Hindawi
Mobile Information Systems
Volume 2021, Article ID 8896013, 18 pages
https://doi.org/10.1155/2021/8896013

Review Article

A Survey of Android Malware Static Detection Technology Based on Machine Learning

Qing Wu, Xueling Zhu, and Bo Liu

College of Computer, National University of Defense Technology, Changsha 410073, China

Correspondence should be addressed to Bo Liu; kyle.liu@nudt.edu.cn

Received 8 July 2020; Revised 4 February 2021; Accepted 24 February 2021; Published 4 March 2021

Academic Editor: Claudio Agostino Ardagna

With the rapid growth of Android devices and applications, the Android environment faces more security threats. Malicious applications stealing users' privacy information, sending text messages to trigger deductions, exploiting privilege escalation to control the system, etc., cause significant harm to end users. To detect Android malware, researchers have proposed various techniques, among which the machine learning-based methods with static features of apps as input vectors have apparent advantages in code coverage, operational efficiency, and massive sample detection. In this paper, we investigated Android applications' structure, analysed various sources of static features, reviewed the machine learning methods for detecting Android malware, studied the advantages and limitations of these methods, and discussed the future directions in this field. Our work will help researchers better understand the current research state, the benefits and weaknesses of each approach, and future technology directions.

1. Introduction

The number of Android applications continues to grow. As of June 2020, there are more than 2.99 million apps in Google Play [1]. At the same time, the number of malicious apps is also increasing rapidly. Experts at AV-TEST Institute had counted around 1.8 million new malicious apps in the first half of 2020 [2]. The malicious apps steal personal information from end users, automatically call and send text messages to trigger deductions, and exploit root privileges to control the system, causing massive harm to end users [3]. Various detection techniques for Android malware have been proposed one after another to cope with these security threats. The machine learning-based detection method is one of the efficient ways. This type of approach has several advantages:

(i) Comprehensiveness of the detection: when detecting malware, traditional methods usually pay close attention to certain malicious functions, such as the framework proposed by Xu et al. [4] against privilege escalation and the technique offered by He et al. [5] for privacy leakage analysis. Most machine learning methods can detect various malicious behaviours at one time and carry out a classification of multiple families, which can be used as a universal and comprehensive detection means.

(ii) Accuracy of the detection: benefiting from the rapid development of machine learning algorithms, the accuracy of identifying malicious applications based on such methods is increasing. Detection frameworks based on SVM, Naïve Bayes, Perceptron, and deep neural network algorithms are continually being proposed. They perform better and better in dealing with malicious application identification, multiclassification, and malicious code location.

(iii) Reduction of dependence on experts: traditional detection methods rely heavily on the rich experience of human experts. However, through feature extraction at many different levels with less experience knowledge, more malicious application patterns can be identified by machine learning methods, which are more conducive to discovering new malicious software and improving detection efficiency.

In the implementation of Android malware detection using machine learning, the two primary sources of the feature are static extraction and dynamic extraction [6]. Static features are extracted from the manifest, Dalvik bytecode, native code, sound, image, and other reversed APK files. Dynamic features are collected from the log records, code execution paths, variable value tracking, sensitive function calls, and other behaviours in the process of application execution by running APK files in a monitored environment.

Although the detection method based on static features has some limitations compared with those based on dynamic ones, such as it is challenging to combat code obfuscation, it also has distinct advantages:

(i) Full code coverage: static feature extraction can cover all code and all resource files by scanning code or symbolic pseudoexecution. In contrast, dynamic feature extraction can hardly cover all code execution paths. Many applications require users to provide login credentials to use most of the features, making it difficult to detect all the functions in dynamic execution, resulting in incomplete feature extraction.

(ii) Reliable detection efficiency: static feature extraction will complete the detection task in the expected time because it does not need to run the application. In contrast, dynamic feature extraction requires triggering various functions in code execution, which will consume lots of time. While the application is running, it takes some time to simulate a click-through interface. The program may perform a very complex computation or enter an infinite loop. These conditions make it difficult for the detection task to be completed within the specified time frame.

(iii) Unperceived by malicious code: static detection does not require the codes' execution, so malicious applications cannot recognize that they are under check. Although some malware attempt to make the static analysis more challenging by setting up interference codes, these added codes may themselves be an identifier to assist in identifying malicious applications.

(iv) Easier to generate generic fingerprint identification: static malicious sample analysis is more inclined to extract features with invariance and universality. In contrast, dynamic analysis is very likely to be affected by the operating environment. Statically extracted features are suitable for fingerprinting and can be used for the rapid predetection of large-scale malicious applications.

There are some surveys about Android malware detection published in the past few years. The authors in [7–9] analysed the Android security mechanism and typical malware detection methods. The authors in[10–12] focused on applying deep learning algorithms such as Restricted Boltzmann Machines, Convolutional Neural Network, Deep Belief Network, Recurrent Neural Network, and Deep Autoencoder to malware detection and analysed the advantages and results achieved. The authors in [13] is mainly concerned with the analysis of Android malware variants' detection methods. The authors in [14] investigated Android malware detection and protection technology based on data mining algorithms. The study in [15, 16] collected the literature research of the past few years, systematically analysed the static detection technology, and discussed datasets, features, algorithms, empirical experiments, and performance measures. The study in [17] introduced the Android architecture, security mechanism, malware classification, and entire detection process, including sample collection, data preprocessing, feature selection, machine learning model construction, and experimental evaluation. The study in [18–20] comprehensively discussed static, dynamic, and hybrid detection techniques. The study in [21] mainly focused on mobile malware detection techniques, analysed signature-based detection, anomaly-based detection, and other traditional detection methods. The study in [22] discussed various threats and the current Android platform security state, introduced three attack types, explained the factors contributing to the increase of malware, and analysed defensive mechanisms of Android protection.

The above surveys have done excellent work, but there are still some aspects that can be improved. For example, the sources of static features, the challenges of obfuscation technology to static analysis, and the deterioration issues of machine learning models are not investigated in detail. Our work aims to provide a comprehensive survey about Android malware static detection based on machine learning technologies. To this end, we searched in IEEE, ACM, Springer, Wiley, Hindawi, and other databases and used Google Scholar and DBLP to find the related papers. It is worth noting that we only use research papers from DBLP since 2016 in statistics of static feature types, machine learning algorithms, dataset usage, papers' number, and evaluation metrics. The reason is that DBLP has become an authentic database, and the papers it saves are of relatively high quality and are relatively few in number. We can manually check the detection technology, algorithm model, and evaluation method used in each article to form accurate statistical results. Based on the papers we collected on Android static malicious application detection, we finish this survey work.

In our work, we analysed the Android application static features and the typical obfuscation methods, discussed machine learning algorithms suitable for Android malware detection, explained evaluation metrics of machine learning models and sustainability issues, investigated the technical route, advantages, and disadvantages of the existing research, and made an outlook on the possible future research directions in this field. The main contributions can be summarized as follows:

(i) We carried out a comprehensive review of Android malware's various static detection methods based on machine learning. The basic principles, feature sources, datasets, performance metrics, contributions, and limitations of the methods were compared vertically.

(ii) We analysed the Android application composition, the source of static feature extraction, and the feature vector generation method in detail.

(iii) The limitations of current methods were discussed, and the future development directions were prospected.

2. Android App's Static Features' Analysis

2.1. Android App's Static Features. Android apps are composed of loosely coupled components bound together by the manifest file. The manifest file describes each component, application metadata, platform requirements, external libraries, required permissions, etc. Activity, service, content provider, and broadcast receiver [23] are the basic components that make up an Android app. Each of them performs different functions.

An Android app is released in the APK package form, a zip archive mainly composed of assets, lib, res, manifest, Dalvik bytecode, and resource files [24]. These files are commonly used as static feature sources. According to each file's role, the feature vector's extraction method and expression are different. As shown in Figure 1, there are mainly the following types of features.

2.1.1. Permission Features. From AndroidManifest.xml, the various permissions that an app requires to use during its runtime are declared in AndroidManifest.xml. Extracting the permissions listed in the file can help determine whether the app is malicious. The Android system provides about 250 kinds of permissions [25], resulting in the feature vector taking a form as a binary vector of about 250 bits long.

2.1.2. Component Features. From AndroidManifest.xml and classes.dex files, the four basic components need to be registered in AndroidManifest.xml. They will be declared and created by calling related system calls in the classes.dex file. The types and quantities of components used in the app will generally be included in the formed feature vectors.

2.1.3. Intent Features. From AndroidManifest.xml and classes.dex files, intents are used to pass messages between components. When an intent is passed to a component, a predefined call-back function is executed to process this intent. In the formation of feature vectors, intents are often used with components because they can help analyse the association between two components.

2.1.4. Constant String Features. From resource and dex files, the strings.xml resource file stores the developer-defined strings, and the dex file stores the string defined by the Smali code. Extracting the content and frequency of these strings from these files can reflect the app's characteristics. Considering that there are many types of strings and some of them are very long, the hash operation is generally carried out first before subsequent processing when forming feature vectors.

2.1.5. Resource File Features. From res and assets directories, including sounds, images, and layout files, many repackaged, cloned malicious applications do not modify resource files so that such files can be used as static features.

2.1.6. Opcode and API Features. From dex files, the frequency of Dalvik opcodes and API calls reflecting developers' programming habits is very suitable for generating detection features. In general, the occurrence number of opcodes and APIs presents a significant distinction between malware and benign apps. Feature vectors can be generated by measuring the frequency of continuous N opcodes and APIs.

