–17 March 2019; pp. 30–36. [CrossRef]

86. Wang, W.; Gao, Z.; Zhao, M.; Li, Y.; Liu, J.; Zhang, X. DroidEnsemble: Detecting Android malicious applications with ensemble of string and structural static features. *IEEE Access* **2018**, *6*, 31798–31807. [CrossRef]
87. Garg, S.; Peddoju, S.K.; Sarje, A.K. Network-based detection of Android malicious apps. *Int. J. Inf. Secur.* **2017**, *16*, 385–400. [CrossRef]
88. Sikder, A.K.; Aksu, H.; Uluagac, A.S. 6thsense: A context-aware sensor-based attack detector for smart devices. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 397–414. [CrossRef]
89. Mahindru, A.; Singh, P. Dynamic permissions based android malware detection using machine learning techniques. In Proceedings of the 10th Innovations in Software Engineering Conference, Jaipur, India, 5–7 February 2017; pp. 202–210. [CrossRef]
90. Salehi, M.; Amini, M.; Crispo, B. Detecting malicious applications using system services request behavior. In Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, TX, USA, 12–14 November 2019; pp. 200–209. [CrossRef]
91. Thangavelooa, R.; Jinga, W.W.; Lenga, C.K.; Abdullaha, J. DATDroid: Dynamic Analysis Technique in Android Malware Detection. *Int. J. Adv. Sci. Eng. Inf. Technol.* **2020**, *10*, 536–541. [CrossRef]
92. Hasan, H.; Ladani, B.T.; Zamani, B. MEGDroid: A model-driven event generation framework for dynamic android malware analysis. *Inf. Softw. Technol.* **2021**, *135*, 106569. [CrossRef]
93. Raphael, R.; Mathiyalagan, P. An Exploration of Changes Addressed in the Android Malware Detection Walkways. In Proceedings of the International Conference on Computational Intelligence, Cyber Security, and Computational Models, Coimbatore, India, 19–21 December 2019; Springer: Singapore, 2019; pp. 61–84. [CrossRef]
94. Jannat, U.S.; Hasnayeen, S.M.; Shuhan, M.K.B.; Ferdous, M.S. Analysis and detection of malware in Android applications using machine learning. In Proceedings of the 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE), Cox'sBazar, Bangladesh, 7–9 February 2019; pp. 1–7. [CrossRef]
95. Kapratwar, A.; Di Troia, F.; Stamp, M. *Static and Dynamic Analysis of Android Malware*; ICISSP: Porto, Portugal, 2017; pp. 653–662. [CrossRef]
96. Leeds, M.; Keffeler, M.; Atkison, T. A comparison of features for android malware detection. In Proceedings of the SouthEast Conference, Kennesaw, GA, USA, 13–15 April 2017; pp. 63–68. [CrossRef]
97. Hadiprakoso, R.B.; Kabetta, H.; Buana, I.K.S. Hybrid-Based Malware Analysis for Effective and Efficiency Android Malware Detection. In Proceedings of the 2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), Jakarta, Indonesia, 19–20 November 2020; pp. 8–12. [CrossRef]
98. Surendran, R.; Thomas, T.; Emmanuel, S. A TAN based hybrid model for android malware detection. *J. Inf. Secur. Appl.* **2020**, *54*, 102483. [CrossRef]
99. Martín, A.; Menéndez, H.D.; Camacho, D. MOCDroid: Multi-objective evolutionary classifier for Android malware detection. *Soft Comput.* **2017**, *21*, 7405–7415. 10.1007/s00500-016-2283-y. [CrossRef]
100. Qaisar, Z.H.; Li, R. Multimodal information fusion for android malware detection using lazy learning. *Multimed. Tools Appl.* **2021**, 1–15. [CrossRef]
101. Mahindru, A.; Sangal, A. MLDroid—Framework for Android malware detection using machine learning techniques. *Neural Comput. Appl.* **2021**, *33*, 5183–5240. [CrossRef]
102. Xu, K.; Li, Y.; Deng, R.H.; Chen, K. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, UK, 24–26 April 2018; pp. 473–487. [CrossRef]
