IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 2, 2021, accepted October 19, 2021, date of publication October 26, 2021, date of current version November 3, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3123187

Recent Advances in Android Mobile Malware Detection: A Systematic Literature Review

ABDULAZIZ ALZUBAIDI
Computer Science Department, College Computing in Al-Qunfudhah, Umm Al-Qura University, Alawali, Mecca 24381, Saudi Arabia
e-mail: aazubaidi@uqu.edu.sa

This work was supported by the Deanship of Scientific Research at Umm Al-Qura University under Grant 18-COM-1-01-0007.

ABSTRACT In recent years, the global pervasiveness of smartphones has prompted the development of millions of free and commercially available applications. These applications allow users to perform various activities, such as communicating, gaming, and completing financial and educational tasks. These commonly used devices often store sensitive private information and, consequently, have been increasingly targeted by harmful malicious software. This paper focuses on the concepts and risks associated with malware, and reviews current approaches and mechanisms used to detect malware with respect to their methodology, associated datasets, and evaluation metrics.

INDEX TERMS Smartphone, intrusion detection, mobile malware, android devices, machine learning.

I. INTRODUCTION

In the last decade, the use of smartphones has accelerated globally. Currently, smartphones support a wide range of tasks, such as taking photographs, recording videos, texting, and performing financial transactions. Smartphones can also support gaming, networking, and educational tasks. Globally, in terms of units shipped and usage, smartphones outstrip both desktop and tablet computers [1], [2].

Smartphones are equipped with one of the two dominant operating systems (OS), i.e., Android and iPhone OS (iOS). Android has maintained its position as the leading mobile OS, with a 72.3% market share compared to iOS, with 27% [3], [4]. A mobile application (app) is a program for mobile devices developed to perform a specific task [5]. As of 2020, there were 2,570,000 apps available on the Google Play Store, 1,840,000 on the Apple App Store, 669 000 on the Windows Store, and 489,000 on the Amazon Appstore [6]. In 2019, 84.5 billion Android apps and 30.6 billion iOS apps were downloaded from these app stores [7].

Smartphones provide a variety of services; however, some store large amounts of valuable information, which introduces significant threats to security and privacy. For example, mobile malware can infect smartphone devices in order to steal classified information, share and track activities, and perform various tasks, such as making unauthorized phone calls [8]. Statistics [9] showed that in 2019 there were more than 3.5 million malware installation packages, including worms, Trojans, and adware. Thus, defending against such malware is an important undertaking, and methods and techniques to detect and prevent malware infections of mobile devices have attracted increasing attention in both academic and industrial fields.

The associate editor coordinating the review of this manuscript and approving it for publication was Tyson Brooks.

A. PROBLEM

The number of mobile malware packages and malware attacks on mobile devices is increasing. According to Victor Chebyshev [10], a security expert at Kaspersky Lab, the number of stalkerware attacks on the personal data of mobile device users increased to 67,500 in 2019, almost double the number of attacks the year before, i.e., 40,386 attacks on unique users in 2018. Stalkerware [11] is spyware software that intercepts personal information, such as photos, videos, and global positioning system (GPS) coordinates, and appears as a parent application. The number of users targeted by adware attacks in 2019 was slightly more than 200,000 and the number of adware installation packages detected in 2018 and 2019 was 440098 and 764265, respectively. Mobile malware can also target financial transactions, and 70,000 mobile banking trojans were downloaded in 2019. Also in 2019, the number of pieces of mobile ransomware malware detected increased by 8186 compared to 2018.

B. MOTIVATION

Intrusion detection approaches for smartphones have received increasing amounts of attention over the past few years. Therefore, there are several published surveys that have discussed detection of Android malware. For example,

Kouliaridis and Kambourakis [12] discussed 19 papers published between 2014 and 2021, in terms of chosen metrics, dataset ages, classification models, and performance improvement techniques. Selvaganapathy *et al.* [13] discussed the framework of Android malware detection and provided a short list of about fourteen public datasets (name and description), then briefly discussed ten feature analysis approaches published between 2014 and 2019 as well as common machine learning tools used to construct them in the Android framework. Liu *et al.* [14] comprehensively discussed ninety published papers from 2010 to 2020, with respect to the collected data, feature types, algorithms used, and evaluation results. Qui *et al.* [15] covered more than thirty published papers, concentrating on deep-learning algorithms, while the authors [16] provided a comprehensive review of twenty two studies published between 2009 and 2019 with respect to their detection methods and evaluation results. Faruki *et al.* [17] covered fifteen Android malware detection frameworks with respect to their goals, methods, deployment and availability for the period between 2009 and 2013.

- Detection of mobile malware has gained more attention recently; therefore, several surveys have been published such as Enck [18], La Polla *et al.* [19], Faruki *et al.* [17], Feizollah *et al.* [20], and Faruki *et al.* [17] focused on papers published between 2006 and 2015, while this review covers papers published between 2010 and 2021.
- Various surveys have concentrated on certain types of feature analysis [21], [22]; however, we have covered all three types: static, dynamic and hybrid.
- Although [12], and [13] discussed recently published papers in the field, this survey provides suitable background information for any researcher interested in investigating intrusion detection in smartphone devices.

In light of the above, this paper aims to provide a comprehensive survey of recent studies published since 2010, covering essential feature extraction analysis: static, dynamic, and hybrid methods. Then, providing details for the public datasets and covering various approaches, not only those using machine learning algorithms, but also applied deep-learning algorithms, considering adversarial attacks and other techniques. Table 1 compares our survey with most current surveys in this field.

C. CONTRIBUTIONS

The primary contributions of this paper are as follows:

- Providing a comprehensive survey of papers published between 2010 and 2021 that focus on intrusion detection approaches implemented on Android devices with respect to the number of samples, extracted features, and performance.
- Classifying malware detection based on extracted features.
- Highlighting malware in terms of its background, patterns, and families.

The remainder of this paper is organized as follows: Section II summarizes the background related to the topic in terms of app patterns, feature extraction techniques, definitions and goals of malware, evaluation metrics, and publicly available datasets. The types of feature extraction used to detect mobile malware, data collection, classifiers, and results are discussed in Section IV. Open problems, lessons learned, and future trends are presented in Section V, and the conclusions are presented in Section VI.

II. BACKGROUND

This section provides general characteristics related to Android malware detection.

A. APPLICATION BEHAVIOR

Understanding the pattern of mobile application is the main key for mobile malware detection. Hence, for a given app (b), and set of all apps (B), a malware classifier implements a function F, such that for each $b \epsilon B$, $F(b) =$ 'abnormal' if b is malicious, otherwise $F(b) =$ 'normal'.

B. MOBILE MALWARE: DEFINITION, GOALS, AND BEHAVIORS

Mobile malware is a malicious software app that targets the operating system, apps, and classified information stored on mobile phones. Therefore, the malware attackers usually upload their apps into official markets as well as third parties or use social engineering strategies to gain unauthorized access and exploit root privileges without user consent [8]. These apps have different patterns to achieve attacking goals as listed in Table 2.

C. MOBILE MALWARE FAMILIES

A mobile malware family is defined as a set of apps that share similar patterns, types of attack, and degree of harm [37]. In addition, each mobile malware family contains sub-viruses that perform specific purposes. For example, DroidKungFu, a commonly used piece of mobile malware, contains several viruses, each of which performs a specific task. For instance, DroidKungFu A (2011) obtained illegal privileges [38], DroidKungFu B (2012) utilized additional command and control (C&C) domains and employed native code to make difficult to detect [39], and DroidKungFu C (2011) was used to gain unauthorized privileges and exploit additional encrypted C&C [40]. The common Android virus families discovered in 2019 are listed in Table 3.

D. FEATURE APPROACHES

Most current studies analyze the features of suspected apps, and rely on static, dynamic, and hybrid methods. These concepts are defined in this section.

1) STATIC APPROACH

The static approach analyzes the source code to determine the functionality of the malware. It is important to note that executing the code is not required. Various features, listed below, are available in the Android application package kit

TABLE 1. Comparison with other surveys, where ✓ = topic covered, X = topic not covered.

Study	Covered Years	Static	Dynamic	Hybrid	Dataset
Enck [18], 2011	Between 2007 and 2011	✓	✓	x	x
La Polla et al. [19], 2012	Between 2004 and 2010	✓	✓	x	x
Amamra at al. [23], 2012	Between 2007 and 2011	x	x	x	x
Eshmawi and Nair [22], 2013	Between 2008 and 2013	x	x	x	x
Farukiet al. [17], 2014	Between 2009 and 2014	✓	✓	✓	x
Feizollah et al. [20], 2015	Between 2010 and 2014	✓	✓	✓	x
Arshad et al. [24], 2016	Between 2009 and 2013	✓	✓	x	x
Zachariah et al. [25], 2017	Between 2009 and 2017	✓	✓	x	x
Odusami et al. [26], 2018	Between 2010 and 2018	✓	✓	x	x
Gyamfi and Owusu [27], 2018	Between 2010 and 2016	✓	✓	✓	x
Yan P and Yan Z [21], 2018	Between 2011 and 2017	x	✓	x	x
Souri and Hosseini [28], 2018	Between 2011 and 2017	✓	✓	x	x
Agrawal and Trivedi [29], 2019	Between 2011 and 2017	✓	x	x	x
Qui et al. [15], 2020	Between 2014 and 2019	✓	✓	✓	x
Kouliaridis et al. [16],2020	Between 2009 and 2019	✓	✓	✓	x
Liu et al. [14], 2020	Between 2010 and 2020	✓	✓	✓	✓
Kouliaridis et al. [12], 2021	Between 2014 and 2021	✓	✓	✓	✓
Selvaganapathy et al. [13], 2021	Between 2014 and 2019	✓	✓	x	x
Our Survey, 2021	Between 2010 and 2021	✓	✓	✓	✓

(APK), and can be examined to differentiate normal from malware behavior.

1) Application and User Behaviors: are based on extracted syntactic and semantic features, and analyze a suspicious app's behavior in order to distinguish normal and malware apps.
2) Resource Consumption. This relies on how resources, such as battery, central processing unitr (CPU), and memory, are consumed.
3) Network Addresses: monitors activities with regard to network events, such as Domain Name Service (DNS) packets, C&C servers looking to the Internet Protocol (IP) address, and signal processing.
4) Requested Permissions: relies on granting resources once the smartphone user has installed an app using three types of permission: normal permissions (launching data and resources outside the app sandbox, e.g., time zone), signature permissions (using a permission that is utilized with the same certificate); and dangerous permissions, which seek to access classified information, i.e., contact lists.
5) Application Code Analysis: Android apps are written in Java language and compiled into the Dalvik format. Therefore, researchers observe API calls that interact with particular devices, for example, by retrieving the phone ID based on the telephony manager. In addition, some studies concentrate on the flow of Java code by editing the names of API calls or the sequence of API calls and utilizing classes, methods, and app instructions to differentiate normal and abnormal apps.
6) System calls: is defined as the way in which interactions occur between apps and devices. Static approaches involve unpacking, disassembling, and analyzing malware code.

The static approach has sub-approaches: signature-based, permission-based, and Dalvik bytecode detection, which are outlined below.

a: SIGNATURE-BASED DETECTION

This method is utilized by many antivirus (AV) apps and constructs a database manually or automatically. The database contains predefined, specified, and classified malware in order to detect suspicious apps. An app that is found to contain potential threats is compared to the malware in the database by using different techniques, such as a byte signature, byte sequences in a file, or data stream (e.g., a hexadecimal signature or a hashing signature initiated by a hash function), and the data are transformed into a series of alphanumeric features, such as the Message-Digest Algorithm 5 (MD5) [41] or the Secure Hash Algorithm 1 [42].

b: PERMISSION BASED DETECTION

This involves resources, such as cameras, contacts, and location and messaging services, being launched with or without user consent. All permissions are defined in the AndroidManifest.xml file. Current studies compare normal patterns as a reference with malicious patterns to determine whether the permissions granted are essential for the app to operate.

2) DYNAMIC APPROACH

This approach relies on an analysis of the behavior of an app in terms of its dynamic features. For example, dynamic hardware features include the battery, memory, CPU, input/output (I/O), sensor, camera, and screen usage. The way

TABLE 2. Behavior of mobile malware.