2.1.7. Native Code Frequency Features. From .so files, many malware use native instructions to perform malicious operations, for the reason that compiled codes of these instructions are more difficult to decompile, which will bring many obstacles to the detection work. Extracting the arm opcode frequency and system call invocation frequency from the .so file can significantly help the detection work.

2.1.8. Control Flow Graph and Data Flow Graph Features. From dex files, the control flow graph and data flow graph can be obtained by analysing the instruction invocation relationship and data flow direction in the codes. Transforming these graphs into vector representation through graph embedding and other methods can distinguish malicious from benign behaviours well.

The above static features are used differently according to detection scenarios. We counted the research papers on machine learning-based static detection of Android malware in the DBLP database between Jan. 2019 and Nov. 2020. After eliminating repetitive and irrelevant articles, we obtained 118 papers. The statistics of static feature usage are shown in Figure 2. All features were used 277 times, of which API features were the most frequently adopted and native opcodes the least. This situation illustrates that these features play different roles in identifying malicious apps.

In recent years, the academic community has paid great attention to the sustainability and deterioration issues faced by learning-based detection models [26, 27]. A key to achieving high sustainability of a classifier lies in the underlying features being able to differentiate benign apps from malware for a long period. Zhang [28] built an API Graph based on API features to enhance malware classifier performance and slow down model aging with the similarity information among evolved Android malware. Xu [29] built and dynamically expanded the feature set based on APIs to

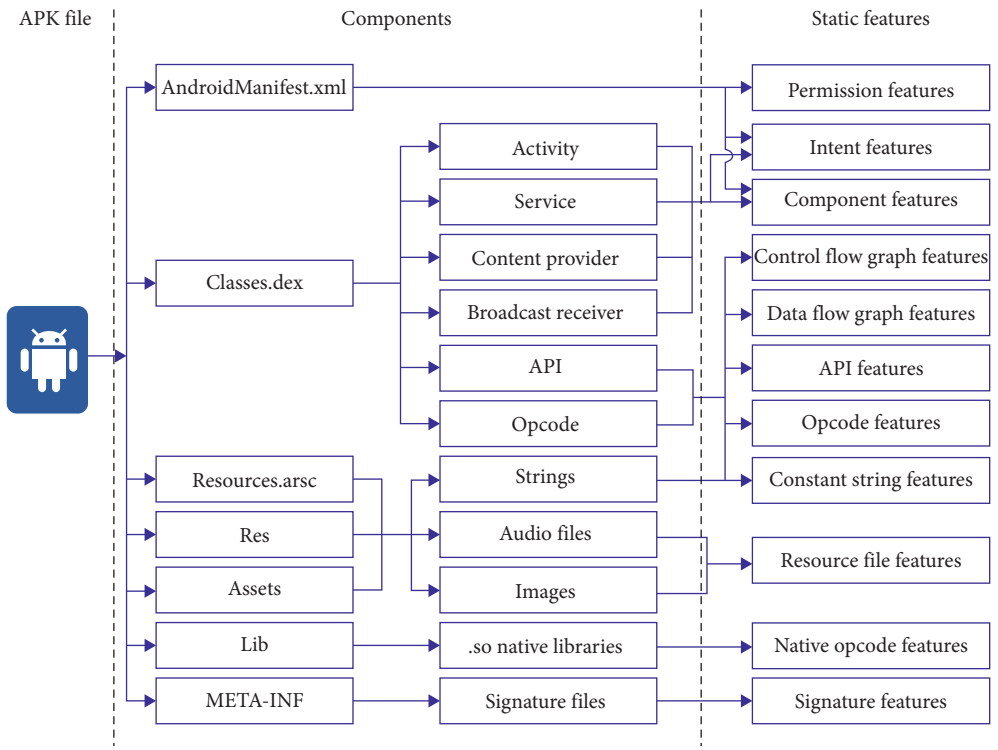

Figure 1: Android app's composition and static feature types.

train a variety of online learning models in a model pool that can determine the drift samples and update the aging model in a weighted voting style.

2.2. Android Malware Datasets with Static Features. To facilitate the analysis and utilization of static features, some researchers provided Android malware datasets with static features.

Drebin dataset [30] contains 5560 apps from 179 different malware families in the period of August 2010 to October 2012, providing eight types of features including hardware features, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network addresses.

MalGenome dataset [31] provided 1260 Android malware samples in 49 different malware families from Aug 2010 to Oct 2011 and analysed the characteristics of malicious applications during installation and activation, including some static features, such as used permissions.

AMD dataset [32] contains 24,553 samples, categorized in 135 varieties among 71 malware families ranging from 2010 to 2016. The authors use the n-gram sequence of Dalvik bytecode to denote an Android app feature. The feature categories include Dalvik bytecode opcode sequences, Java VM-type signatures, string value of const-string instructions, and type signatures for invoked functions.

ISCX Android Botnet Dataset [33] includes 1929 botnet samples belonging 14 different families collected between 2010 and 2014. The static features they analysed are mainly embedded and obfuscated URLs.

Mystique [34] generated over 10,000 samples of Android malware by combining different attack and evasion features. Their features include entry points for malicious attack behaviors, permissions, intent filters, and source and sink APIs.

PRAGaurd dataset [35] contains 10,479 samples, obtained by obfuscating the MalGenome [31] and the Contagio [36] datasets with seven different obfuscation techniques that obfuscate some static features to avoid inspection by analysis tools.

Datasets such as AndroZoo [37], Contagio [36], VirusShare [38], and VirusTotal [39] provide detailed classification descriptions, but do not provide clear static features, which need to be extracted by the researcher.

2.3. Static Feature's Obfuscation Technology. To make the applications more difficult for reverse engineers to understand or be checked by detection tools, malicious application developers will obfuscate static features to a certain extent. The main methods are as follows.

2.3.1. Identifier Renaming. The name of the identifier in the code is usually meaningful, and developers generally follow similar naming rules. To make it difficult for reverse engineers to understand code logic and reduce potential information leakage, the malware developers may use random strings to replace the identifiers' names [40]. This kind of renaming is more effective for humans than for machine analysis. Since the identifier renaming method does not

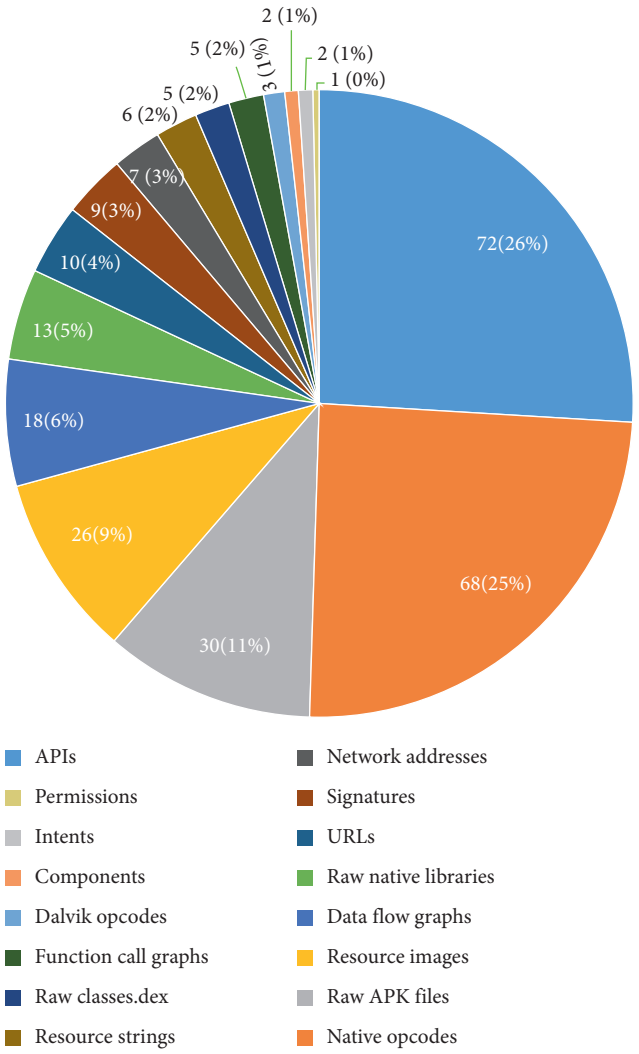

Figure 2: Statistics of static feature usage in static machine learning-based research papers from Jan. 2019 to Nov. 2020 in DBLP.

work with Android API functions, Ficco [41] bypassed this technique's obstacle by comparing the API call sequence.

2.3.2. String Confusion. Encoding and encrypting the strings in the resource and code files and decrypting and reading them at runtime can achieve the effect of bypassing the detection [42]. In this case, the encryption function and encoding function should be included in the investigation scope and examined carefully. For binary programs, previous works [43, 44] had proposed various approaches to identify cryptographic functions in a program, such as AES, DES, and RC4. For Android apps, Suarez-Tangil et al. [45] dealt with this kind of obfuscation by strengthening the checking of cryptographic system API calls.

2.3.3. Call Indirection. The malicious developer modifies the original method's invocation entry, inserting a new randomly named method before invoking the original. Calling the original method through the newly inserted method will add many unrelated nodes to the control flow graph, rendering some detection methods based on these technologies ineffective. Garcia [46] constructed a detection framework that used sensitive API usage, data flows between APIs, intent action, and package API usage features to detect malicious apps that use various obfuscation techniques, including call indirections.