103. JADX. Available online: https://github.com/skylot/jadx/ (accessed on 19 May 2021).
104. McLaughlin, N.; Martinez del Rincon, J.; Kang, B.; Yerima, S.; Miller, P.; Sezer, S.; Safaei, Y.; Trickel, E.; Zhao, Z.; Doupé, A.; et al. Deep android malware detection. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AZ, USA, 22–24 March 2017; pp. 301–308. [CrossRef]
105. Amin, M.; Tanveer, T.A.; Tehseen, M.; Khan, M.; Khan, F.A.; Anwar, S. Static malware detection and attribution in android byte-code through an end-to-end deep system. *Future Gener. Comput. Syst.* **2020**, *102*, 112–126. [CrossRef]
106. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. DL-Droid: Deep learning based android malware detection using real devices. *Comput. Secur.* **2020**, *89*, 101663. [CrossRef]
107. Vu, L.N.; Jung, S. AdMat: A CNN-on-Matrix Approach to Android Malware Detection and Classification. *IEEE Access* **2021**, *9*, 39680–39694. [CrossRef]
108. Millar, S.; McLaughlin, N.; del Rincon, J.M.; Miller, P. Multi-view deep learning for zero-day Android malware detection. *J. Inf. Secur. Appl.* **2021**, *58*, 102718. [CrossRef]
109. Acar, Y.; Stransky, C.; Wermke, D.; Weir, C.; Mazurek, M.L.; Fahl, S. Developers need support, too: A survey of security advice for software developers. In Proceedings of the 2017 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 24–26 September 2017; pp. 22–26. [CrossRef]
110. Mohammed, N.M.; Niazi, M.; Alshayeb, M.; Mahmood, S. Exploring software security approaches in software development lifecycle: A systematic mapping study. *Comput. Stand. Interfaces* **2017**, *50*, 107–115. [CrossRef]
111. Weir, C.; Becker, I.; Noble, J.; Blair, L.; Sasse, M.A.; Rashid, A. Interventions for long-term software security: Creating a lightweight program of assurance techniques for developers. *Softw. Pract. Exp.* **2020**, *50*, 275–298. [CrossRef]

112. Alenezi, M.; Almomani, I. Empirical analysis of static code metrics for predicting risk scores in android applications. In Proceedings of the 5th International Symposium on Data Mining Applications, Cham, Switzerland, 29 March 2018; Springer: Cham, Switzerland, 2018; pp. 84–94. [CrossRef]
113. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A. Lightweight detection of android-specific code smells: The adoctor project. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; pp. 487–491. [CrossRef]
114. Pustogarov, I.; Wu, Q.; Lie, D. Ex-vivo dynamic analysis framework for Android device drivers. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1088–1105. [CrossRef]
115. Amin, A.; Eldessouki, A.; Magdy, M.T.; Abdeen, N.; Hindy, H.; Hegazy, I. AndroShield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information* **2019**, *10*, 326. [CrossRef]
116. Tahaei, M.; Vaniea, K.; Beznosov, K.; Wolters, M.K. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, Yokohama, Japan, 8–13 May 2021; pp. 1–17. [CrossRef]
117. Goaër, O.L. Enforcing green code with Android lint. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops, Melbourne, VIC, Australia, 21–25 September 2020; pp. 85–90. [CrossRef]
118. Habchi, S.; Blanc, X.; Rouvoy, R. On adopting linters to deal with performance concerns in android apps. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 6–16. [CrossRef]
119. Wei, L.; Liu, Y.; Cheung, S.C. OASIS: Prioritizing static analysis warnings for Android apps based on app user reviews. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 672–682. [CrossRef]
120. Luo, L.; Dolby, J.; Bodden, E. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15–19 July 2019. [CrossRef]
121. Wang, Y.; Xu, G.; Liu, X.; Mao, W.; Si, C.; Pedrycz, W.; Wang, W. Identifying vulnerabilities of SSL/TLS certificate verification in Android apps with static and dynamic analysis. *J. Syst. Softw.* **2020**, *167*, 110609. [CrossRef]
122. Gupta, A.; Suri, B.; Kumar, V.; Jain, P. Extracting rules for vulnerabilities detection with static metrics using machine learning. *Int. J. Syst. Assur. Eng. Manag.* **2021**, *12*, 65–76. [CrossRef]
123. Kim, S.; Yeom, S.; Oh, H.; Shin, D.; Shin, D. Automatic Malicious Code Classification System through Static Analysis Using Machine Learning. *Symmetry* **2021**, *13*, 35. [CrossRef]
124. Bilgin, Z.; Ersoy, M.A.; Soykan, E.U.; Tomur, E.; Çomak, P.; Karaçay, L. Vulnerability Prediction From Source Code Using Machine Learning. *IEEE Access* **2020**, *8*, 150672–150684. [CrossRef]
125. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated vulnerability detection in source code using deep representation learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; pp. 757–762. [CrossRef]
126. Chernis, B.; Verma, R. Machine learning methods for software vulnerability detection. In Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, Tempe, AZ, USA, 21 March 2018; pp. 31–39. [CrossRef]
127. Wu, F.; Wang, J.; Liu, J.; Wang, W. Vulnerability detection with deep learning. In Proceedings of the 2017 3rd IEEE International Conference on Computer and Communications (ICCC), Chengdu, China, 13–16 December 2017; pp. 1298–1302. [CrossRef]
128. Pang, Y.; Xue, X.; Wang, H. Predicting vulnerable software components through deep neural network. In Proceedings of the 2017 International Conference on Deep Learning Technologies, Chengdu, China, 2–4 June 2017; pp. 6–10. [CrossRef]
129. Garg, S.; Baliyan, N. A novel parallel classifier scheme for vulnerability detection in android. *Comput. Electr. Eng.* **2019**, *77*, 12–26. [CrossRef]
130. Ponta, S.E.; Plate, H.; Sabetta, A.; Bezzi, M.; Dangremont, C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 26–27 May 2019; pp. 383–387. [CrossRef]
131. Namrud, Z.; Kpodjedo, S.; Talhi, C. AndroVul: A repository for Android security vulnerabilities. In Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, 4–6 November 2019; pp. 64–71.
132. Cui, J.; Wang, L.; Zhao, X.; Zhang, H. Towards predictive analysis of android vulnerability using statistical codes and machine learning for IoT applications. *Comput. Commun.* **2020**, *155*, 125–131. [CrossRef]
133. Zhuo, L.; Zhimin, G.; Cen, C. Research on Android intent security detection based on machine learning. In Proceedings of the 2017 4th International Conference on Information Science and Control Engineering (ICISCE), Changsha, China, 21–23 July 2017; pp. 569–574. [CrossRef]