Type	Behavior	Specimen
Adware	Attached to unknown software, it relies on advertisements such as pop-ups to start showing commercial advertisements without the owner's consent.	MobiDash [30]
Worm	Takes advantage of the limitations of a vulnerable app and system for distribution and activation by themselves, which transmitted through text messages.	Cabir [31]
Spyware	Eavesdrops on activities occurring among infected devices to gather, utilize, and distribute personal information without the knowledge of the owner.	InfoStealer [32]
Botnet	Obtains full access to a device and its contents once installed; communicates with and receives instruction through command and control (C&C) servers.	Zeus [33]
Mobile Trojan	Found in a normal app and, once installed, can steal sensitive information and send messages without the owner's permission.	Hummer [34]
Backdoor	Allows attackers to go in and out of a system without being recognized, bypassing security restrictions to gain unauthorized access to personal information.	PhantomLance [35]
Ransomware	Hides information from the device user, locks the device, then demands payment so that the owners can retrieve their own information and unlock the device.	Cryptolocker [36]

in which software and firmware features affect the behavior of an app can also be analyzed. Here, software features include app patterns, permissions, privileges, network traffic, data access, information flows, and firmware features include the operating system (OS), system calls, and various predefined functions. In several analysis scenarios, dynamic analysis is performed in an isolated environment, e.g., a sandbox. Apps are observed under such conditions to examine whether their behavior is consistent with a normal pattern. Anomaly detection is a common method used for dynamic analysis. Anomaly detection compares the current pattern observed when the code is executed with the expected pattern, employs a sandbox to monitor features, builds regular behavior during the training phase, and then reports possible threats based on any deviation from the constructed model. Other approaches that can be utilized in dynamic analysis involve monitoring classified information or leakage from third parties and emulation detection, which identifies privilege escalation inside the kernel utilizing virtual machine introspection and can be used to examine patterns based on recent activities that occur outside of the emulator.

TABLE 3. Ten most common viruses discovered in 2019.

Name of family	Behavior	Purposes
FakeInst	Trojan	Sending premium rate SMS messages
OpFake	Trojan, Adware	Sending premium rate SMS messages and deceiving the target in terms of his/her browser being out of date through the use of pop-up messages.
SNDApps	Trojan	Sending private information to the server without the user knowing.
Boxer	Trojan	Sending premium rate SMS messages.
GinMaster	Trojan	Storing the victim's private information, such as mobile ID, mobile number, and other important data.
VDLoader	Trojan	Flooding devices in terms of messages and sending private information to a remote server.
FakeDolphin	worm	Deceiving the victim by mimicking the Dolphin browser and then signing up subjects without their consent and redirecting them when they browse to websites where the FakeDolphin is downloaded.
KungFu	Backdoor	Gaining unauthorized access to the victim's devices, downloading a malicious app package, and sending the stored information from the memory of the device to the server.
Basebridge	Trojan	Sending premium rate SMS messages and blocking data consumption monitoring.
JIFake	Trojan	Sending premium rate SMS messages, gathering personal information, and tracking location data.

3) HYBRID APPROACHES

These techniques rely on leveraging the benefits of both static and dynamic approaches. Hybrid approaches begin by examining the code, permissions, and components of the app (static), and then analyzing the patterns of the app (dynamic). Table 4 presents a comparison of the static, dynamic, and hybrid approaches.

Figure. 1 represents the taxonomy of recent methods used to detect mobile malware.

III. ANDROID MALWARE DETECTION APPROACHES BASED ON MACHINE LEARNING

The typical Android malware detection framework uses machine learning comprised of four main phases: collected data, extracted features, selected features, and finally applied machine learning, to classify the instances into normal or malware apps. These phases are described in this section.

A. COLLECTED DATA

It is important to obtain data to construct malware detection scheme. There are two types of datasets: private (collected by the authors for their works and not available to the public) and public (gathered for academic and industrial purposes and available for the public or some require permission to obtain the samples). Each type of dataset has one or two categorizes: normal apps that can be downloaded from authorized stores such as Google Play store while malware apps that were developed by attackers and might be found in third parties. Table 5 summarizes eleven public datasets in terms of period of collection, sample size, brief description, availability and resource.

TABLE 4. Comparison among feature approaches.

Approach	Features							
	Detecting known attack	Detecting new attack	False alarm	Comprehensive information	Extracted features	Identification	Storage complexity	Time complexity
Static	✓	x	Low	Quite	Easy	Fast	Low	Low
Dynamic	✓	✓	High	More than static	Difficult	Slow	High	High
Hybrid	✓	✓	High	More than static and dynamic	Difficult	Slow	High	High

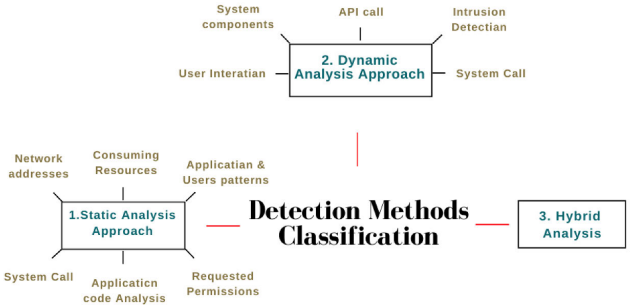

FIGURE 1. Taxonomy of recent methods used to detect mobile malware.

B. PREPROCESSING DATA

The authors should consider the preprocessing phase to reduce possible errors that could have occurred during data collection processes. This phase includes data cleaning, normalization, discretization, factorization, missing value processing, and dataset segmentation [54], [55].

C. EXTRACTED AND SELECTED FEATURES

This phase consists of two sub-phases: extracted and selected phases.

a: EXTRACTED FEATURES

Any proposed intrusion detection system should extract and analyze different types of features to examine the pattern from the Android Application Package (apk). It comprises of an input phase (represented by raw apks) and output (information extracted from Android-Manifest.xml, and Classes.dex files such as APIs, permissions, and intents. The goal of this phase to make the extracted features in an interpretable format for further analysis and extraction of features.

b: SELECTED FEATURES

Once the features are extracted, and analyzed, the next step is to determine which features are more valuable and can improve the obtained results by ranking them. Various techniques can be applied to rank features such as mutual information [56], chi-square test [57], correlation analysis [58], and posterior probability [59].

D. APPLIED MACHINE LEARNING ALGORITHM AND MODEL EVALUATION

The fourth phase aims to select a suitable machine learning model to construct the model and perform the classification of samples as normal and malware. This phase relies on splitting data into training (used to fit the parameters and train the model), validation (used to predict the responses for the observations and provide an unbiased evaluation of a model fit on the training dataset while tuning the model's hyper parameters) and testing sets (provides an unbiased evaluation of the final model fit on the training dataset). Several metrics such as accuracy, precision, recall, and F1-score are used to measure the performance of the implemented schemes [60].

IV. MALWARE DETECTION APPROACHES

In this section, we analyze recent mobile malware detection approaches, which can be broadly categorized into static, dynamic, and hybrid analyses.

A. STATIC ANALYSIS APPROACH

Static analysis methods rely on employing several features such as application and user behaviors, resource consumption, network addresses, requested permissions, application code analysis, and system calls. This section summarizes recently published papers that have used one or more static features.

Shamili *et al.* [61] utilized a statistical mechanism that relies on a support vector machine (SVM) [62] by monitoring user patterns on a mobile device. The researchers used the MIT Reality dataset [63] in terms of the activities of 75 subjects over 25 days, and extracted 20 features, such as duration of calls, time interval between calls, incoming/outgoing SMS, sending packets and activities. They implemented an intrusion that could send an SMS message every 20 minutes to construct a dataset that includes half the infected instances, and created a simulated network comprising 25 smartphones for training and 50 subjects. The average obtained accuracy was between 83% and 87%.

Burguera *et al.* [64] proposed a system called Crowdroid, which aimed to investigate Android apps in terms of their patterns. The Crowdroid system relies on collected data (installed and observed app behaviors) and utilizes a tracing tool to record log files. Then, these data are stored in a centralized database, the information is parsed to facilitate information extraction, and the feature vectors are generated. Finally, the analyzing and clustering processes are performed using k-means [65] to create a normality model for each app that can be used to detect malware. To evaluate the Crowdroid, the authors performed two experiments. In the

TABLE 5. Common public datasets.

Dataset	Period of collection	# of samples	Brief Description	Availability	Resource
Malgenome [43]	2010 - 2011	1260	The processing of collected malware samples relies on installation methods, activation mechanisms and malicious payloads with 49 families	Request	http://www.malgenomeproject.org/policy.html/
Drebin [44]	2010 - 2012	129013	The processing of collected samples relies on extracted HW components, requested permissions, app components, filtered intent, API calls, and network addresses (123453 normal apps, 5560 malware apps with 179 families)	Request	https://www.sec.cs.tu-bs.de/~danarp/drebin/
Contagio Mobile Mini-Dump [45]	2008 - 2020	370	It is a blog, which allowed researchers to upload and download malware samples	Yes	http://contagiodump.blogspot.com/?m=1
PRAGuard [46]	2013	10479	The auhtors gained benifit from [43], [45] datasets, using seven obfuscation techniques for evaluation 10479 malware with 50 families	Not available (maintenance reasons)	http://pralab.diee.unica.it/en/AndroidPRAGuardDataset
AMD [47]	2010 - 2016	24650	It is including various types of malware, such as trojans, backdoor, and ransomware, with total of 71 malware families	currently unavailable	http://amd.arguslab.org/
AndroZoo [48]	2011- now	15,972,640	The authors implemented a crawler to collect samples from several resources such as Google Play, anzhi market, App China, Torrents, and [43]	Request	https://androzoo.uni.lu/
AndroCT [49]	2010 - 2019	35974	Concentrated on dynamic profiles based on function calls	Yes	https://zenodo.org/record/4470320#.YBEtnehKhPY
Artifacts [50]	2017	125	Built a dynamic dataset with 59GB traces of method calls and Intent-based inter-component communication	N/A	N/A
Cai et al. [51]	2010 - 2017	30634	This dataset relies on tracing method calls and inter-component communications (ICCs) of android apps	Yes	https://www.dropbox.com/s/2s05kalgkcqwchn/andro-evo-supplement.zip?dl=0.
DroidFax [52]	2010-2017	17664	Examined how android apps built in terms of concentrating on static and dynamic analysis features	Request	http://chapering.github.io/projects/droidfax/
Kharon's Dataset [53]	2011-2016	3,000,000	The authors used several resources such as [43], [45], as well as implemented a craft tool to collect data in terms of static and dynamic analysis with developing seven malware	Request	http://kharon.gforge.inria.fr/

first experiment, they used self-written malware (calculator, countdown, and money converter), and obtained 60 execution traces for each app: 50 related to normal apps, while 10 were related to malware; and found that the Crowdroid was able to distinguish the apps with an accuracy of 100% for all apps. The second experiment relied on employing real malware, using the Steamy Window app with PJAPPS,[1] (records classified information such as IMEI, the ID of the device, and contact numbers, in order send it to the server), and Monkey Jump2 with HongTouTou [2] (has the ability to install apps and insert spam in context), and generated 20 feature vectors (15 as normal, and five as malware). Using k-means, the detection rate for the Steamy Window and Monkey Jump2 apps were 100% and 85%, respectively.

Shabtai *et al.* [66] proposed a host-based intrusion detection system (HIDS) model using a modified knowledge-based temporal abstraction (KBTA) model [67] in terms of their functionalities, computations, and resources to adapt it to use by smartphones rather than personal computers (PCs). The proposed model observes the behavior of Android apps' to distinguish normal apps from malware apps as shown in Figure. 2.

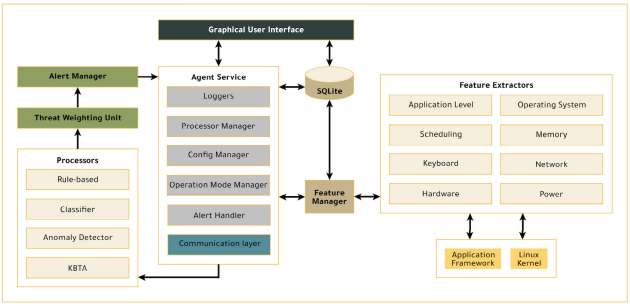

FIGURE 2. Host-based intrusion detection system architecture [66].

The authors developed five malware apps, as defined in Table 6. They evaluated their system based on the CPU consumption of the IDS and observed approximately 3% CPU consumption with peaks of less than 9% when processing data with a 2 seconds sampling interval.

Alpcan *et al.* [68] proposed a model that relies on a probabilistic diffusion method to identify patterns of smartphone use. The authors examined the relationships in daily usage and computed the dependencies of the instances and features.

[1] http://bit.ly/juL7Rh
[2] http://bit.ly/iOu5AA

TABLE 6. Description of applications.

Application	Definition
Schedule SMS and Lunar lander	consists of two sub-apps: the first aims to send messages to others who are in the contact list, and Lunar is a game app able to read content and send SMS messages.
HTTP upload	able to access, read, and write to/from an SD card in order to submit this information to a predefined address.
Snake	able to take a picture in order to send it to a remote server.
Tip calculator	aims to consume CPU.
Malware injection	downloads abnormal apps in order to gain permissions for outgoing calls.