2.3.4. Junk Code Insertion. Malicious developers often add junk instructions to the code files, such as NOP, jump, and register operation instructions, to increase the code's complexity and eliminate the original static characteristics. Some detection methods based on opcode statistics [47, 48] will be disturbed by this obfuscation technology. In comparison, the method based on the source-sink data flow [49] will normally work because the influence of the inserted junk instructions on the data flow is limited.

2.3.5. Dynamic Code Loading. The Android app can load native code and additional Dalvik bytecode from local resource files, other apps, or remote networks. Malicious developers often load dynamic code through the Ldalvik/system/DexClassLoader function or use the Ljava/lang/reflect/Method function to make reflection calls, making it challenging to locate malicious code. The typical detection method is to check the app's sensitive functions to judge whether it is malicious. Poeplau [50] constructed the super control flow graph (sCFG) to check sensitive function calls and parameter passing to judge whether dynamic calls are vulnerable or malicious.

For the above obfuscation methods, various static detection techniques have different countermeasures. Generally speaking, it is difficult for malicious developers to obfuscate all static features. Static detection typically improves the detection accuracy by using multiple features comprehensively [45, 46, 51, 52].

3. Machine Learning Algorithms and Related Performance Metrics

Using machine learning algorithms for Android malware classification must consider the algorithm efficiency and accuracy. Generally speaking, shallow machine learning algorithms are used to construct simple classification models, with the advantages of simple implementation and fast running, but the precision is relatively low. Using complex machine learning models, detection accuracy is high, but efficiency is not desirable. Many methods are compromised between the two.

Logistic regression, Naïve Bayes, Support Vector Machine, *k*-Nearest Neighbour, Decision Tree, and Random forest are suitable shallow machine learning algorithms for detecting Android malware. (1) Logistic regression [53] is a generalized linear regression analysis model for estimating a particular thing's probability. The purpose of logistic regression is to find a best-fit model that describes the relationship between the dependent variable and a set of independent variables. (2) The Naïve Bayes [54] is based on

the Bayesian theorem and assumes that the feature conditions are independent. The Bayesian network [55], also known as the reliability network, is suitable for expressing and analysing uncertain and probabilistic events and can make inferences from incomplete, inaccurate, or uncertain knowledge. (3) The Support Vector Machine (SVM) [56] is a generalized linear classifier that classifies data in a supervised learning manner. The decision boundary is the maximum-margin hyperplane for solving the learning samples. The SVM can perform nonlinear classification by the kernel method and is one of the common kernel learning methods. (4) The idea of the k-Nearest Neighbour (kNN) [57] algorithm is to find k nearest neighbour samples of a sample. Most of them belong to a specific category and have similar attributes. (5) The Decision Tree [58] is a nonparametric supervised learning method that can summarize decision rules from a series of data with features and labels and use the tree's structure to present these rules to solve classification and regression problems. (6) The Random forest [59] is a classifier that contains multiple decision trees. Its output category is determined by the output categories of most decision trees.

Deep neural network models suitable for detecting Android malware mainly include Deep Belief Network, Convolutional Neural Network, Recurrent Neural Network, Generative Adversarial Network, Multimodal Machine Learning, Multiple Kernel Learning, Graph embedding, and Representation learning. (1) The Deep Belief Network (DBN) [60] is a probabilistic generation model that establishes a joint distribution between observation data and labels. It is composed of multiple restricted Boltzmann machines. The layer-by-layer training method is used to solve the problem that the traditional neural network training method is not suitable for the multilayer network. (2) The Convolutional Neural Network (CNN) [61] is a feedforward neural network with convolutional computation and deep structure. It has the ability of representation learning and can perform translational shift-invariant classification of input information according to its hierarchical structure. (3) The Recurrent Neural Network (RNN) [62] is a recursive neural network with sequence data as input and is recursive in the sequence's evolution direction. It is Turing-complete and has advantages when learning nonlinear features of sequences. (4) The Generative Adversarial Network (GAN) [63] is one of the most promising methods for unsupervised learning in complex distribution. The model produces a reasonably good output through mutual game learning between the framework's generated and discriminant models. (5) Multimodal Machine Learning (MMML) [64] aims to realize the ability to process multisource modal information. It can perform various tasks, such as representation learning, collaborative learning, and modal conversion. (6) Multiple Kernel Learning [65] uses multikernel functions to map and combine various features so that the data can be more reasonably expressed in the new combined space. (7) Graph embedding [66] maps high-dimensional sparse graph data into low-dimensional dense vectors, which can solve the problem that the graph data is difficult to feed into machine learning algorithms efficiently. (8) Representation learning [67] is a learning feature representation technique that transforms raw data into a form that can be effectively recognized by machine learning. It avoids the hassle of manually extracting features, allowing computers to learn to use features, while also learning how to extract features.

In the DBLP database, there are 225 classifier algorithms used in 118 papers of static machine learning-based Android malware detection from Jan. 2019 to Nov. 2020. Figure 3 shows these algorithms' distribution. There are more shallow learning algorithms used in the research. The SVM algorithm ranks first with used times of 36. In the deep neural network models, the CNN algorithm ranks first with used times of 16.

To evaluate the performance of machine learning algorithms, researchers developed various evaluation metrics. The traditional metrics are shown in the following equation:

$$\begin{aligned}
\text{accuracy} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},\\
\text{precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}},\\
\text{recall rate} &= \frac{\text{TP}}{\text{TP} + \text{FN}},\\
\text{false positive rate (FPR)} &= \frac{\text{FP}}{\text{TN} + \text{FP}},\\
\text{false negative rate (FNR)} &= \frac{\text{TP}}{\text{TP} + \text{FP}},\\
F_1 \text{ score} &= \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}},
\end{aligned} \tag{1}$$

where TP is the number of true positive samples, FN is the number of false negative samples, FP is the number of false positive samples, and TN is the number of true negative samples. Another commonly used evaluation metric is AUC (Area Under Curve), which is defined as the area enclosed by the receiver operating characteristic curve and the coordinate axis. The higher the AUC value, the better the effect of the model.

The ML-based Android malware detection models are prone to deteriorate for the rapid emergence of malicious apps. Researchers have introduced a series of metrics to evaluate the model's sustainability, including AUT (Area Under Time) [68], Stability [69], Algorithm Credibility [70], and Algorithm Confidence [70].

AUT is a metric proposed by Pendlebury [68], which defines the area under the performance curve over time to represent the model's sustainability, as shown in the following equation:

$$\text{AUT}(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(k+1) + f(k)]}{2}, \tag{2}$$

where f is the performance metric (e.g., F_1-score, Precision, Recall, etc.), N is the number of test slots, and $f(k)$ is the performance metric evaluated at the time k. The perfect classifier with robustness to time decay has an AUT metric closer to 1.

The Stability metric proposed by Cai [69] of a classifier indicates how stable the classifier is without retraining or any other model updates and is measured by a tuple $<e^s, n>$, where e^s is classification accuracy the classifier achieves in an average case when trained on apps of year x and tested on apps of year $x+n$, $n \geq 1$.

Jordaney [70] proposed Algorithm Credibility and Algorithm Confidence metrics to assess the decision of the classifiers and identify aging classification models before the performance starts to degrade. They first defined a p-value $p^C_{z^*}$ for an object z^* in a set of objects K, which means the proportion of objects in the class K that are at least as dissimilar to other objects in a set C as z^*. The p value is defined as the following equation:

$$\begin{aligned} \alpha_{z*} &= A_D(C, z^*), \\ \forall i \in K \cdot \alpha_i &= A_D\left(\frac{C}{z_i, z_i}\right), \\ p^C_{z^*} &= \frac{\left|\left\{j\colon \alpha_j \geq \alpha_{z*}\right\}\right|}{|K|}, \end{aligned} \tag{3}$$

where $A_D(C/z, z)$ tells how different an object z is from a set C. Based on the p value, they then defined Algorithm Credibility $A_{\text{Cred}}(z^*)$ and Algorithm Confidence $A_{\text{Conf}}(z^*)$ as the following equation:

$$\begin{aligned} A_{\text{Cred}}(z^*) &= p^C_{z^*}, \\ A_{\text{Conf}}(z^*) &= \frac{1 - \max(p(z^*))}{A_{\text{Cred}}(z^*)}, \end{aligned} \tag{4}$$

where $A_{\text{Cred}}(z^*)$ is defined as the p value for the test object z^* corresponding to the label chosen by the algorithm under analysis and $A_{\text{Conf}}(z^*)$ is defined as 1.0 minus the maximum p value among all p values except the p value chosen by the algorithm. Through these two metrics, it is possible to understand whether the choices made by an algorithm are supported with statistical evidence, making it easier to discover concept drift and model aging issues.

4. Literature Review

It is worth noting that many papers use more than one machine learning algorithm. We analysed in detail which algorithm the researchers work on, or which algorithm is discussed or improved, or which algorithm has achieved the best results in the author's experiment, as a classification basis.

4.1. Logistic Regression. Tiwari and Shukla [71] proposed a method to detect android malware using permissions and API. The authors divided the detection of malicious applications into four steps: reverse engineering, feature extraction, feature vector generation, and classification. They reversed the APK file with reverse engineering tools and obtained AndroidManifest.xml and Smali files. Permissions from AndroidManifest.xml and APIs from Smali files were extracted to generate combined feature vectors. They achieved 96.56% accuracy for combined features using the logistic regression algorithm.

Milosevic et al. [72] presented two machine learning-aided approaches for static analysis of Android malware. The first approach is based on permission features. The Precision of 0.823, Recall of 0.822, and F-score of 0.821 are achieved using the logistic regression model as a classifier. The other approach extracts features from code files. Android apps are first reversed into multiple Java files; then, the natural language processing method is used to generate feature vectors through the bag-of-words model. SVM with SMO, logistic regression, simple logistic regression, and AdaBoostM1 with SVM algorithms are integrated into the framework, and they achieved Precision, Recall, and F-score values of 0.958, 0.957, and 0.956, respectively.

4.2. Naïve Bayes and Bayesian Network. Yerima [73] proposed an effective Bayesian classification method to deal with Android malware. The authors developed three tools: API call detectors, command detectors, and permissions detectors. They extracted features from API call, resources, assets, libraries, and Permissions, respectively. Through experimental data analysis, top-n attributes with the most discriminative ability are selected to form effective features. Finally, a Bayesian classifier is trained to make decisions. Their experimental dataset contains 1000 malware samples and 1000 benign apps. Under the condition of using 20 attributes as classification features, the performance reaches Accuracy, Precision, and AUC with 0.921, 0.935, and 0.97223, respectively.

Sanz [74] proposed a method for categorizing Android apps through machine learning techniques. They extracted three different feature sets: the frequency of the printable strings, the various permissions of the app itself, and the app's permissions gathered from the Android market. They used Random Forest, J48, kNN, Bayesian Networks, Naïve Bayes, and SVM as classifiers to carry out experiments on 820 samples of seven different families and concluded that Bayes TAN was the best classifier obtaining an AUC of 0.93.

4.3. Support Vector Machine. Zhao [75] presented a Feature Extraction and Selection Tool (FEST) based on machine learning approaches for malware detection. According to the predefined rules, the authors first implemented a feature extraction tool named AppExtractor. Then, they proposed a feature selection algorithm named FrequenSel, which selects features by finding the difference of permission and API frequencies between malware and benign apps. In experiments, the authors tested various classification algorithms and found that the SVM algorithm was the best.

Nissim et al. [76] introduced a framework named ALDROID based on the active learning method. Their framework aimed to select only new informative applications (benign and especially malicious) to reduce security experts' labelling efforts. They first extracted permissions from manifest files, counted the number of activities, services, receivers, and content providers as features, and finally

used the SVM as a classification algorithm. The highest performance achieved an Accuracy of 98.8%, a TPR of 90%, and an FPR of 0.0008.

Xu et al. [77] analysed intercomponent communication- (ICC-) related characteristics and proposed a method of identifying malware called ICCDetector, which could capture interactions between components or across app boundaries. The ICCDetector outputs all ICC sources and sinks from the APK file. These sources-sinks and other ICC-related features can be extracted to form feature vectors. The authors experimented with a dataset of 5264 malware and 12,026 benign apps with the SVM algorithm as a classifier. They achieved an accuracy of 97.4%, with a lower FPR of 0.67%. Furthermore, they discovered 43 new malicious apps from the benign dataset using the ICCDetector tool.

4.4. k-Nearest Neighbour. Wu [78] developed a system called DroidMat, considering the static information, including permissions, component deployments, intent messages, and API calls for characterizing the Android app's behaviour. Firstly, the DroidMat extracts the information from the manifest file and regards components as entry points for drilling down to trace API calls. Next, it applies the k-means algorithm to enhance malware modelling capability. The number of clusters is decided by the Singular Value Decomposition (SVD) method on the low-rank approximation. Finally, it uses the kNN algorithm to classify the app as benign or malicious. The model achieved 97.87% Accuracy, 87.39% Recall, 96.74% Precision, and 91.83% F_1-score on the dataset from the "Contagio mobile" site.

Baldini and Geneiatakis [79] investigated the simple machine learning classifiers' performance. The authors performed an extensive comparison using various well-known distance measures over the Drebin dataset. Results show that the distance measure's proper choice can provide a significant enhancement to the classification accuracy. Specifically, Hamming and CityBlock can boost the classifiers' performance in mobile malware detection. For instance, CityBlock can improve the kNN algorithm's false positive rate by up to 33% compared to the Euclidean distance.

4.5. Decision Tree and Random Forest. Sanz [80] developed a method that extracts several features from the manifest file to build machine learning classifiers. These feature sets are the permissions required for the app and the features under the uses-features group. They generated an input vector for all possible permissions, then used Naïve Bayes, J48, Random Forest, and other classifiers to perform experiments, and achieved the best results on the Random Forest containing 100 trees with an AUC of 98% and an Accuracy of 94.83%.

Canfora [47] employed the probability of n consecutive opcodes in the code segment as features such as 2-opcodes' sequences [(move, invoke), (invoke, add), ...], and each parenthesis is a component of the feature vector. If (move, invoke) probability is 0.001, (invoke, add) probability is 0.003, etc. Then, the value of this feature vector is [0.001, 0.003, ...]. The authors trained two classifiers, SVM and Random Forest, to do binary classification. Results show that 97% accuracy can be obtained on average when 2-opcodes is used. Kang [48] did a similar job. They also used n-opcodes as a feature to test Naïve Bayes, SVM, Partial Decision Tree, and Random Forest classification algorithms. For $N = 3$ and $N = 4$, the SVM shows the best F_1-score of 98%, and Random Forest shows the best performance in terms of both training and prediction speeds.

4.6. Deep Belief Network. Su [81] proposed DroidDeep, a malware detection approach for the Android platform based on the deep learning model. Requested permission, used permission, sensitive API call, action, and app components were used as static features. The authors built a Deep Belief Network model to learn the features. In experiments with 3986 benign apps and 3986 malware, DroidDeep achieved a 99.4% detection accuracy.

Li [82] proposed a weight-adjusted malware detection approach named DroidDeepLearner. The approach uses both dangerous API calls and risky permission combinations as features to build a Deep Belief Network model, which can automatically distinguish malware from benign ones. The results show that their approach achieves over 90% accuracy, with only 237 features on the Drebin dataset.

4.7. Convolutional Neural Network. Zhang et al. [83] proposed DeepClassifyDroid, which takes a three-step approach as follows: feature extraction, feature embedding, and detection, to discriminate malware from android apps based on the Convolutional Neural Network. They embedded permissions, intent-filters, API calls, and constant strings from the disassembled codes in a unified joint-vector space. Then, they trained a CNN model with two convolutional layers, a pooling layer, and a full connection layer to learn these vectors. Experiments show that the approach achieves an accuracy of 97.4% with few false alarms on a dataset of 5546 malware and 5224 benign apps.

Nix and Zhang [84] investigated the effectiveness of CNN and LSTM for Android apps' classification using system API call sequences. They encoded each API call using a one-hot vector, and then, each segment is encoded by a matrix of size $n \times m$, which serves as the input to a CNN model. They compared their CNN model with LSTM and other n-gram-based methods. Both CNN and LSTM significantly outperformed n-gram-based methods, and the performance of CNN is the best. The experiments show that the results achieve 99.4% Accuracy, 100% Precision, and 98.3% Recall on a dataset of 1016 APK files.

Ganesh [85] proposed a CNN-based deep learning model can extract the patterns of malware. They demonstrate that CNN is appropriate for malware detection by using data transformation. The APK file is parsed and decompiled using Androguard and Smali disassembler. Then, the extracted manifest file is converted into a 12×12 vector of permissions, which is fed into the trained CNN model. Their solution identifies malware with 93% accuracy

on a dataset of 2500 Android apps, of which 2000 were malicious and 500 were benign.

4.8. Recurrent Neural Network. Amin et al. [86] proposed an end-to-end deep learning architecture that detects and attributes Android malware via opcodes extracted from bytecode files. They confirmed that bidirectional long short-term memory (BiLSTM) neural networks can be effectively applied to detect Android malware's static behaviour without using handcrafted features. Experimental results report an accuracy of 99.9% and an F_1-score of 99.6% on a large dataset of more than 1.8 million Android applications.

Lee et al. [87] proposed a stacked RNNs and CNNs-based classification model for learning the generalized correlation between obfuscated strings from the package's and certificate owner's name. The model uses the embedded method and the GRU unit to extract features and uses additional CNN units to optimize the extraction process. Their experiments demonstrate that the feature extraction process is robust to obfuscation and sufficiently lightweight for Android devices and that the CNN-RNN method improved classification performance by 16% against n-gram features and reduced training time by 50% against an RNN model.

Ma [88] proposed Droidetec, a deep learning-based method for android malware detection and malicious code localization, to model an application program as a natural language sequence. Droidetec adopts a depth-first algorithm to extract API sequences from the Android app as features. Based on that, the BiLSTM network is utilized for malware detection. Each unit in the extracted behaviour sequence is inventively represented as a vector, allowing Droidetec to automatically analyse the semantics of sequence segments and eventually discover the malicious code. Experiments with 9616 malicious and 11,982 benign programs show that Droidetec reaches an accuracy of 97.22% and an F_1-score of 98.21%. In all, Droidetec has a hit rate of 91% to find out malicious code segments properly.