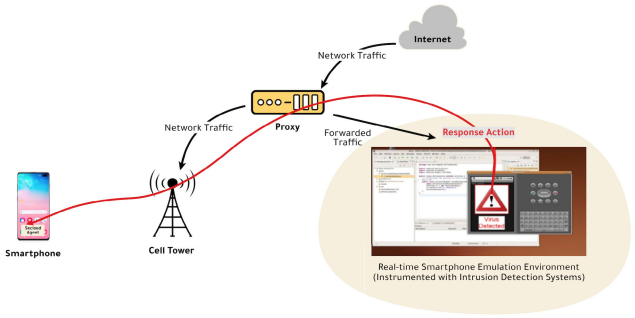

FIGURE 3. Secloud's high-level architecture [73] comprised of: (1) Cloud agent used for downloading the app, then registering the owner's information, allowed to record activities in order to send it to the emulator. (2) The emulator has a copied version of the original device with similar content files and operating states. (3) The proxy server, which manages traffic between real devices and the emulator.

To generate the mapping between features and instances, they used a bipartite graph algorithm [69] and implemented a diffusion graph to compute the equivalent vertices in the graph. To evaluate the model, Alpcan *et al.* utilized two datasets: a private dataset that relies on user activities, and the MIT dataset [63] in terms of phone calls, SMS messages, and communication logs over 244 days. They performed two experiments. The first experiment utilized the private dataset, and the L2 normal distance [70] between the current monitor feature and a feature vector was utilized in a training set to examine similarities, and then the Mahalanobis distance [71], Kullback-Leibler divergence and self-organizing maps (SOM) [72] were applied to compute the weight of the SOM neurons. The authors selected the top 3, 6 and 9 deviation days from 200 days, and calculated precision and recall metrics. Comparing the Kullback-Leibler divergence, Mahalanobis distance, and L1, the Kullback- Leibler divergence achieved better results for all deviation days, with recall and precision of 1 and 0.50, respectively.

A cloud-based security framework called Secloud was introduced by Zonouz *et al.* [73], which comprises three parts, as illustrated in Figure.3. The authors initially utilized DroidDream malware, which was distributed into 50 apps, and then performed eight different attacks in both a real device and a virtual machine to examine infected apps. Secloud framework was able to detect seven out of eight attacks. They also observed the resources consumed and found that the overhead of CPU usage was 4.63% for the real device, while for the emulator, the average CPU utilization, when idle, was 10.96%, which increased to 27.17% when it was running. They also measured the memory consumption and found that regular checking of security when the device was running and idle consumed 41.17% and 39.39%, respectively.

Alam and Vuong [74] detected Android malware by building a model that leveraged a machine learning algorithm and extraction/selection features. The authors employed Random Forest (RF) [75], in terms of the different settings of trees (4, 6, 8, 16, 32), the depth of each tree (1, 2, 4, 8, 16, and 32) with out-of-bag (OOB) error, root-mean-square error (RMSE) (the square root of the mean squared error) based on 5-cross validation, and how many samples were identified correctly. They selected and modified a dataset [76] of 7535 normal and 24807 malware apps, and extracted 42 features based on battery, binder, CPU, memory, network, and permissions. Among different settings, the best RMSE was 0.0171, OOB: 0.0002, Root MSE:0.0171− > Tree 160, TPR is 0.999857.

Dampolous *et al.* [77] implemented a model to allow a dynamic movement from the cloud to the host, without modification of the IDS using cloud and host based. The cloud side is compromised by a cloud manager (observing activities using a detection engine) and the detection manager on the device. There are three components; event sensor (gathering data and extracted features), system and detection manager (select which detection method is used to identify malicious patterns), and response manager (handling suspicious actions identified by the detection manager). The authors, then introduced four detection mechanisms: SMS profiler (examining suspicious use of services and recognizing unknown malware), iDMA (observing the pattern of current apps to detect malware), iTL (responsible for touching activities), and Touchstroke (observing keystroke actions). The researchers used iPhone 5s as a host and prepared a cloud environment using Amazon's Elastic Compute Cloud (EC2). They employed random forest (RF) [75] as the classifier with 17 extracted features based on the detection mechanism, and obtained accuracies between 80.6% and 99.6%. They examined CPU usage and battery consumption for all detection methods in the device and cloud as shown in Table 7.

Riad and Ke [78] implemented a floppy tool called RoughDroid, which involves three phases. First, floppy analysis attempts to identify seven features: hardware components, voice over internet protocol (VOIP) calls, app activities, API calls, location, and sending or receiving messages. This phase can be performed in a parallel sweep to check the Android app and collect the features. Second, a vector space is created once the features are elicited, and then mapped into joint vector spaces with 550,000 features overall (features are

TABLE 7. Performance results for CPU, memory, and battery in the testing phase [77].

Type of Detection	Scenario	CPU (%)	Memory (%)	Battery (mAh/sec)
SMS Profiler	Device	97	79	0.10
	Cloud	21	62	0.07
iDMA	Device	100	83	0.20
	Cloud	25	62	0.07
iTL	Device	100	88	0.21
	Cloud	45	62	0.05
Touchstroke	Device	98.7	79.7	0.14
	Cloud	26	62.0	0.05

TABLE 8. Histogram features.

Item	Experiment1	Experiment 2
Number of detection algorithms	6	6
Number of feature selection methods	3	3
Number of devices	5	5
Number of feature groups	3	3
Number of iterations	20	20
Total number of runs	5400	5400
Testing on applications not in the training set	-	+

valued geometrically). Third, the authors implemented a detection algorithm that is divided into a features table to represent apps and features, $FS = (A, F)$, respectively, Decisions: $FS = (APP, C\ U app_d)$, $appd$: decision features, C: conditional features, and finally approximation, which categorizes into low, upper and boundary approximation to approximate the accuracy. In this phase, each row represents a determined feature obtained from a specific feature set from a given Android app. The authors utilized [44] and selected 137171 apps (131611 normal apps and 5560 malware apps) to validate their scheme. They obtained a detection rate of 95.60% and an FPR of 1%.

Shabtai and Elovici [79] proposed a host-based framework which continuously observes features and activities such as CPU, memory, and battery usage called Andromaly. Andromaly performs threat assessments by computing the weight of a given app in order to generate an alarm if the app is classified as malware. The authors installed 23 games and 20 tool apps on an HTC G1 smartphone and extracted 88 features. To rank the extracted features, the authors applied information gain (IG) [80], chi-square [57], and fisher score (FS) [81] techniques. To evaluate Andromaly, the authors utilized k-means [65], logistic regression [82], histogram [83], decision tree [84], Bayesian network [85], and naïve Bayes [86] techniques and evaluated Andromaly in two scenarios, which relied on including and not including game apps in the training set. The histogram features are presented in Table 8. Among all the configurations, for the first scenario, the decision tree used the top 20 features from IG with a TPR of 0.9973, FPR of 0.004, AUC of 0.998, and accuracy of 0.997. For the second scenario, the best configuration was linear regression using the top 20 features by FS, with a TPR of 0.828, FPR of 0.199, AUC of 0.888, and accuracy of 0.818.

Curti *et al.* [87] investigated the impact of the relationship between energy consumption and malware in smartphones by comparing normal apps and malware activities. The authors selected energy and Wi-Fi consumption features and implemented two models. The first model is the general consumption, which aims to observe the energy consumption of all hardware components, e.g., GPS and Bluetooth, and defined the global consumption equation below.

$$C = \sum_{i=1} f_i + g_i = B + P_s \tag{1}$$

Here, f_i is the base consumption of the i-th component, g_i consumption is related to a specific (legal/malicious) activity of the i-th component, and $B = \sum_i f_i$ is the base consumption of all components, and $P_s = \sum_i g_i$.

The second model is called the Wi-Fi consumption model, which monitors the incoming and outgoing wireless network traffic, and calculates consumption using the following equation.

$$E(p) = E \times p = (P_{txpkt}) \times p \tag{2}$$

Here, p is the number of packets, $E(p)$ is the energy value for p packets, and E is the energy value for a single packet. To evaluate their approach, the authors used Sony Xperia U and a Samsung S Advance, and defined an experimental protocol that relied on observing the connection to the modem/router to utilize energy. Therefore, they created a dataset of signatures using YouTube, Skype, and Shazam apps, and then implemented two attacks: a ping flood (to represent illegitimate incoming traffic) and repeated HTTP GET requests (for outgoing traffic). For an integrated Skype call with a ping flood attack, in which the scenario started with a Skype call for 312 sec, and then the attack was released. The authors found that the consumption increased from 652 mA to 673 mA after only 22 sec, Skype could no longer connect due to lost bandwidth. Another experiment was performed to investigate browsing YouTube while simultaneously submitting HTTP GET requests. Here, the users could access YouTube and an attack was initiated approximately five minutes later. They found that the energy consumption increased from 690 mA to 780 mA, and the battery level consumed was 25%.

Damshenas *et al.* [88] introduced a cloud-based malware detection framework called M0Droid which tracks installed apps to create a signature for each app. M0Droid is comprised of a server analyzer and client agent app, as illustrated in Figure 4.

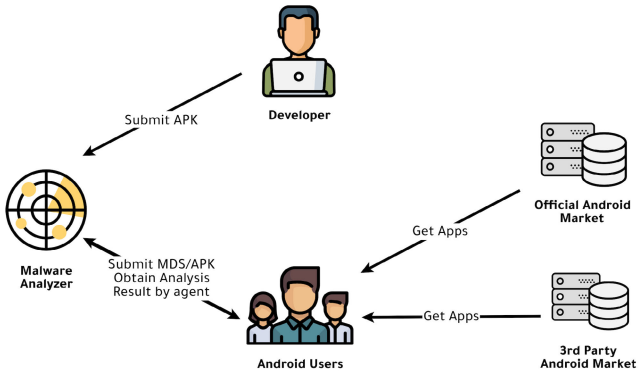

FIGURE 4. **M0Droid architecture [88]. Server analyzer that examines an app's APK file by employing a virtual emulator to observe device activities, and a client agent app that runs in the background to determine whether the app is in the database (if not, it sends an SHA1 checksum to the server for further analysis).**

To extract the features, the authors used asset packaging, while employing the Android debugger for communicating with the emulator. Damshenas *et al.* [88] constructed two private whitelist databases: one of which contained 100 normal apps, while the another was a blacklist database that contained 100 malware apps [43]. They selected 246 apps from both lists (black and white), and M0Droid obtained accuracy of 60.61%.

Merlo *et al.* [89] investigated how energy consumption can be used to identify intrusions against mobile devices by identifying high-, low-, and middle- levels measurements. In the high-level measurements (monitoring apps), the authors utilized PowerTutor [90] to read hardware component actions from the /sys file and calculate the power consumption of the CPU, Wi-Fi, GPS, 3G, display, and audio. The calculation process at this level relies on computing a single component per second, and it was found that a Skype call, WiFi and CPU had a fixed consumption of 631 mW, 719 mW, and 200 mW, respectively. They found that the total power consumption of a Skype call, for example, increased from 960 mW to 1000 mW. The second level was low-level measurement (focusing on battery consumption). The authors launched an AB8500 driver to extract features and identified examples of consumption that exceeded 250 ms. The third level was a middle-level measurement. The authors identified three attacks, that is a Skype call, a ping flood combined with a Skype call, and a ping flood attack, and found that the values were Skype call: 70 sec and 127 mW; ping flood attack: 70 sec and 15 mW; and combination attack 70 sec and 145 mW.

In their study, Yuan *et al.* [91] examined Android malware detection in terms of the extraction of three main features: system state (CPU usage, battery consumption, memory usage), processes (process ID and CPU utilization), and network traffic. The authors applied the naïve Bayes to identify the patterns. To evaluate the model, they collected 60 apps as a private dataset (45 normal apps and 15 malware apps) using an HTC G10. Among these apps, they randomly selected 15 apps, downloaded them to the device, and launched the malware detection model, observed the activities that represented as vectors, and categorized them as normal and malware classes. They produced various attack samples, and categorized the data into three groups with a varying number of samples (mixing samples at a ratio of $1 : H$). For group one, the total number of samples were 504, and the number of attack samples were 51. For group two, the number of total samples were 802, and the number of attack samples were 82, while group three comprised of 860 samples and 80 attack. The model achieved accuracies of 76.2%, 88.5%, and 87.2% for groups one, two and three, respectively.

James *et al.* [92] detected Android malware by employing signal processing and statistical learning. The authors initially performed ground truth determination using fast fourier transform, a low-pass butterworth filter, and inverse fast fourier transform techniques. Then, they employed blind source separation and normalization to compare similarities (in power consumption behavior) by computing the correlation of the reference signal. Two experiments were performed. For the first experiment, they predefined seven duty cycles (0% (no intrusion), 1%, 2%, 3%, 4%, 8%, and 12%), implemented a malware app, ran YouTube as a normal app for five minutes, and obtained an accuracy varying from 65% to 100%. The second experiment was performed using SVM [62], which extracted the vector of the features from a time series of abnormal signals and calculated the mean power in mW for the first feature and standard deviation (SD) for the second feature. Using 105 malware signals with various applied duty cycles, they achieved an accuracy of 70%.

Kou and Wen [93] introduced a lightweight intrusion detection system (LIDS) model that has four main phases. First, a modified Libpcap source file captured network packets depending on the transmission control protocol (TCP)/IP stack. Second, they utilized Snort technology, which has a high detection probability and cross-platform features. Third, predefined rules were set and stored in a local database. Fourth, a database was constructed to store all performed actions. Figure 5 outlines the workflow of the proposed model.

Portokalidis *et al.* [94] implemented a cloud-based malware detection approach called Paranoid Android (PA), which consists of a client (smartphone device that has a tracer tool to track activities for submission to a server) and a cloud environment with a replayer that is responsible for receiving activities and launching a proxy to connect to the Internet and intercept traffic. Figure 6 summarizes the architecture of the PA approach.

Portokalidis *et al.* employed two detection methods: a virus scanner and dynamic trait analysis, to evaluate the scheme in terms of the amount of trace data produced, overhead, and server performance. In relation to the amount of trace data, the authors observed different situations, such as, when the device is idle (64 B/s), making a call (21 B/s), browsing (2K), and audio playback (22.5 MB). For the overhead, they observed various activities such as browsing, making a call, and playing audio, and found that the average overhead was

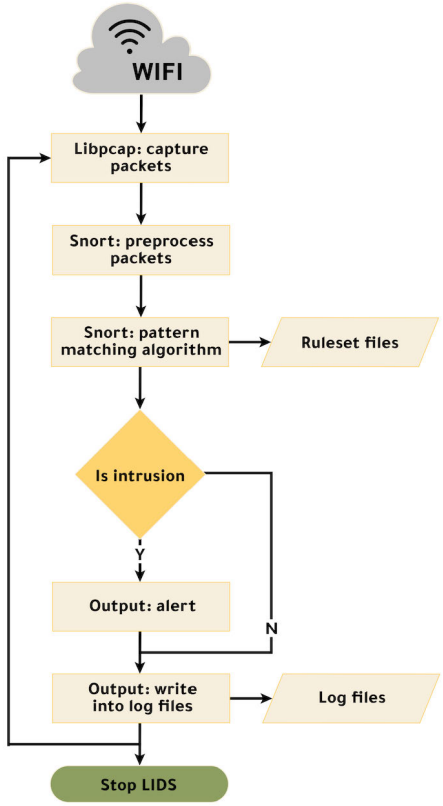

FIGURE 5. LIDS workflow [93].

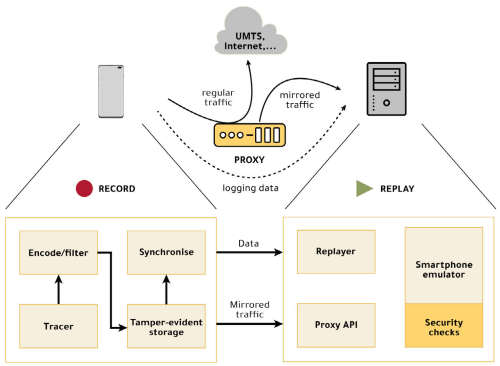

FIGURE 6. Overview of Paranoid Android architecture [94]: The virtual environment of the phone emulator synchronizes with a real device, to perform dynamic analysis to detect potential threats using system call features.

lower than 2.5 KiBps. In relation to performance and scalability, they examined 100 device replicas, and found CPU utilization of: running games (90% and 100%), accessing audio (20% to 25%), browsing (30% to 100%), and idle (0% to 5%).

Using the DroidBox model, Wei *et al.* [95] extracted the features from DNS packets to obtain information from response packets (e.g., IP addresses, duration of each record) using independent component analysis, which were used to determine patterns in malware. The authors created their own

TABLE 9. Evaluation of the detection of unknown samples.

Classifier	TPR	FPR	TNR	FNR	ACC	AUC
Random forest	1.000	0.029	0.971	0.000	0.975	0.998
PART	1.000	0.117	0.883	0.000	0.900	0.917
JRIP	1.000	0.043	0.957	0.000	0.964	0.979
Ensemble methods	1.000	0.021	0.979	0.000	0.982	0.989

dataset of network patterns and domain names, comprising 310 types of malware from several resources including Symantec, F-Secure, Lookout, and Panda Security, and then identified 102 unique malware instances. DroidBox conducts dynamic analysis of app behavior, examined the patterns, extracted network activities such as incoming and outcoming networks, and then sorted them. They obtained average accuracy of 50% - 98%, depending on the size of the features. They conducted another experiment using naïve Bayes and logistic regression to identify malware (with a feature size of 10) and obtained an accuracy between 81.74% and 100%, precision between 69% and 72%, and recall between 88% and 94%.

Kumar *et al.* [96] designed a network-based model to examine the network flows in communication. Therefore, traffic was produced for Android apps (600 samples were produced for malware apps using Andrubis [97] and Cuckoo [98]) using the Wireshark tool. Then, they employed the RFC- 5103 BitFlow export method [99] to extract the connection duration, destination port, packets sent, packets received, payload bytes sent, payload bytes received, initial flags in forward and reverse directions, and the union of flags in forward and reverse directions. The authors conducted several experiments using ensemble methods J48 [84], RF [75], JRIP [100], ripple down rule learner (RIDOR) [101], and a partial decision tree (PART) [102]. The authors performed three ensemble combinations. The first combination relied on J48, RF, JRIP, RIDOR, and PART, and obtained accuracy values of 99.1%, 98.5%, and 99.8%, using majority voting, maximum probability, and the product of probabilities [96]. The second ensemble combination comprised J48, RF, JRIP, and PART, and obtained accuracy values of 99%, 98.9%, and 99.1% for majority voting, maximum probability, and product of probabilities, respectively. The third ensemble method included a combination of J48, RF, and PART, and obtained accuracy of 99.3%, 98.7%, and 98.9% for majority voting, maximum probability, and product of probabilities, respectively. The results are listed in Table 9. Another experiment was conducted to examine unknown samples with and without interval times using RF, PART, JRIP, and ensemble methods. The results are presented in Tables 9 and 10.

Narudin *et al.* [103] developed a framework for detecting Android malware. The authors downloaded 20 normal apps from Google Play, accessed each app for varying times (10, 20, and 30 minutes) and captured network traffic,

TABLE 10. Evaluation of the detection of unknown samples after intervals of time.

Classifier	TPR	FPR	TNR	FNR	ACC	AUC
Random forest	0.932	0.038	0.962	0.068	0.955	0.947
PART	0.926	0.112	0.888	0.074	0.897	0.893
JRIP	0.919	0.047	0.953	0.081	0.945	0.937
Ensemble methods	0.939	0.025	0.975	0.061	0.966	0.957

then installed 1000 malware apps from [43] and manually collected 30 malwares, which refer it to the latest malware dataset. They utilized TCP packets and tShark[3] and extracted 11 features: source/destination IP address, source/destination host port number, frame length and number, http protocol used to submit data from client to server, number of frames received by unique source/destination in the last T s from the same source, number of packets flowing from source to destination, and vice versa. The authors applied the ClassifierSubsetEval method to rank extracted features. Figure 7 summarizes the implemented framework.

The authors employed RF [75], J48 [84], multi-layer perceptron (MLP) [104], Bayesian network [85], and k-nearest neighbors (KNN) [105] techniques to construct the framework, and performed three experiments. The first experiment was based on a fixed sample size (400 samples), and RF obtained the best results among the rest with TPR and FPR of 99.96% and 0.04%, respectively. Then, they performed another experiment based on varying sample sizes, and RF achieved higher results in terms of TPR and FPR with 99.97% and 0.34%, respectively. The third experiment was performed using the latest malware dataset and found that KNN achieved the best result in terms of TPR, FPR, precision, and recall with values of 84.57%, 15.43%, 88.4%, and 84.6%, respectively.

Ribeiro *et al.* [106] implemented an Android malware detection approach, called Host-based IDS (HIDS). The HIDs comprises six phases. In the first phase, a private dataset constructed with 12,000 apps (6,000 normal apps from Google Play, and 6,000 malware apps from Marcher, Android Locker, Secrettalk_Device, AndroidXbot, and Radardroid2Map). The second and third phases rely on extracted and normalized features. Ribeiro *et al.* extracted 15 features: CPU and memory consumption, cached memory, total sent/received bytes per second (kB/s), total sent/received packets per second (packet/s), battery usage and temperature, running process, downloaded apps, screen ON/OFF, total number of open TCP sockets, number of messages sent to unlisted numbers and outbox, and number of outgoing calls. In the fourth phase, the authors used machine learning and statistical classifiers to classify the entry of the datasets as normal or malware app, then produced a binary vector; $y \in \mathbb{R}^{m*1}$. The binary vector will be employed to assess the probability of intrusion for a period of n times using the following equation.

[3]http://www.wireshark.org/docs/man-pages/tshark.html

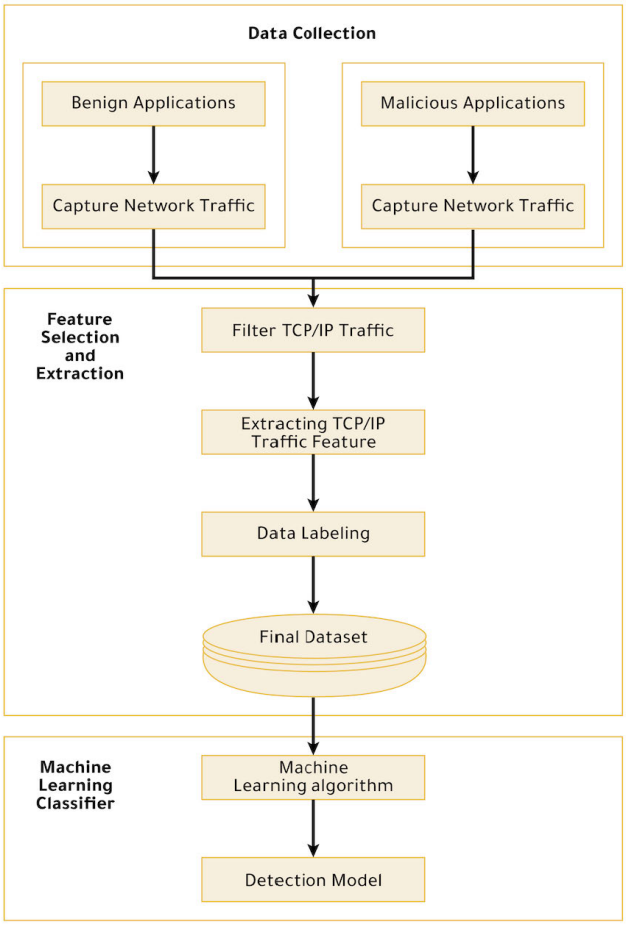

FIGURE 7. Experiment workflow of developed framework of [103].

$$P_{0n} = \frac{\sum_{i=1}^{m} Y_i}{m} A \tag{3}$$

Here, A is the accuracy obtained and m is the number of samples. The overall probability of intrusion for the previous and existing periods of n is computed in the fifth phase using the following equation.

$$P(n) = 1 - \Pi_{i=1}^{\alpha}(1 - P_0(n - \alpha + i)) \tag{4}$$

Here, α denotes the number of consecutive alerts received up to the n^{th} data acquisition period. In the sixth phase, the researchers evaluated the implemented approach using different percentages of malware instances in the training and testing phases. Among machine learning and statistical algorithms, the approach achieved accuracy between 91.12% and 100%.

Barbhuiya *et al.* [107] introduced a cloud-based detection process called DroidLight. The authors utilized the client side to collect various activities such as CPU utilization, submitted network packets every 5 seconds, stored them on a secure digital card, and forwarded stored activities to the server every 5 minutes. The evaluation phase was performed on the

server-side. The authors implemented three malware apps: DroidDDoS (sends thousands of network packets to a remote web server), DroidThief (stores a copy of files inside the SD to transmit them to a remote web server), and DroidHijack (consumes CPU). Meanwhile, the authors used three assumptions, which relied on the status of the device, WhatsApp and YouTube apps utilized, and the sample size. The authors obtained accuracies ranging between 86.7% and 96.6% based on sample size, and between 0% and 100% for identifying the malware family. They also monitored the consumption of CPU and memory used to identify DroidLight, and found that it consumed 0.86% of the CPU resources and 4.6% of the memory resources.