4.9. Generative Adversarial Network. Amin [89] proposed a Generative Adversarial Network-based model to detect Android malware inspired by the famous two-player game theory for rock-paper-scissor problems. Inside the discriminator and generator, they incorporated LSTM as deep learning architecture to learn the opcode-based binary sequential data on a large and unlabelled dataset. The test data sequences are passed through the context window, determining the bytecodes' sequences that differ from the previously recorded ones. If the sequence mismatch at one or more locations, it would help evaluate and characterize the behaviour of the APK. The technique achieved an F_1-score of 99% with a receiver operating characteristic of 99%.

4.10. Multimodal Deep Learning and Multiple Kernel Learning. Kim [90] proposed a model based on Multimodal Deep Learning to detect Android malware. The model has five initial networks and a final network. The initial networks take five types of features as inputs and output intermediate vectors to train the final network. The features are refined using an existence-based or similarity-based extraction method, which can reflect Android apps' properties from various aspects. The authors achieved 98% and 99% accuracy on the VirusShare and MalGenome dataset, respectively.

Narayanan et al. [91] proposed MKLDroid, a unified framework that systematically integrates multiple views of apps for performing comprehensive malware detection and malicious code localization. MKLDroid uses a graph kernel to capture structural and contextual information from apps' dependency graphs and identify malicious code patterns in each view. Subsequently, it employs Multiple Kernel Learning (MKL) to find a weighted combination of the views, which yields the best detection accuracy. Besides multiview learning, MKLDroid can locate fine-grained malice code portions in dependency graphs. On benchmark datasets, MKLDroid achieves more than 97% F-measure. In the malicious code localization experiments on a dataset of repackaged malware, MKLDroid identifies all the malice classes with 94% average recall.

4.11. Graph Embedding. Pektas and Acarman [92] used the API call graph to represent all possible execution paths that a malicious app can track during its runtime. The API call graph was embedded into a low-dimension numeric vector feature set, which was introduced to the deep neural network. The authors built a CNN model as the classifier, which contains two convolution layers, one pooling layer, one flatten layer, and one dense layer, to decide whether a given app was malicious or benign. They evaluate four different graph embedding methods, namely, DeepWalk, Node2vec, Structural Deep Network Embedding, and Higher-Order Proximity Preserved Embedding, and found that SDNE provides more discriminative features. The result reached 98.86% accuracy when graph embedding size is equal to 32.

4.12. Representation Learning. Narayanan [93] designed a semisupervised representation learning framework named apk2vec to automatically generate a compact representation for a given app. Apk2vec is an integration technology that draws on the idea of doc2vec to embed an app into a vector. It can encompass information from multiple semantic views, use labels associated with apps, and combine RL and feature hashing to build apps' profiles efficiently. The evaluations with more than 42,000 apps demonstrated that apk2vec's app profiles perform well in malware detection, familial clustering, app clone detection, and app recommendation tasks.

All the methods that we have analysed are vertically compared, as shown in Table 1.

5. Limitations and Future Directions

5.1. Limitations of the Static Machine Learning-Based Detection Method. We have discussed various algorithms used in past research works. These algorithms perform very well

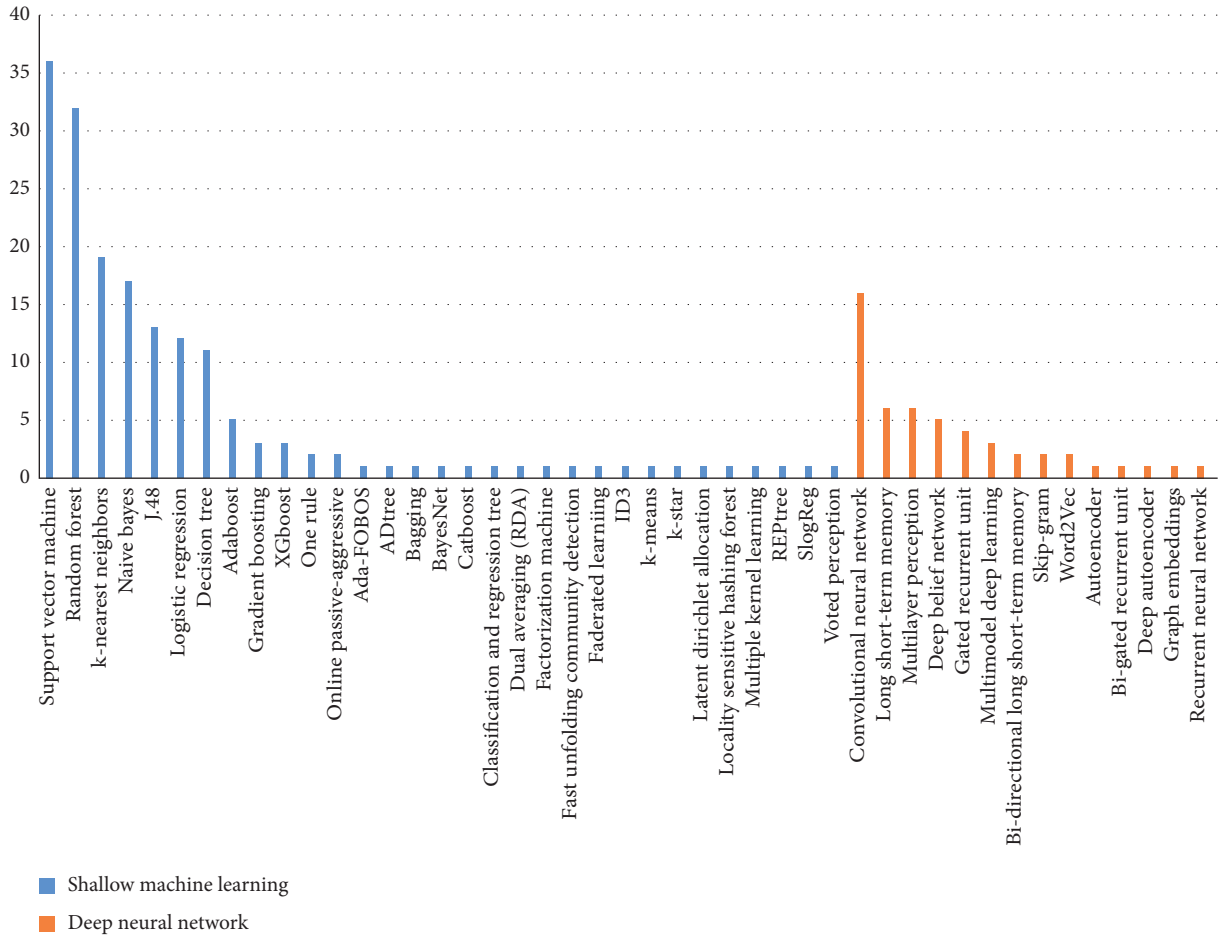

Figure 3: Statistics of the machine learning algorithm used in static Android malware detection papers from January 2019 to November 2020 in DBLP.

in some respects, but there are also some limitations, mainly as follows:

(1) Lack of standard benchmark datasets: there are 17 malware datasets used to verify the practical effect in 118 papers that detect Android malware based on static machine learning from Jan. 2019 to Nov. 2020. We can see the statistics of these datasets in Figure 4. These datasets only provide malicious samples, resulting in the dilemma that researchers must collect benign samples from several app stores. The lack of standard benchmark datasets makes it challenging to evaluate which detection method is better or worse judicially.

(2) There is no guarantee that the classifier model based on the existing dataset still has a good detection effect on new malicious applications. Many algorithms may achieve good detection results on some datasets for a while. However, as time goes, the new malicious samples may not be suitable for classification using the previous learning model, or the previously trained model leads to poor results.

(3) The ability to resist obfuscation and other targeted attacks is generally weak. The article [87] mentions antialiasing, but it is limited to the package name and the owner's name in the certificate and is invalid for obfuscation of Smali and native code. Obfuscation attacks conceal many of the original features, causing some static machine learning methods not to work very well.

5.2. Future Directions. According to the published papers of DBLP from Jan. 2016 to Nov. 2020, Android malware detection has always been a hot research direction. As shown in Figure 5, the number of papers on Android malware detection based on machine learning is roughly equivalent in recent years. The detection method using static features has always been an absolute advantage. It can be concluded that the static machine learning detection method will still be a hot spot in the foreseeable future. At the same time, new detection technologies are continually emerging, and they will be more lightweight, fast, stable, and robust.

(1) The performance of static deep learning methods will reach a higher level. Among the Android malware detection papers based on static machine learning in DBLP from Jan. 2019 to Nov. 2020, the Accuracy metric was used 88 times. As shown in Figure 6, we

Table 1: Commonly used machine learning algorithms in Android malware detection.