Wu *et al.* [108] implemented a static analysis framework, called DroidMat, which initially extracted permissions, activities, services, receiver, and provider features, and then represented those features as vectors. The authors utilized 1738 apps (1500 normal apps and 238 malware apps from Google Play and [43], respectively). The authors examined the average usage for all samples and found that malware apps had higher usage in terms of permission, service, and receiver component features than normal apps. Then, the authors employed k-means clustering [65] and the expectation-maximization (EM) algorithm [109], and utilized the singular value decomposition method to determine the clustering required based on low rank approximation. They also used the KNN [105] to determine whether the apps were normal or malware, and found that DroidMat obtained recall, precision, F-measure, and accuracy of 87.39%, 96.74%, 91.83%, and 97.87%, respectively.

In their study, Zhu *et al.* [110] utilized a dataset consisting of 2130 apps (1065 normal apps and 1065 malware apps from Google Play and VirusShare), and then extracted four static features: permissions, sensitive APIs, system events, and URLs. The authors constructed a DroidDet system and employed the term frequency-inverse document frequency (TF-IDF) [111] and cosine similarity [112] to compute and rank each feature. The authors identified a permission rate metric to distinguish between normal and malware apps, as follows.

$$PR = \frac{pm}{sz} \tag{5}$$

Here, PR refers to the permission rate, pm is the total number of permissions requested by each app, and sz is the size of the Smali file in MB. The authors [110] used a rotation forest adapted with principal component analysis [113], and SVM. For the rotation forest, they obtained accuracy, recall and precision of 88.26%, 88.40%, and 88.16%, respectively, while SVM obtained accuracy, recall, and precision of 3.33%, 2.22%, and 4.03%, respectively.

A framework called DREBIN was introduced by Arp *et al.* [44] to detect intrusion among Android devices. The authors used a dataset consisting of 123,453 normal apps installed from Google Play, Chinese and Russian app markets, and 5560 malware apps using [43]. The authors extracted various features, including permissions, API calls, and network access features that were mapped into a joint vector space for further analysis. The authors used a linear SVM and performed two scenarios. In the first scenario, the authors compared DREBIN and popular AV scanners using whole samples, and found that DREBIN achieved accuracy of 93.90%, while the best AV scanner achieved accuracy of 96.41%. The authors used in the second scenario the malware samples, and found that DREBIN achieved accuracy of 95.90%, while the best AV scanner achieved accuracy of 98.63%.

In their work, Aafer *et al.* [114] developed a DroidAPIMiner tool using a dataset consisting of 19,987 apps (16,000 normal apps from Google Play and 3987 malware apps from McAfee and [43]), extracted API calls and package-level information and requested permission features for each app, and then ranked these features in terms of the probability of threats to produce a set of feature vectors associated with class labels. They employed decision tree [84], C4.5 [115], KNN [105], SVM [62] techniques. The authors computed the differences in permission request usages between normal and malware apps and selected the top k permissions that were commonly requested by malware. The authors selected the top 80 k permissions feature and 169 kk package level with varied parameters of API features and achieved accuracy of 67%, and 99%, respectively.

Milosevic *et al.* [116] conducted two experiments utilizing permissions and source code analysis features using 400 apps (200 normal apps and 200 malware apps [88] to detect malware intrusion. The first experiment extracted permission features, employed SVM, naïve Bayes, J48, JRIP, and AdaBoost, and reported results in terms of precision, recall, and F-score. Among all classifiers, AdaBoost achieved better results in terms of precision, recall, and F- score with values of 89.5%, 89.4%, and 89.4%, respectively, while the second experiment relied on analyzing the source code using the bag-of-words method [117] to train the models using J48, naïve Bayes, SVM with sequential minimal optimization, RF, JRIP, logistic regression, and AdaBoostM1 with SVM- based techniques. Here, they combined algorithms using a majority voting decision system and examined clustering using the farthest first, simple k-means, and EM algorithms. Comparing among all classifiers, AdaBoostM1 obtained better results for Precision, Recall, and F-score with values of 95.8%, 95.7%, and 95.6%, respectively.

Saracino *et al.* [118] were able to detect app patterns by implementing a multi-level anomaly detector scheme for Android Malware (MADAM) at the kernel, app, user, and package levels. Figure 8 illustrates the components of the MADAM scheme, which consists of app risk assessment, global monitoring, pre-app monitoring, and user interface. The authors extracted 14 features and classified them into five groups: system calls, SMS, critical API, user activity, and app metadata. Then, KNN [105], linear discriminant classifier [119], quadratic discriminant classifier [120], MLP with backpropagation [104], Parzen classifier, and

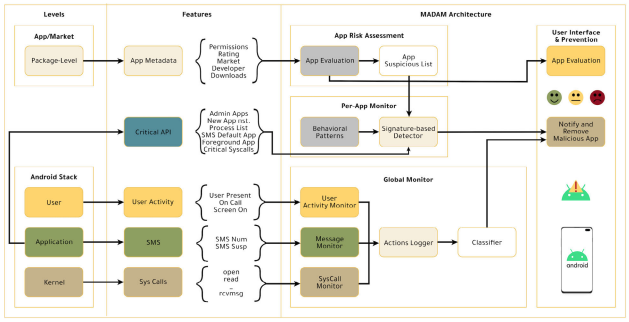

FIGURE 8. **Multi-level Anomaly Detector for Android Malware (MADAM) scheme [118].**

radial basis function [121] techniques employed for initial evaluation using 2,784 malicious apps from 125 families, from [43], [45], and VirusShare. The best reported accuracy was 96.9%. A further experiment performed by the group created three different pattern usage configurations: low, normal, and heavy usages, using 91 downloaded apps. They used the KNN [105] technique to detect malware. The results are presented in terms of FP, FPR, and FP/day as follows: 3, $1.10^{-}5$, 0.5 (light usage); 8, $2.8.10^{-}5$, 1.1 (medium level); and 75, $2.6.10^{-}4$, 10.7 (heavy usage). Benchmark testing is based on the calculated overhead consumption of CPU, memory, I/O, 2D and 3D graphic video games, with total consumption of 0.9%, 9.4%, 4%, 0%, and 0%, respectively.

Lu *et al.* [122] collected 400 apps (200 normal apps and 200 malicious apps) and extracted permission logs, which were initially saved in a sequential manner in CSV format, and then the file was translated into (.scale) for later categorization of the features. The authors utilized signatures in antivirus programs and applied a LibSVM [123] algorithm to classify the apps, and obtained 100%, 0%, 2%, 98%, and 99% for TP, FN, FP, TN, and recognition rate, respectively.

Yerima *et al.* [124] implemented a reverse engineering tool to parse a .dex file into several files, and then convert them into feature vectors. The authors used a dataset with a total of 2000 apps (1000 normal apps and 1000 malicious apps), extracted 58 requested permissions, applied MI [56] to rank the features, and identified the 48 most relevant features. The authors employed the naïve Bayesian classifier and used the top 5, 10, 15, and 20 k features with varied training sets of 100, 250, 500, 1000, and 1600. The developed scheme achieved accuracy of 92.1% using 1600 samples. Later, the authors [125] further developed their work by extracting 358 features of API calls, SMS, commands, and permissions, and categorized the extracted features into app attribute features, permission features, and a combination of these features. To rank the extracted features, they employed MI and selected the 10, 15, 20 and 25 top features, and concluded that using the top 10 features achieved accuracy of 88.4%, TPR of 85.4%, TNR of 91.3%, FPR of 8.7%, equal error rate (EER) of 11.7%, and AUC of 89.5%, whereas for $k = 50$, the results for accuracy, TPR, TNR, FPR, EER, and AUC were 97.5%, 97.3%, 97.7%, 2.3%, 2.5%, and 99.3%, respectively.

Merlo *et al.* [126] designed a tool to observe the Wi-Fi and CPU energy consumption using three methodologies. The first methodology, called high-level measurement, observed and read actions in relation to CPU, Wi-Fi, display, 3G radio, GPS, and audio through the /sys file to compare each consumption level, using a profiler service (saving activities on the phone) and PowerTutor (responsible for the aggregated profiling of the devices). The second method involved low-level measurement and collected data regarding battery voltage and performed an experiment by sampling at 4 Hz using a demon service. The third measurement, called the middle level, used an extracted power utilization feature, utilizing Skype calls, ping flood attacks, and Skype activities to assess power consumption in terms of low and high application levels. They found that Skype lost 0% of power before a ping flood attack and saw a 1% loss once the attack was initiated. The average measured traffic was 12%. The average error was $6.56E - 2$, with a maximum value of $1.65E - 1$ and a minimum value of $1.25E - 3$.

Shabtai *et al.* [127] proposed a scheme comprising a GUI (to observe communication with users), an alert handler (to produce alerts), and a feature extractor and aggregator (to identify and measure features in terms of a specific period of time, and to analyze app behavior). The authors constructed a dataset consisting of 15 apps (5 normal apps and 10 malicious apps), performed an evaluation based on the original normal apps that requested network access permission and injected repackaged apps to monitor the traffic patterns of popular, normal, and malicious apps. The authors examined the system and selected eight apps, 150 samples, and 20 features, then employed linear regression [128], decision tree [84], SVM [62], Gaussian process for regression [129], isotonic regression [130], and decision regression tree [131] techniques. The proposed scheme achieved a TPR of 82% and accuracy of 87.5%. Anomalous records for different versions of the same app were detected using a decision tree and achieved TPR of 91% and ACC of 94%, while for self-updating malware, it showed a low detection rate of between 45% and 67.9%. Finally, the authors measured the CPU consumption (utilizing a Galaxy S GI-i900) of $13\% \pm 1.5$, and the time to learn a model for 50 samples was $249ms \pm 27.4$.

Talha *et al.* [132] developed a APK Auditor tool, consisting of three components. The first component is the Android client, which utilized for downloading Android apps, and then requested permissions features were extracted. The second component is a signature database that used for storing the extracted features. The central server, which is considered as the third component, employed to monitor the communication between the client and the database. To evaluate their APK Auditor tool, the authors used a public dataset with a total of 8762 apps (1853 normal apps installed from the Google Play Store, and 6909 malicious apps from [44], [45], and [43]. They used the following equation to compute the

accuracy for distinguishing apps:

$$P = \frac{e^{a+bx}}{1 + e^{(a+bx)}} \quad (6)$$

Here, a is a logistic regression and X is an independent variable that calculates the app's malware score (AMS) for the APK Auditor, and found that the average accuracy was 88.28% for all predictions.

Rathore *et al.* [133] introduced a system for detecting adversarial attacks. The authors initially collected samples with a total of 11290 apps (5721 normal apps installed from Google Play, and 5569 malware apps from [44]), extracted permission features and represented them as feature vectors. To construct the system, the authors employed several classifiers, including logistic regression [82], SVM [62], DT [84], RF [75], boosting algorithms and deep neural network (DNN) [134]), ranking the features, and then selecting the top 5, 10, 15, and all 197 features. Comparing all classifiers and feature sets, RF obtained the highest accuracy with a value of 93.81% using all features; however, the system fell against adversarial attacks. Therefore, the authors developed new variants of malware using reinforcement learning, proposing two attacks: single-policy attack (white box), which has as complete knowledge of the detection system, and multi-policy attack (gray-box), which has no knowledge of the model architecture. The authors obtained fooling rates of 44.28%, and 29.44% for white and gray boxes, respectively.

Wang *et al.* [135] initially developed four intrusions: private leakage, SMS financial charges, malware installation, and privilege escalation, and then implemented an APK tool to extract features from Smali files to produce an API call graph using Androguard [136] to construct an adjacency matrix. They then implemented the WxShall-extended algorithm to detect malware by examining calls in relation to actions, and employed the machine's pattern chain state by defining a pattern graph. The system developed by [135] was called DroidChain, and used a dataset comprised of 1260 malware apps [43]. A comparison was performed using SMS, phone number, user account, financial charges, malware installation and privilege escalation features, and obtained a total accuracy of 73.29%.

Elish *et al.* [137] developed a workflow using the definition-use graph [138] to extract the TriggerMetric feature of each API call, which determined whether a trigger was executed by the user. The authors used 4117 apps (2684 normal apps from Google Play, and 1433 malware apps from [43]) as their dataset, and extracted features that created by analyzing data dependencies and specific control flows. The authors defined two rules classifiers. The first rule is based on assurance scores. The examining app classifies as a normal app if the assurance score V is equal to or greater than the assurance threshold T. In the second rule, the authors applied a weighted cosine similarity function [112] to compute the similarity between the vector of the app W, and the average malware vector M, and then used the exponential function 2^x to calculate the weighted similarity vector. The app is classified as a normal app if it is classified as normal in rules 1 and 2; otherwise, the app will be classified as a malware app. The authors computed assurance scores for known malware apps and normal apps, and found that 66.6% of the malware apps had positive assurance scores, while 80.5% of normal apps had 100% of assurance scores. The authors evaluated their workflow using known malware, in terms of accuracy and FNR, and gained 97.99%, and 2.1%, respectively.

Idrees *et al.* [139] designed a PIndroid approach to identify Android malware apps. The authors used a dataset consisting of 1745 apps (445 normal apps and 1300 malware apps from [43]–[45], VirusTotal, the Zoo, and MalShare), and then extracted permissions and intent features. The authors preprocessed features to produce a vector by applying an unsupervised randomization filter, and applied six classifiers: RF [75], naïve Bayes [86], decision tree [84], decision table [140], sequential minimal optimization [141], and MLP [104]. They reported that popular apps have fewer normal permissions than less-popular apps (8-16 permissions for popular apps and 306 permissions for less-popular apps). The average for using intents based on the degree of harm for normal apps was between 1 and 40, while for the malware apps it was in the range of 2 to 8. They used the Pearson correlation coefficient to calculate the correlation between intents and permissions and concluded that the decision table achieved better results for TPR, FPR, precision, recall, and AUC with values of 99.3%, 0.6%, 99.5%, 99.6%, 99.6% and 99.6%, respectively.

Isohara *et al.* [142] developed a model based on an analysis of the malware behavior. The model collects activities at the kernel level using a log collector to track system call actions in terms of process management (i.e., clone, execve, and fork) and file I/O (accept, bind, open, read, etc.). The authors analyzed the activities collected by implementing a log analyzer tool to examine potential threats using a signature detection method based on regular expressions to obtain error rate detection results. They used an HTC Magic smartphone and implemented a command program (strace) to collect activities and applied two mechanisms to reduce noise: selecting system calls and selecting a process tree. They selected five normal apps: Friend, Fring, Facebook, Norton mobile security, and a wireless tether app, and produced 16 different signatures. In an analysis of 230 sample apps, they found that 37 apps leaked sensitive personal data (IMEI and Android ID), 80 apps matched signatures that used the names of mobile ads, 14 apps executed exploit code, and 13 apps executed the ''su'' command.

Xie *et al.* [143] proposed a behavioral-based malware detection system (pBMDS) that relies on observing and learning from the transitions of app and user patterns. The screen touch and keystroke events are collected to construct a profile of each user to analyze processes and build a graph pattern, which will be used as input patterns to learn user behavior for the apps. The authors applied a hidden Markov

model (HMM) [144] to examine system calls to determine if the existing traces belong to malware by calculating the probability of sequence observation. The authors implemented three types of malware: Cabir, Commonwarrior, and Lasco, and then created an attack strategy based on SMS with ten subjects, with a total of 317 complete and 29 incomplete messages. The system was evaluated in terms of detection rate, FPR and FNR as illustrated in Table 11.

TABLE 11. Experiment results based on a training size of 125.

	SMS	MMS	Email	Bluetooth
#HMM states	25	40	35	35
Detection rate (%)	92.1%	96.4%	95.2%	94.9%
FPR (%)	6.3%	2.8%	3.7%	4.5%
FNR (%)	1.6%	0.8%	1.1%	0.6%

Shamshirband and Chronopoulos [145] designed a scheme that employed a high-performance extreme learning machine (HP-ELM). The authors used two datasets: a malware dataset which consisted of 1280 apps (20 normal apps and 1260 malware apps from [43]), while the second dataset was based on CTU-13 and contained 13 real botnets. Here, tcpdump was used to capture and select features: DstAddr, sport, Proto, SrcAddr, Dport, Dur, State, sTos, dTos, TotBytes, SrcBytes and TotPkts features. The authors used the F-score [146] and Fisher score [81] to identify the most informative features. The designed scheme was able to obtain accuracies using the malware dataset of 96.73% and 96.89% for all features and the top five features, and for the second dataset, it obtained accuracies of 96.25% and 95.9% for all and the top five features, respectively.

Another static model called MaMaDroid was implemented by Onwuzurike *et al.* [147] to detect Android malware. The size of the collected samples was 43,940 samples (8,447 normal apps from Google Play store and 35,493 malware apps from Drebin [44]). The authors employed Markov chains to construct feature vectors to represent the sequences of abstracted API calls for each app, and utilized PCA to select informative features. Then, they performed a comparison among four classifiers: random forests, 1-Nearest Neighbor (1-NN), 3-Nearest Neighbor (3-NN), and SVM, to evaluate the model in terms of detecting unknown malware samples, and obtained an F-measure of 0.99. They also examined their model using sustainability within time spans of one and two years, and achieved accuracies of 0.87 and 0.75, respectively.

In their study, Zhang *et al.* [148] proposed a framework called APIGRAPH, which consisted of a *node*, representing a key entity such as an API call, exception, permissions and *edge*: showing the relation between two entities. The framework converts API entities into embedded clusters by extracting API semantics from the graph. The authors constructed a private dataset spanning of 6 years (between 2012 and 2018) which consisted of 322,594 samples (290,505 normal apps and 32,089 malware apps), and performed experiments using four frameworks: MamaDroid [147], DroidEvolver [149], Drebin [44], and Drebin-DL [150] using RF, Model Pool, SVM, and DNN. The authors performed two types of analyses: maintainability and sustainability analyses. For the maintainability analyses, they examined the number of malware to label for a period between 2013 and 2018 for a fixed retrain threshold, and found that APIGraph was able to save the number of samples to label, with values of 33.07%, 37.82%, 96.30% and 67.29%, for [44], [147], [149], and [150], respectively, while for the varied retrain threshold, the authors used the AUT (F_1, 12m) metric for each classifier, which was introduced by [151], and showed that the APIGraph is significant for all ratios and for all frameworks. Another evaluation analysis based on sustainability was based on examining all classifiers before and after leveraging the API relation graph, and found that the average improvement for [44], [147], [149], and [150] was 19.2%, 19.6%, 15.6% and 8%, respectively.

Suarez-Tangil *et al.* [152] proposed a DroidSieve system to detect obfuscated and non-obfuscated samples of Android malware. The authors collected a dataset with a total of 124479 samples (107,078 normal apps and 17401 malware apps from McAfee Goodware, [43], [44], [46], and Marvin Goodware). Then, they extracted resource-centric feature, such as the certificate generated, package directory and extension, and syntactic features such as API calls and permission. To construct the model, the authors employed four machine classifiers: SVM, RF, extra trees, and XGBoost. The authors evaluated DroidSieve in terms of detecting malware and classifying malware families. For examining malware, DroidSieve achieved 99.82%, 99.81%, 98.42%, and 0.008%, for accuracy, F1-score, detection rate, and false positive rate, respectively, while for identifying malware family, the DroidSieve gained 98.12% and 97.84% for the accuracy and F1-score, respectively. A further evaluation performed based on a mix of obfuscated and unobfuscated malware, the DroidSieve detected the malware with accuracies of 99.71% and 99.15% for families identification.

Avdiienko *et al.* [153] developed a mining unusual data flow (MUDFLOW) system that relies on three phases. First, the authors collected Android samples with total of 18204 apps (2,866 normal apps from Google Play store, and 15,338 malware samples from [43] and VirusShare). Second, they extracted static taint analysis features, based on sensitive resources, such as calendarContact Provider, accessed URL, intent, and non-sensitive sources and sinks such as wallpaper app. Third, SVM was employed as a machine learning, and obtained accuracy of 86.4%. Among all collected apps, the authors found that 10,552 apps were considered as malicious apps in terms of leaking sensitive data.

Tables 13, 14, 15 and 16 summarize static feature analysis approaches.

B. DYNAMIC ANALYSIS APPROACH

This approach uses various features such as analyzing the system components that include CPU usage, processes running

at the kernel level of the OS, and user/app-level functions, such as Bluetooth and WiFi. It is also important to study how smartphone owners interact with their devices in various ways, for example, pressing buttons, zooming, navigating pages, and tapping the screen. All these observations have to be performed during the run times of the apps.

Alzailayee *et al.* [154] introduced a dynamic analysis scheme called DanLog, which monitors the pattern and signature of API calls to identify the lower-level features, while extracting 31 dynamic features such as phone state and SMS messages received for the high-level feature. The authors collected Android samples comprising 1940 apps (970 normal apps and 970 malware apps from McAfee Labs), and then employed a sandbox to compare the obtained results before and after enhancements. For example, for the feature of getDeviceId, which belongs to TelephonyManager category, the results before and after using the sandbox were 10 and 14, and for the feature of getLineNumber, that belongs to the TelephonyManager category, the results were 1 and 8 in terms of with and without using the sandbox. Another comparison performed to compare the normal and malware apps with respect to 44 extracted features from high-level behavior, granular events, and API calls. Among all features, PHONE_STATE feature was found in 905 malware apps and 537 normal apps, while SEND_MESSAGE was found in 6 malware apps and zero normal apps.

Later, Alzaylaee *et al.* [155] compared the effectiveness of using emulator and real devices in terms of the extraction of dynamic features. They modified DynaLog [154] to import a contact list to the device using the SD card, identify which apps were downloaded from a third party in order to delete them, and examined the phone's status, i.e., airplane mode, battery level, and outgoing calls. Using the same dataset used in [154], two environments were prepared: an actual device and a constructed emulator environment. A total of 178 features were extracted, ranked, and were represented as 0 or 1. Then, they employed SVM [62], naïve Bayes [86], simple logistic [82], MLP [104], RF [75], and J48 [84]. In the first experiment, they ran samples in both environments, and found that the actual phone environment was able to identify apps with accuracy of 94.3%, while the emulator obtained identified apps with 70.5% accuracy. The second experiment aimed to detect malware apps in both environments, using the top 100 features, and obtained TPR of 92.4%, FPR of 0.98%, TNR of 90.2%, and FNR of 0.76%, using the MLP classifier.

In their study, Yerima *et al.* [156] compared three methods. The first method is a random input generation method that used the Monkey tool to produce pseudo-random events, such as clicks, swipes, screen touches, and scrolling. The second method is a state-based input generation method that uses actions in the apps such as states and events as transitions, utilizing the DroidBot tool. The third method is a hybrid generation method that incorporates random- and state-based models. The authors used two datasets: the first dataset, *dataset*1, was used in [154], while the second dataset *dataset*2 comprised 13530 apps. They extracted 178 dynamic features that relied on API calls and intents, and then utilized SMO [141], naïve Bayes [86], simple logistic [82], MLP [104], PART [102], RF [75], and J48 [84] classifiers. They examined the features in terms of comparable behavioral footprints among random and state models, observed code coverage using *dataset*1, and obtained F-measures of 92.6%, 94.3%, and 93.4% for random, state, and hybrid models, respectively. Incorporating dynamic (API calls and intents) and static (permissions) features in *dataset*2 improved the results obtained from 87.7%, 86.7%, and 83.2% to 93%, 92.6%, and 91.8% for state-based, hybrid, and random-based scenarios, respectively.

Alzaylaee *et al.* [157] generated two types of input: stateful and stateless (random-based), by utilizing the DynaLog tool [154]. They initially distributed 8 Android devices, with SIM cards to allow calling, texting, and connecting to a network. They also added an SD card containing various types of data such as photos, text, and other multimedia. With this procedure, the authors downloaded 100 apps per device every day and installed 100 apps per day. Therefore, they constructed a dataset comprising 31125 apps (19620 normal apps and 11505 malware apps). The authors extracted 300 requested permissions, 97 features from app attributes, and 23 action event features. To evaluate their system, the authors employed MLP with different configurations, and the best obtained results were TPR of 97.6%, TNR of 90.86%, FPR of 9.14%, FNR of 2.24%, and accuracy of 95.21%. They also evaluated their system using naïve Bayes [86], simple logistic [82], RF [75], SVM with radial base function, SVM linear, decision tree [84], and PART [102], and found that the MLP classifier outperformed the other seven classifiers in relation to TPR of 99.56%, precision of 98.09%, recall of 99.56% and w-FM of 98.82%.

Vidal *et al.* [158] built a scheme to observe the patterns of downloaded Android apps based on their origin. The authors first tracked the activities to capture sequence cells for all apps, which were identified by the Zygote process. This process aims to activate the server to perform further comparisons between normal and malware executions shared by users. Hence, if an app is labeled as rightful, the new sequences are knowledge-based candidates. Therefore, 2000 apps were monitored; then, the analysis action was performed to produce metrics, such as scores gained by aligning sequences of system calls of legitimate and suspicious apps, to make a decision about the nature of a suspected app. The authors [158] applied a sequence-alignment algorithm derived from the Needleman–Wunsch algorithm [159], which provides a calibration function (finding the average saturation length of a set of reference samples), and enhanced the accuracy using a previous scoring function [160], which expressed as follows:

$$F(X_i, Y_j) = \begin{cases} 1 & \text{if } X_i = Y_j \\ 0 & \text{if } X_i \neq Y_j \end{cases} \tag{7}$$

Here, $F(X_i, Y_j)$ is defined as the similarity between items X_i and Y_j. Sub-datasets were constructed from the

original dataset with respect to various numbers of instances of sequence alignment processes. The authors selected two main datasets: DREBIN [44] and Genome [43], with a total of 2850 apps, and extracted 300 boot sequences for each app. The analysis was performed one app at a time to observe the patterns, and obtained 98.61%, 6.88%, 95.8%, and 96.8% for TPR, FPR, accuracy, and AUC, respectively.