References	Algorithms	Feature sources	Dataset	Number of samples	Metrics	Values	Contribution	Limitation
[71]	LR	Permissions API calls	PRAGaurd Google Play	669 malware 652 benign	Accuracy	0.9725	Using logistic regression as a classifier makes training and testing get a quick response.	Feature dimensions can be further reduced by feature filtering.
[72]	LR C4.5 RF RT SVM	Permissions Bag-of-words from java file	Collected by the researchers	400 apps	Precision Recall F_1	0.958 0.957 0.956	Using the ensemble learning method, high performance can be achieved.	The dataset constructed by the authors was small, and no test was conducted on the large dataset.
[73]	NB	API calls Resources Assets Libraries Permissions	MalGenome Official and third markets	1000 malware 1000 benign	Accuracy TNR FPR TPR FNR AUC Precision	0.921 0.937 0.063 0.906 0.094 0.9722 0.935	Proposed a Bayesian classification achieved a significantly reasonable detection rate.	Accuracy will reduce when using fewer feature types, so trade-offs need to be made.
[78]	kNN k-means	Permissions Components Intent messages API calls	Contagio Google Play	238 malware 1500 benign	Accuracy Recall Precision F_1	0.9787 0.8739 0.9674 0.9183	Proposed DroidMat using kNN algorithm to classify the app as benign or malicious.	Unable to detect malicious code sent remotely as an upgrade.
[79]	kNN	Permissions Components API calls Network addresses	Drebin	5559 malware 123,452 benign	Accuracy	0.9948	Analyses different distance measures for the KNN in the context of A ndroid malware detection.	The classification performance of multi-family malware was not tested.
[75]	SVM kNN	Permissions API calls	Drebin Several public malware libraries Google Play	3986 malware 3986 benign	TPR FPR Accuracy Recall	0.975 0.032 0.975 0.975	Presented FEST with a high accuracy for malware detection.	Cannot be applied to multiple categories.
[80]	NB BN SVM J48 kNN RF	Permissions Uses-feature tag of manifest	VirusTotal 2013 Benign apps collected by the researchers	333 malware 333 benign	TPR FPR AUC Accuracy	0.94 0.05 0.98 0.9483	Developed a method that extracts several features from the manifest file to build machine learning classifiers.	Only use the manifest file to extract features, the information source is insufficient.
[76]	SVM	Permissions Classes.dex file	Contagio MalGenome Google Play	10,000 malware 30,000 benign	Accuracy TPR FPR	0.988 0.90 0.0008	Presented an efficient method for improving the updatability of the anti-virus tool.	The effect is not good for java reflection, java native code, and obfuscated malicious code.
[47]	RF SVM	n-opcode of classes.dex	Drebin Google Play	5560 malware 5560 benign	Accuracy	0.9688	Employed the probability of n consecutive opcodes as feature.	Features only consider opcode n-gram. Adding more factors may be better.
[48]	NB SVM PDT RF	n-opcode of classes.dex	MalGenome Google Play	1260 malware 1260 benign	F_1	0.98	Tested multiple classification algorithms using opcode as features.	Features only consider opcode n-gram. Adding more factors may be better.

Table 1: Continued.

References	Algorithms	Feature sources	Dataset	Number of samples	Metrics	Values	Contribution	Limitation
[74]	RF kNN BN SVM	Frequency of strings Permissions from app Permissions from markets	Collected by the researchers	No malware 820 benign	AUC	0.93	The proposed method enables fast classification of benign applications.	Did not consider malicious applications.
[77]	SVM	ICC-related components Explicit/ Implicit intents Intent filters	Drebin Google Play	5264 malware 14,264 benign	Accuracy TPR FPR	0.974 0.931 0.0067	Proposed a method leverage on ICC mechanism for malware detection.	Only considered ICC-related features.
[82]	DBN	API calls Risky permissions	Drebin Google Play	1400 malware 1400 benign	Accuracy Recall	0.90 0.9429	Proposed the Weight-Adjusted DroidDeepLearner using dangerous APIs and risky permissions as features to build a DBN model.	Comparative experiments and indicators are slightly less, failing to show performance in detail.
[81]	DBN SVM	Permissions API calls Actions Components	MalGenome Drebin Google Play	3986 malware 3986 benign	Accuracy	0.994	Proposed DroidDeep, an Android malware detection approach based on DBN and SVM.	The test dataset is small and cannot explain its large-scale performance.
[83]	CNN	Permissions Intent filters API calls Constant strings	Drebin Chinese app markets	5546 malware 5224 benign	Precision Recall Accuracy F_1	0.966 0.983 0.974 0.974	Presented DeepClassifyDroid based on CNN model.	Unable to resist attacks that confuse strings and code.
[85]	CNN	Permissions	MalGenome Debrin Apk mirror Apk4fun	2000 malware 500 benign	Accuracy	0.93	Propose a CNN-based model extracts the patterns of Android malware.	Only considered permission features.
[84]	CNN	API call sequences	Contagio Third party app stores	216 malware 1016 benign	Accuracy Precision Recall	0.994 1 0.983	Their work is one of the first attempts to construct DNNs for Android application/ malware classification.	Only considered API sequences as features.
[86]	BiLSTM LSTM CNN DBN	Opcodes	AMD Drebin VirusShare	5560 malware 123,453 benign	Accuracy F_1	0.999 0.996	Proposed BiLSTMs for Android malware classification achieved excellent results.	Only considered opcodes as features.
[87]	RNN CNN	Package name Certificate owner name Permissions Intent actions	VirusTotal	2,000,000 apps	AUC TPR FPR	0.9986 0.977 0.01	Proposed a classification method for using stacked RNNs and CNNs.	Should consider the potential effects of different feature fusion methods.
[88]	BiLSTM	API call sequences	AMD Google Play	9616 malware 11,982 benign	Accuracy F_1 FPR	0.9722 0.9821 0.0211	Their work can automatically locate the malicious code by introducing the LSTM-Attention mechanism.	Only considered API sequences as features.
[89]	LSTM-GAN	Opcodes	Drebin AMD VirusShare	2 million apps	Accuracy Precision F_1 FPR AUC	0.991 0.99 0.99 0.002 0.987	Proposed lstm-gan to cater malware detection.	Should take into account other elements in the payload of an app.

Table 1: Continued.

References	Algorithms	Feature sources	Dataset	Number of samples	Metrics	Values	Contribution	Limitation
[90]	MDL	Permissions Components Environmental information Strings Opcode frequency API frequency	VirusShare MalGenome Google Play	21,260 malware 20,000 benign	Accuracy Precision Recall F_1	0.98 0.98 0.99 0.99	The first study of multimodal deep learning methods to be used in the Android malware detection.	The generation of feature vectors depends on the malicious feature database, which requires human experts' experience.
[91]	MKL	APIs Permissions Information source-sink Dalvik instructions CFG signature	Drebin VirusShare Google Play AndroidDrawer AnZhi AppsApk FDroid SlideMe Mystique	5560 apps 24,317 apps 15,000 apps 2399 apps 3027 apps 2481 apps 1007 apps 5770 apps 3000 apps	Precision Recall F_1	0.988 0.982 0.985	Proposed MKLDroid that systematically integrates multiple views of apps for performing malware detection and malicious code localization.	Lack of dataflow analyses.
[92]	GE CNN	API call graph	AMD AndroZoo Drebin ISCX Android Botnet Dataset	33,139 malware 25,000 benign	Accuracy Recall Precision F_1	98.86 98.47 98.84 98.65	Proposed methods using pseudo-dynamic analysis of Android apps, constructing API call graph, and embedding graphs into a low dimension feature vector.	The time cost of graph embedding to generate features is relatively high.
[93]	RL Skip-gram	Interprocedural Control flow graph	Drebin VirusShare Google Play	19,944 malware 20,000 benign	Precision Recall F_1	88.07 90.41 89.22	Presented apk2vec, a semi-supervised multimodal RL technique to automatically build data-driven behaviour profiles of Android apps.	Accuracy will be affected when the number of labelled samples was too small.

LR: Logistic Regression, RF: Random Forests, RT: Random Tree, SVM: Support Vector Machine, NB: Naïve Bayes, *k*NN: *k*-Nearest Neighbours, BN: Bayesian Network, PDT: Partial Decision Tree, DBN: Deep Belief Networks, CNN: Convolutional Neural Networks, LSTM: Long-Short Term Memory, BiLSTM: Bi-directional LSTM, GAN: Generative Adversarial Networks, MDL: Multimodal Deep Learning, MKL: Multiple Kernel Learning, GE: Graph Embedding, and RL: Representation Learning.

can see that most Accuracy metrics are over 90%. It can be predicted that the detection methods will continue to improve efficiency and speed, while maintaining this crucial Accuracy metric.

(2) The detection method will be more inclined to support large-scale detection. Many algorithms in past research works were evaluated over small datasets. Although excellent performance metrics were achieved, these algorithms' scalability was not verified on large datasets. With the increasing number of applications in the future, there will be a growing need for fast detection methods to support massive apps.

(3) The detection model needs to have the ability to identify zero-day attacks and new malware. Static detection based on machine learning has excellent classification ability for known malicious applications. However, it is easy to misreport new and unknown 0-day samples because the new virus weakens the known features. Future detection technology will be developed towards improving 0-day detection ability.