Wu and Hung [161] implemented a dynamic analysis approach called DroidDolphin. The authors initially monitored and recorded 25 API calls for analysis purposes, and then downloaded a repackaged APK in an Android virtual device (AVD) to gather various information, such as logs of incoming and outgoing network data, as well as activities that involve write and read operations. They employed an n-gram [162] to represent features with 56354 dimensions. To evaluate DroidDolphin, the authors employed LibSVM [123] using a private dataset comprising 64,000 apps. Among the different quantities of samples, the approach achieved accuracy of 86.1%, F-score of 85.7%, recall of 82%, and precision of 90% using the quantity of $32K32K$. The second scenario tested relied on recollection data, called a re-log mechanism, which examined 500 normal and 500 malware apps over eight run times, and obtained accuracy of 78.6%, F-score of 78.7%, recall of 77%, and precision of 81%. The authors also performed an evaluation comparing cross-validation and the testing set, using quantities of $32k32k$, and obtained accuracy of 86.1% and 92.5% for cross-validation and the test set, respectively.

An automated and continuous malware detection system called DroidEvolver was introduced by Xu *et al.* [149], which consists of initialization and detection phases. The initialization phase is responsible for dealing with known apps labeled as normal and malware apps, producing two types of sets, features and detection models, and then submitted to the detection phase. During this phase, the DroidEvolver extracted APIs, recorded the Android API binary presence, and produced the feature vectors for all apps, which will be sent into the detection phase. The detection phase aims to classify unknown samples, and provides updated information to feature and model sets. To evaluate DroidEvolver, the authors used AndroZoo [48] dataset to collect 68,016 apps (33,294 benign apps and 34,722 malware apps). The authors compared their system with MaMaDroid [147] in terms of six time periods using F-measure, precision and recall metrics. The highest performance of F-measure, precision and recall among all sets for their system was 96.75%, 97.36% and 96.70%, while for MaMaDroid it was 81.99%, 91.99%, and 84.80%, respectively.

Cai and Jenkins [163] studied Android apps by tracing method calls and intent ICCs. The authors installed 3,000 normal apps from the Google Play store, and then used [164] to find possible app pairs based on matching the ICCs, then selected 20 pairs randomly to perform further experiments on the emulator. The authors defined 122 metrics, based on general, ICCs and security perspectives. Later, Cai and Jenkins [165] examined how the patterns of Android apps changed over time; therefore, they introduced a sustainable malware detector using feature engineering. The authors employed 122 behavior metrics from [163], and added further features: the extent, frequency, and distribution for the source and sink invocations of sensitive API calls. The authors downloaded 6432 apps (3431 normal and 3001 malware apps) from several resources with different years. They created two groups for the collected apps based on the developed year, then compared the metrics, and found 52 metrics to be the most informative out of 155. They, they built their system to detect known and unknown malware samples. The authors used the RF [75] classifier and obtained an accuracy of 93%. They trained their classifier based on a sustainability metric over a span of five years, and achieved accuracy of 82%.

Grace *et al.* [166] implemented a scheme called Risk-Ranker that observes the irregular behavior of unknown apps and categorizes these patterns into high, moderate, and low risk. Figure.9 illustrates the systematic architecture of Risk-Ranker, which is classified into first- and second-order analyses. Among 15 different Android markets, the authors collected 118,318 apps including 718 malware apps and 322 types of zero-day malware. Using first- and second-order analysis, Risk-Ranker recognized 220 and 499 malware apps, respectively. The scheme found that 9877 apps had native code, and 499 were referred to zero-day attacks.

Droidscribe is a method introduced by Dash *et al.* [167] aims to perform multi-class classification of Android malware. This approach initially employed CopperDroid [168] to extract system calls and their arguments in terms of four main features: network access, file access, binder methods, and execute file. Then, the authors collected 5,560 samples from [44] and employed SVM, Conformal Prediction (CP) [169] and hybrid prediction (CP augmenting SVM). Among the three methods used, hybrid prediction was able to obtain the highest performance, with 94% accuracy, 92% precision, and 96% recall.

Sun *et al.* [170] presented a cloud-based malware detection framework called Patronus, which consisted of client and server sides. The client-side determines if there is a predefined policy and extracts the permission features for each process to produce an alarm, while the server side is responsible for finding any policies not checked on the client side, which are considered threats that aim to bypass the system. To validate Patronus, the authors used 737 apps (500 normal apps from Google Play and 237 malware apps from [43]and [45]), and employed the Monkey tool to produce 500 events, such as click, touch, and gesture actions. They initially calculated the overhead for CPU, memory, I/O, 2D, 3D and found values of 0.9%, 8%, 10.9%, 0.1%, and 1.6%, respectively. Then, the authors investigated the impact of battery usage using a game app, and observed that their system consumed 3% more battery power than the original setting. Finally, they computed the precision and recall, which gave values of 87% and 92% for BaseBridge (213 samples), 100% and 100% for FakeAV (9 samples), and 65% and 69% for MobileTx (15 samples).

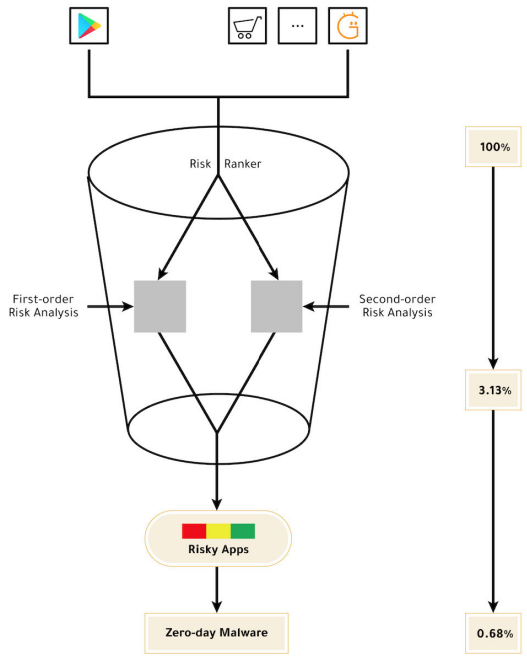

FIGURE 9. Risk-Ranker architecture [166]. The first-order analysis concentrated on alarms for high- and medium-risk apps. The second-order risk observes apps behaviors, such as encryption and dynamic code. The analysis process provides a sorted list that rates each app based on the level of potential harm.

A system that leverages systemic app-level profiling and a machine-learning algorithm was proposed by Cai *et al.* [171], called Droidcat. The authors built a dataset consisting of 34343 apps (17,365 normal apps and 16,978 malware apps). They examined each app for ten minutes, accessed the APK through the Monkey tool, and reduced the number of samples to 271 apps (136 normal apps, 135 malware apps). The authors defined 122 metrics based on structure (method calls, declaring classes, callback), ICC (intent, carrying data through URI only), and security (distribution of sources and sinks). Cai *et al.* employed RF [75] as the classifier for detecting malware, and obtained precision, recall and F1-scores of 97.53%, 97.34% and 97.39%, respectively. They performed another comparison to classify the malware family, and achieved precision, recall and F1 of 97.96%, 97.91%, and 97.84%, respectively. Finally, they examined their approach in terms of sustainability, using apps with a span of eight years, resulting in small standard deviations in F1-score of 1.34-2.38%.

Cai [172] implemented DroidSpan system, which is based on behavioral profiling and evaluated it in terms of sustainability metrics. The author gathered apps for a period of eight years and created a dataset comprising 26382 samples (13,627 normal apps and 12,755 malware apps, from Google Play, VirusShare and [48]). Then, Cai extracted 52 features: the extent of sensitive access (4 features), categorization of sensitive data and operations accessed (22 features), and sensitive method-level control flows (26 features). To construct the system, the author compared various machine-learning methods: RF, SVM with both linear and radial basis function kernels, decision trees, k-NN, naïve Bayes with three models (Gaussian, Multinomial, and Bernoulli), AdaBoost, gradient tree boosting, extra trees, and the bagging classifier, in terms of F1-measure, recall and precision, and found that RF obtained the best performance compared to others. The author compared DroidSpan and MaMaDroid [147] in terms of stability and re-usability. For stability analyses, relies on same-period detection, DroidSpan achieved averages of 92.88%, 92.68% and 92.61% for precision, recall and F1- score, while MaMaDroid obtained 88.81%, 88.34% and 88.48% for precision, recall and F1-score, respectively. For stability analysis, the author calculated the average accuracy with 28 tests for training and testing data over seven different years and found that the baseline for MaMaDroid was 42.43%, while the baseline for DroidSpan was 71.81%.

Chen *et al.* [173] developed an adversarial attack method called DroidEye, which aims to bypass the detection of Android apps. The authors used a public dataset from the Comodo Cloud Security Center Lab comprising 14804 apps (8059 normal apps, and 6745 malware apps), then extracted a set of 812 features, including 105 permissions, 68 filtered intents, 8 application attributes, 330 API calls, 259 new instances, and 42 exceptions. Counting featurization was performed to transform the binary feature space into continuous probabilities to eliminate the adversarial gradient of the learning mode. Chen *et al.* employed two attack models: L0 [174] and L∞ [175], and used SVM to construct their method. The performance of DroidEye using TPR and FPR were 92.10% and 5.25%, respectively. Later, the authors [176] used two crafting algorithms, the adversarial objective function (C&W), and perturbing influential features based on the indicative forward derivatives (JSMA). Chen *et al.* applied these algorithms to two methods: MaMadroid [147], using a sequence of API call features and Drebin [44] using permissions, activities, and API call features. The authors considered different scenarios such as attackers knowing only the feature set, or they know both the feature and training sets. The collected data consisted of 11,439 samples (5879 normal apps and 5560 abnormal apps) from PlayDrone [177], and 1,000 random samples from [44]. Using a neural network, the model was able to detect malware and distinguish malware families with F-measures of 88% and 96%, respectively.

Karbab and Debbabi [178] introduced a framework called MalDy, by applying natural language processing using bag-of-words (BoW) to detect Android malware. The authors set up their framework using Android platforms (DroidBox (2016)[4]) and Windows platform (Win32). For the Android platform, the authors collected 56,000 samples from [43], [44], [48], Maldozer [179] and PlayDrone,[5] while they employed the ThreatAnalyzer sandbox using threat track[6] to

[4]https://tinyurl.com/jaruzgr
[5]https://archive.org/details/android_apps
[6]https://threattrack.com/

gather 20,000 instances for the Windows platform. Among these platforms, the authors extracted dynamic features such as file, network, register access records, then employed N-gram (count sequence of words, and produce event log features), feature hashing (calculated compacted feature vector), and term frequency-inverse document frequency (TFIDF) (compute the feature vectors of input pattern reports). The authors utilized CART, RF, extremely randomized Trees, KNN, SVM and XGBoost as machine learning algorithms. They compared the classifiers using the hashing function and obtained F1-scores of 99.61%, 99.62%, and 93.39% for the Android platform using [43], [44], and Maldozer [179], respectively, while they used hashing and TFIDF for the Windows platform, and achieved F1-scores of 94.86% and 95.43%, respectively.

In their study, Zhang *et al.* [180] developed a cloud-based malware detection system called SaaS. The authors implemented a tool to crawl samples from Android markets, which aimed to extract certification for each app and delete any duplicate samples, giving a total of 1495 collected samples. The authors extracted the instruction sequences of the DEX file and employed three algorithms: the fuzzy hash algorithm (responsible for constructing a fingerprint for each instance, and examined whether the app is repackaged or not), N-gram algorithm (utilized for extracting opcode n-gram features from the Dalvik virtual machine), and the gray level co-occurrence matrix (extracts an app's GLCM-6 features and produces a gray-scale image). Among 1495 samples, SaaS achieved accuracy of 99.6%. Further improvement was performed on the constructed model to deal with complicated networks using a larger sample comprising 13682 apps (5318 normal apps, 8364 malware apps). The authors extracted sensitive APIs and permissions. They employed J48, RF, SMO, naïve Bayes, and bagging, and found that the bagging classifier obtained the best precision of 96.5%, whereas when using RF the system achieved the best results in terms of TPR, FPR, and ROC with values of 96.3%, 0.038% and 99.2%, respectively.

Tables 17 and 18 summarize dynamic feature analysis approaches.