(4) The antiattack capabilities of the machine learning model used in Android malware detection will be further enhanced. Many machine learning algorithms are vulnerable to poisoning attacks, spoofing

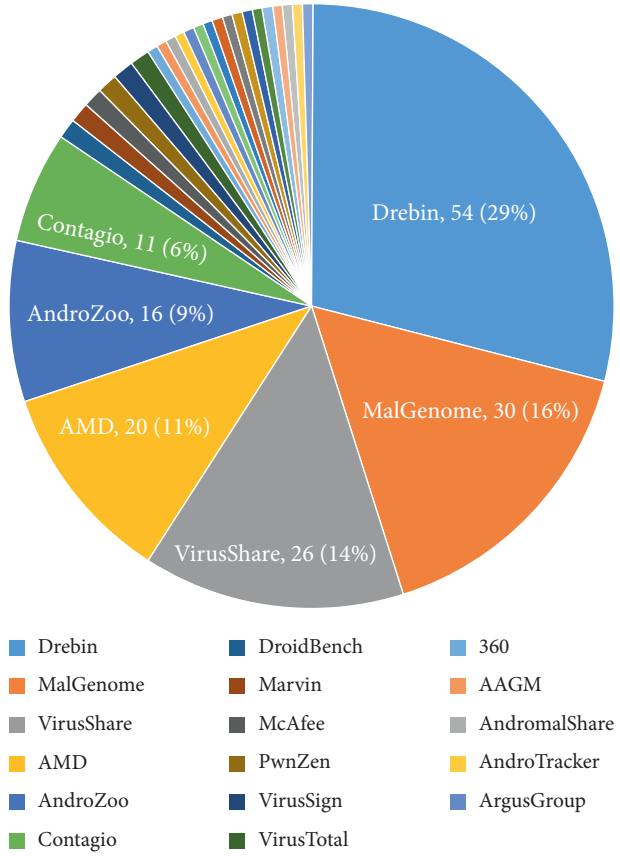

Figure 4: Statistics of malware datasets used in static Android malware detection papers from Jan. 2019 to Nov. 2020 in DBLP.

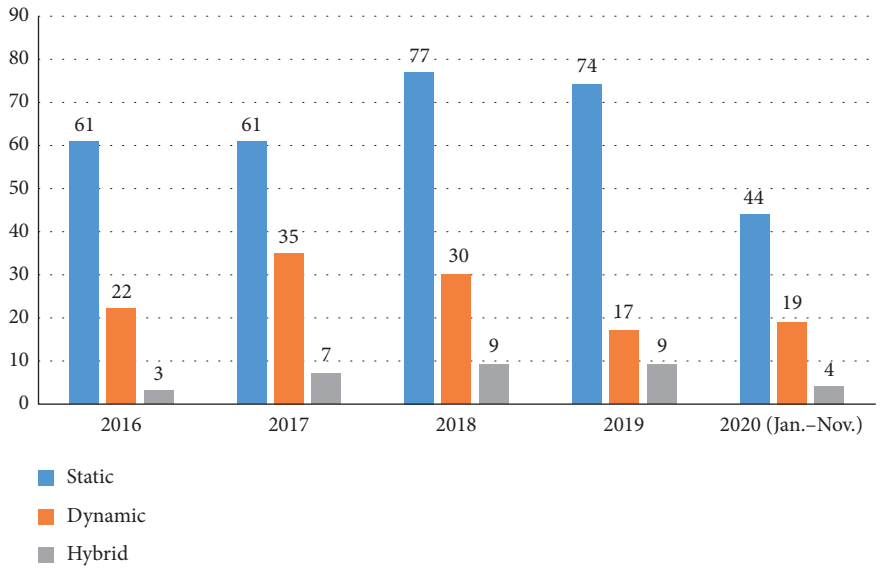

Figure 5: Statistics of machine learning-based Android malware detection papers from 2016 to Nov. 2020 in DBLP.

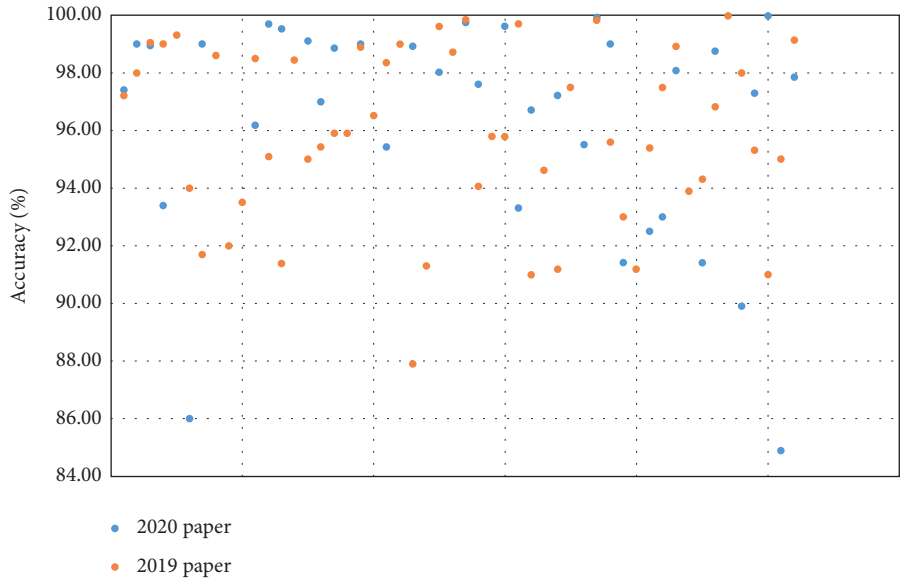

Figure 6: Statistics of the Accuracy metric of machine learning-based Android malware detection papers from Jan. 2019 to Nov. 2020 in DBLP.

attacks, impersonate attacks, and inversion attacks [94]. In the literature surveyed, no effective protection measures are proposed for possible attacks, which will be improved in the future.

6. Conclusion

With the continuous growth of Android devices and applications, Android apps' security has attracted more and more attention. This paper studied Android app composition, analysed the source of static features, reviewed Android malware static detection technology based on machine learning, and discussed the future development direction. We analysed the algorithm model, core ideas, datasets, and performance metrics of the existing methods through the vertical comparison method and pointed out the advantages and limitations. Compared with other types of Android malicious application detection technology, the static detection method based on machine learning has advantages in the comprehensiveness, accuracy, and less expert dependence of detection, although it also has some weaknesses. This paper's work may provide Android application security researchers with reference, help them quickly grasp various methods, master key issues, and understand the development trend of technology.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant no. 61572513.

References

[1] AppBrain, "Number of android applications," 2020, https://www.appbrain.com/stats.

[2] AV-TEST Inst., "Malware statistics & trends report," 2020, https://www.av-test.org/en/statistics/malware/.

[3] Mcafee, "McAfee mobile threat report Q1," 2020, https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf.

[4] Y. Xu, G. Wang, J. Ren, and Y. Zhang, "An adaptive and configurable protection framework against android privilege escalation threats," *Future Generation Computer Systems*, vol. 92, pp. 210–224, 2019.

[5] Y. He, X. Yang, B. Hu, and W. Wang, "Dynamic privacy leakage analysis of Android third-party libraries," *Journal of Information Security and Applications*, vol. 46, pp. 259–270, 2019.

[6] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection," *Digital Investigation*, vol. 13, pp. 22–37, 2015.

[7] S. K. Sahay and A. Sharma., "A survey on the detection of android malicious apps," *Advances in Computer Communication and Computational Sciences*, pp. 437–446, Springer, Singapore, 2019.

[8] R. Riasat, "A survey on android malware detection techniques," *DEStech Transactions on Computer Science and Engineering Wcne*, 2016.

[9] N. Yadav, A. Sharma, and D. Amit, "A survey on Android malware detection," *International Journal of New Technology and Research*, vol. 2, p. 12, 2016.

[10] A. Naway and Y. Li, "A review on the use of deep learning in android malware detection," 2018, http://arxiv.org/abs/1812.10360.

[11] J. Qiu, "A survey of Android malware detection with deep neural models," *ACM Computing Surveys*, vol. 53, pp. 1–36, 2020.

[12] Z. Wang, Q. Liu, and Y. Chi, "Review of android malware detection based on deep learning," *IEEE Access*, vol. 8, pp. 181102–181126, 2020.

[13] A. Arulmurugan, B. Poonguzhali, M. Sugammathi, and M. Madhumathi, "A survey on the novel approach to detect malware variants by user oriented behavior based system for android," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 3, no. 1, pp. 1319–1324, 2018.

[14] K. Kulkarni and A. Y. Javaid, "Open source android vulnerability detection tools: a survey," 2018, http://arxiv.org/abs/1807.11840.

[15] Y. Pan, X. Ge, C. Fang, and Y. Fan, "A systematic literature review of android malware detection using static analysis," *IEEE Access*, vol. 8, pp. 116363–116379, 2020.

[16] K. Dema and T. Jamtsho, "A systematic review on android malware detection," *Asian Journal For Convergence In Technology*, vol. 5, no. 3, pp. 83–86, 2019.

[17] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020.

[18] M. Odusami, "Android malware detection: a survey," *International Conference on Applied Informatics*, Springer, Cham, Switzerland, 2018.

[19] M. A. Ashawa and S. Morris, "Analysis of android malware 2 detection techniques: a systematic review," *International Journal of Cyber-Security and Digital Forensics*, vol. 8, no. 3, pp. 117–188, 2019.

[20] E. C. Bayazit, O. K. Sahingoz, and B. Dogan, "Malware detection in Android systems with traditional machine learning models: a survey," in *Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, IEEE, Ankara, Turkey, June 2020.

[21] V. Kouliaridis, K. Barmpatsalou, G. Kambourakis, and S. Chen, "A survey on mobile malware detection techniques," *IEICE Transactions on Information and Systems*, vol. E103.D, no. 2, pp. 204–211, 2020.

[22] P. Bhat and K. Dutta, "A survey on various threats and current state of security in android platform," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–35, 2019.

[23] Google Inc., "Application fundamentals," 2020, https://developer.android.com/guide/components/fundamentals.

[24] Google Inc., "Analyze your build with APK Analyzer," 2020, https://developer.android.com/studio/build/apk-analyzer.

[25] Google Inc.: Manifest Permission, 2020, https://developer.android.com/reference/android/Manifest.permission.html.

[26] X. Fu and H. Cai, "On the deterioration of learning-based malware detectors for Android," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, Montreal, Canada, May 2019.

[27] H. Cai and J. Jenkins, "Towards sustainable android malware detection," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, Gothenburg, Sweden, May 2018.

[28] X. Zhang, "Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, USA, November 2020.

[29] K. Xu, "Droidevolver: self-evolving android malware detection system," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, Stockholm, Sweden, June 2019.

[30] D. Arp, "Drebin: effective and explainable detection of android malware in your pocket," *NDSS*, vol. 14, 2014.

[31] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, IEEE, San Francisco, CA, USA, May 2012.

[32] Y. Li, "Android malware clustering through malicious payload mining," *International Symposium on Research in Attacks, Intrusions, and Defenses*, Springer, Cham, Switzerland, 2017.

[33] A. F. A. Kadir, N. Stakhanova, and A. A. Ghorbani, "Android botnets: what urls are telling us," *International Conference on Network and System Security*, Springer, Cham, Switzerland, 2015.

[34] G. Meng, "Mystique: evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, Xi'an, China, June 2016.

[35] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: an extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[36] Blogspot, "Contagio mobile malware," 2021, http://contagiodump.blogspot.com/.

[37] K. Allix, "Androzoo: collecting millions of android apps for the research community," in *Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, IEEE, Austin, TX, USA, May 2016.

[38] VirusShare, "Because sharing is caring," 2021, https://virusshare.com/.

[39] VirusTotal: VirusTotal, 2021, https://www.virustotal.com/.

[40] S. Cimato, A. De Santis, and U. Ferraro Petrillo, "Overcoming the obfuscation of Java programs by identifier renaming," *Journal of Systems and Software*, vol. 78, no. 1, pp. 60–72, 2005.

[41] M. Ficco, "Comparing API call sequence algorithms for malware detection," in *Proceedings of the Workshops of the International Conference on Advanced Information Networking and Applications*, Cham, Switzerland, 2020.

[42] A. Kovacheva, "Efficient code obfuscation for Android," in *Proceedings of the International Conference on Advances in Information Technology*, Cham, Switzerland, 2013.

[43] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, Raleigh, NC, USA, October 2012.

[44] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*, Berlin, Heidelberg, 2011.

[45] G. Suarez-Tangil, "Droidsieve: fast and accurate classification of obfuscated android malware," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, Scottsdale, AZ, USA, March 2017.

[46] J. Garcia, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," Department of Computer Science, George Mason University, Tech. Rep 202, 2015.

[47] G. Canfora, "Effectiveness of opcode ngrams for detection of multi family android malware," in *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security*, IEEE, Toulouse, France, August 2015.

[48] B. J. Kang, "N-opcode analysis for android malware classification and categorization," in *Proceedings of the 2016 International Conference on Cyber Security and Protection of*

Digital Services (Cyber Security), IEEE, London, UK, June, 2016.
[49] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[50] S. Poeplau, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," *NDSS*, vol. 14, 2014.
[51] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 1–29, 2018.
[52] S. Millar, "DANdroid: a multi-view discriminative adversarial network for obfuscated Android malware detection," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, Orleans, LA, USA, March 2020.
[53] Hosmer Jr, W. David, L. Stanley et al., *Applied Logistic Regression*, John Wiley & Sons, Hoboken, NJ, USA, 2013.
[54] I. Rish, "An empirical study of the naive Bayes classifier," in *Proceedings of the IJCAI 2001 workshop on empirical methods in artificial intelligence*, Washington, DC, USA, August, 2001.
[55] F. V. Jensen, *An Introduction to Bayesian Networks*, UCL Press, London, UK, 1996.
[56] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.
[57] L. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.
[58] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.
[59] L. Breiman, "Random forests," UC Berkeley TR567, 1999.
[60] G. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.
[61] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.
[62] H. Salehinejad, "Recent advances in recurrent neural networks," 2017, http://arxiv.org/abs/1801.01078.
[63] I. Goodfellow, J. Pouget-Abadie, and M. Mirza, "Generative adversarial nets," *Advances in Neural Information Processing Systems*, http://arxiv.org/abs/1406.2661, 2017.
[64] T. Baltrušaitis, C. Ahuja, and L.-P. Morency, "Multimodal machine learning: a survey and taxonomy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 2, pp. 423–443, 2018.
[65] M. Gönen and E. Alpaydın, "Multiple kernel learning algorithms," *Journal of Machine Learning Research*, pp. 2211–2268, 2011.
[66] S. Yan, "Graph embedding and extensions: a general framework for dimensionality reduction," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 1, pp. 40–51, 2006.
[67] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: a review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.
[68] F. Pendlebury, "{TESSERACT}: eliminating experimental bias in malware classification across space and time," in *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19)*, Washington, DC, USA, August 2019.
[69] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, pp. 1–28, 2020.
[70] R. Jordaney, "Transcend: detecting concept drift in malware classification models," in *Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17)*, Vancouver, BC, Canada, August 2017.
[71] S. R. Tiwari and R. U. Shukla, "An android malware detection technique based on optimized permissions and API," in *Proceedings of the 2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, IEEE, Coimbatore, India, July 2018.
[72] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided Android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266–274, 2017.
[73] S. Y. Yerima, "A new android malware detection approach using bayesian classification," in *Proceedings of the 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, IEEE, Barcelona, Spain, March 2013.
[74] B. Sanz, "On the automatic categorisation of android applications," in *Proceedings of the 2012 IEEE Consumer Communications And Networking Conference (CCNC)*, IEEE, Las Vegas, NV, USA, January 2012.
[75] K. Zhao, "Fest: a feature extraction and selection tool for Android malware detection," in *Proceedings of the 2015 IEEE Symposium on Computers and Communication (ISCC)*, IEEE, Larnaca, Cyprus, July, 2015.
[76] N. Nissim, R. Moskovitch, O. BarAd, L. Rokach, and Y. Elovici, "ALDROID: efficient update of Android anti-virus software using designated active learning methods," *Knowledge and Information Systems*, vol. 49, no. 3, pp. 795–833, 2016.
[77] K. Xu, Y. Li, R. H. Deng, and Deng, "Iccdetector: icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.
[78] D.-J. Wu, "Droidmat: android malware detection through manifest and api calls tracing," in *Proceedings of the 2012 Seventh Asia Joint Conference on Information Security*, IEEE, Tokyo, Japan, August 2012.
[79] G. Baldini and D. Geneiatakis, "A performance evaluation on distance measures in KNN for mobile malware detection," in *Proceedings of the 2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, IEEE, Thessaloniki, Greece, April 2019.
[80] B. Sanz, "MAMA: manifest analysis for malware detection in android," *Cybernetics and Systems*, vol. 44, pp. 6-7, 2013.
[81] X. Su, "A deep learning approach to android malware feature learning and detection," in *Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA. IEEE*, Tianjin, China, October 2016.
[82] W. Li, "An android malware detection approach using weight-adjusted deep learning," in *Proceedings of the 2018 International Conference on Computing, Networking and Communications (ICNC)*, IEEE, Maui, HI, USA, March 2018.
[83] Yi Zhang, Y. Yang, and X. Wang, "A novel android malware detection approach based on convolutional neural network," in *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, Guiyang, China, March 2018.
[84] R. Nix and J. Zhang, "Classification of android apps and malware using deep neural networks," in *Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Anchorage, AL, USA, May 2017.
[85] M. Ganesh, "CNN-based android malware detection," in *Proceedings of the 2017 International Conference on Software*

Security and Assurance (ICSSA), IEEE, Altoona, PA, USA, July 2017.

[86] M. Amin, T. A. Tanveer, M. Tehseen, M. Khan, F. A. Khan, and S. Anwar, "Static malware detection and attribution in android byte-code through an end-to-end deep system," *Future Generation Computer Systems*, vol. 102, pp. 112–126, 2020.

[87] W. Y. Lee, J. Saxe, and R. Harang, "SeqDroid: obfuscated Android malware detection using stacked convolutional and recurrent neural networks," *Deep Learning Applications for Cyber Security*, pp. 197–210, Springer, Cham, Switzerland, 2019.

[88] Z. Ma, "Droidetec: android malware detection and malicious code localization through deep learning," 2020, http://arxiv.org/abs/2002.03594.

[89] M. Amin, B. Shah, A. Sharif et al., "Android malware detection through generative adversarial networks," *Transactions on Emerging Telecommunications Technologies*, p. e3675, 2019.

[90] T. G. Kim, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[91] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to Android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1222–1274, 2018.

[92] A. Pektaş and T. Acarman, "Deep learning for effective Android malware detection using API call graph embeddings," *Soft Computing*, vol. 24, no. 2, pp. 1027–1043, 2020.

[93] A. Narayanan, "APK2VEC: semi-supervised multi-view representation learning for profiling Android applications," in *Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM)*, IEEE, Beijing, China, November 2018.

[94] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu, and V. C. M. Leung, "A survey on security threats and defensive techniques of machine learning: a data driven view," *IEEE Access*, vol. 6, pp. 12103–12117, 2018.