C. HYBRID ANALYSIS APPROACH

Onwuzurike *et al.* [181] modified a virtual device to capture the sequence of API calls from runtime execution traces of apps, and introduced an approach called AuntieDroid. The authors employed the Monkey tool and recruited participants to execute the APK samples with virtual devices to simulate apps. To identify the patterns and frequent analysis activities utilized by API call malware, Onwuzurike *et al.* developed a trace parsing tool using *dmtracedump* [182]. The total size of the dataset utilized was 5068 apps (2625 normal apps and 2443 malware apps). The process of extracting features relies on the family and packages of the app. A model based on abstracted sequences of API calls was created using a Markov chain technique [183], and RF [75] was selected as the classifier method. Three experiments were performed to assess AuntieDroid. Table 12 summarizes the result obtained using MaMaDroid, as a static method, AuntieDroid, as a dynamic method, and a hybrid approach.

TABLE 12. Experiment results using human and monkey stimulator in dynamic analysis.

Analysis method	Stimulator	Mode	F1	Precision	Recall
MaMaDroid (static)	-	Family	0.86	0.84	0.88
		Package	0.91	0.89	93
AuntieDroid (dynamic)	Human	Family	0.85	0.80	0.90
		Package	0.88	0.84	0.92
	Monkey	Family	0.86	0.84	0.89
		Package	0.92	0.91	0.93
Hybrid	Static & Human	Family	0.87	0.86	0.88
		Package	0.90	0.88	0.91
	Static & Monkey	Family	0.88	0.88	0.89
		Package	0.92	0.92	0.93

Tong and Yan [184] developed a framework that assumes two situations: know and unknown instances. First, for known instances, the framework produced a behavior for individual and sequential systems calls with different calling depths: $1; 2; \ldots X$, which are related to file and network accesses. Then, the frequency of sequential system calls for each depth of the app was calculated and expressed as a fraction, i.e., the number of times sequential system calls appeared compared to the total number of sequential system calls. The normal and malware apps were compared to create a set of observed patterns to differentiate the samples. Second, for unknown instances, the authors applied a dynamic method to record runtime activities with respect to Y, the depth of file and network accesses, extracted behavior related to the instances, and compared them to regular behavior to make a decision. The authors compiled a dataset of 2000 malware apps [43], implemented a tool to track process_id and process_name, and then integrated them with system calling.

Tong and Yan [184] generated two pattern sets: malware pattern sets (*MP*) and normal pattern sets (*NP*), with sizes of 430 apps (130 normal apps, and 300 malware apps), and 910 apps (267 normal apps, and 643 malware apps). They defined following parameters: *tn* and *tm* are threshold values to determine *NP* and *MP*, *Nt* and *Mt* are threshold values for normal and malware pattern matches. They used two set values: the first set is $tn = 1 : 9, Nt = 1, tm = 5 : 1$, and $Mt = 22$, and $Mt = 22$, and the second set is $tn = 2{:}31$, $Nt = 1$, $tm = 5{:}9$, and $Mt = 29$. The authors created four groups: group one contained 273 apps (126 normal apps and 147 malware apps), group two comprised 382 apps (187 normal apps and 195 malware apps), group three included 602 apps (293 normal apps and 309 malware apps), and group four had 920 apps (437 normal apps and 483 malware apps). The framework achieved accuracies of 90.19%

TABLE 13. Recent studies using static feature analysis method, where ACC: Accuracy, DR: Detection Rate, FPR: False positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
Crowdroid [64], 2011	5	3 Implemented two apps from Monkey Jump2 with HongTouTou	App behavior	k-means	DR	85%-100%
HIDS [66], 2010	5	Private	SMS messages, Access SD card, Permissions	KBTA	DR	94%
[68], 2010		Private, [63]	Activities, Phone calls, SMS messages, Communication logs	L2 [70], Mahalanobis distance [71] SOM [72]	Precision Recall	1 0.33
Secloud [73], 2013	50	NA	App Behavior	NA	CPU usage Memory consuming	4.63% 35.93% -39.39%
[74], 2013	32342	[76]	Battery, Binder, CPU & Memory usage, Network, Permissions	RF	RMSE OOB	0.0171 0.0002
Dampolous et al. [77], 2014	NA	NA	System calls, HW sensors, API calls, System devices, SMS & email messages	RF	ACC	80.6% - 99.6%.
RoughDroid [78] , 2018	137171	[44]	HW components, VOIP calls, App activities, API calls, Location, SMS	NA	DR FPR	95.60% 1%
Andromaly [79], 2010	43	NA	NA	k-means [65], Logistic regression [82], Histogram [83], Decision tree [84], Bayesian network [85], naïve Bayes [86]	TPR FPR AUC ACC	0.9973 0.004 0.998 0.997
Curti et al. [87], 2013	3	Private	Outgoing traffic	NA	Energy consumption Battery consumed	690 - 780 mA 25%.
M0Droid [88], 2015	246	Private	NA	NA	ACC	60.61%
Merlo et al. [89], 2014	3	Private	HW components	NA	Total power consumption	960 mW to 1000 mW
Yuan et al. [91], 2013	60	Private	CPU usage, Battery consumption, Memory usage, Number of processes & threads running, Downloaded apps, Inbound & outbound network traffic, Number of SMS and MMS messages sent	naïve Bayes	ACC	76.2% - 88.5%
James et al. [92], 2017	2	Private	Power consumption	SVM	ACC	0.70
PA [94], 2010	NA	NA	System call	NA	Energy consuming CPU utilization	7.6% - 30% 5%- 100%
DroidBox [95], 2012	310	Private	App behavior, DNS response packets	NA	ACC Precision Recall	0.8174- 1 0.69 - 0.72 0.88 - 0.94

and 91.76% for sets one and two, respectively. The authors also compared six types of malware, BaseBridge, FakeInstaller, DroidKungFu, Lotoor, FakeBattScar, and GoldDream, with accuracies of 77.78%, 89.71%, 80%, 85.42%, 46.43%, and 100%, respectively.

Lindorfer *et al.* [185] implemented a method integrates static and dynamic analysis to detect unknown apps, called MARVIN. The authors assigned a value for unknown apps between zero (indicates that the app is normal) and ten (indicates that the app is malware), while the middle values

TABLE 14. Continued: Recent studies using static feature analysis method, where ACC: Accuracy, DR: Detection rate, FPR: False positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
STREAM [195], 2013	1777	[43], VirusTotal Malware Intelligence, Google Play	Battery, Binder, Memory, Network, Permissions	RF [75], naïve Bayes [86], MLP [196], Bayesian network [85], logistic regression [82] , Decision tree [84]	DR	94.53%
Kumar et al. [96], 2017	600	VirusTotal, Google Play	Connection duration, Destination port, Packets sent & received, Payload bytes sent & received, Initial & union flags in forward and reverse directions	J48 [84], RF [75], JRIP [100], RIDOR [101], PART	TPR FPR TNR FNR ACC	0.939 0.025 0.975 0.061 0.966
Narudin et al. [103], 2016	1020	[43] , Google Play	NA	RF [75], J48 [84], MLP [104], Bayesian network [85], KNN [105]	TPR FPR Precision Recall F1-score	84.57% -99.97% 0.04% - 15.43% 0.884 0.846 0.845
Chen et al. [197], 2015	23	NA	Internet behaviors(HTTP GET, POST), Sensitive data (GPS, IMEI, IMSI), Host rationality	Three-step check action algorithm	Suspicious Normal Malware	11 10 2
Ribeiro et al. [106], 2020	NA	NA	CPU and memory consumption, Cached memory, Total sent/received bytes/package per second, Battery usage and temperature, Running process, Downloaded apps, Screen ON/OFF, Total number of open TCP sockets, Number of messages sent to unlisted numbers and outbox, Number of outgoing calls	One Rule [201], Decision tree [84], naïve Bayes [86], Bayesian network [85], logistic regression [82], RF [75], KNN [105], SVM [62], k-means [65], Univariate [199], Multivariate [200] Gaussian models	NA	NA
Ribeiro et al. [106], 2020	12000	Marcher, Android Locker, Secrettalk_Device, AndroidXbot, Radardroid2Map	NA	SVM and RF	ACC	0- 100%
DroidLight [107], 2020	3	Private	Network packets, Read from SD, CPU usage	NA	ACC CPU usage Memory consumption	0%- 100% 0.86% 4.6%
Sahs and Khan [201], 2012	2172	NA	Control flow graph, Permissions	NA	Precision Recall F1-score	Decreased from 0.90 to 0.10 0.80 - 0.95 Decreased from 0.90 to 0.30
DroidMat [108], 2012	1738	[45], Google Play	Permissions, Activities, Services, Receiver, Provider	KNN	Recall Precision F1-score ACC	0.8739 0.9674 0.9183 0.9787
DroidDet [110], 2018	2130	VirusShare, Google Play	Permissions, Sensitive APIs, System events, URLs	RF, SVM	ACC Recall Precision	0.8991 0.9122 0.8884
DREBIN [44], 2014	129013	[43], Google Play, Chinees and Russian Markets	Permissions, API calls, Network access features	SVM	ACC	0.9390

TABLE 15. Continued: Recent studies using static feature analysis method, where ACC: Accuracy, DR: Detection Rate, FPR: False Positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
DroidAPIMiner [114], 2013	19987	McAfee, [43], Google Play	API calls, Requested Permissions	Decision tree [84], C4.5 [115], KNN [105], SVM [62]	ACC	0.67 - 0.99
Milosevic et al. [116], 2017	400	NA	Permissions	SVM, naïve Bayes, J48, JRIP, AdaBoost, Logistic Regression	Precision Recall F1-score	0.895- 0.958 0.894-0.957 0.894- 0.956
MADAM [118], 2018	3183	[43], [45], VirusShare	Permission, sys calls, SMS, Critical API, User activity, App metadata	KNN [105], Linear discriminant classifier [119], Quadratic discriminant classifier [120], MLP [104], Parzen classifier, Radial basis function [121]	NA	NA
Aung and Zaw [202], 2013	700	NA	Permissions, Modified configuration files, Sent SMS messages, Calls	k-means [65] Euclidean distance [203], J48 [84], RF [75], CART [204]	TPR FPR Precision Recall AUC ACC	0.916- 0.978 0.064- 0.157 0.916- 0.849 0.916- 0.978 0.909- 0.87 0.9158- 0.9072
Yerima et al. [125], 2015	NA	NA	API calls, Sending/receiving messages, Commands, Permissions	naïve Bayes	ACC TPR EER AUC	0.884 - 0.975 0.854 - 0.973 0.117- 0.025 0.895 - 0.993
Yerima et al. [124], 2013	2000	NA	Requested Permissions	Bayesian classifier	ACC EER FPR TPR	0.921 0.079 0.061 0.904
Merlo et al. [126], 2015	3	Private	CPU, WiFi, Display, 3G radio, GPS, Audio through the /sys file , Battery voltage	NA	Average measured traffic Average error	12% $6.56E-2$
Shabtai et al. [127], 2015	15	Private	Requested network access permission, Traffic patterns of apps	Linear regression [128], Decision tree [84], SVM [62], Gaussian process for regression [129], isotonic regression [130], Decision regression tree [131]	TPR ACC	0.82 - 0.91 0.875- 0.94
Talha et al. [132] , 2015	8762	[45], [44], and [43], Google Play	Requested Permissions	AMS Algorithm	ACC	0.8828
DroidChain [135], 2016	1260	[43]	SMS , Phone number , User account , Financial charges, Nalware installation, Privilege escalation	WxShall-extended algorithm	ACC Precision Recall	0.7329 - 0.9388 0.7193 - 0.9968 0.4241 - 0.9285
Elish et al. [137], 2015	4117	[43] , Google Play	TriggerMetric feature of each API call	Rule-based classification, $DPVC$ Similarity function	ACC FNR FPR	0.9799 0.021 0.49
PIndroid [139], 2017	1745	[43], [45], [44], VirusTotal, the Zoo, MalShare	Permissions, Intents	naïve Bayes [86], Decision tree [84], Decision table [140], Sequential minimal optimization [141], MLP [104]	TPR FPR Precision Recall	0.952 - 0.993 0.006 - 0.033 0.956 - 0.995 0.956 - 0.996
Li et al. [205] , 2018	129013	[44]	Permissions required, API call	Deep Neural learning	F1-score AUC FPR Precision Recall	0.9608 0.9716 0.001 95.23 0.9608
Mas'ud et al. [206] , 2014	60	Private	Log files , System call	naïve Bayes [86], KNN [105], J48 [84], MLP [104], RF [75]	TPR FPR ACC	0.57- 0.90 0.23- 0.57 0.50-0.83

TABLE 16. Continued: Recent studies using static feature analysis method, where ACC: Accuracy, DR: Detection rate, FPR: False positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
Isohara et al. [142], 2011	230	Google play	Analyzed app	System calls	Leaked sensitive data Matched signatures Executed exploit code Executed the "su" command	37 80 14 13
pBMDS [143], 2010	3	Private	User activities, System calls	HMM [144]	DR FPR FNR	0.921- 0.964 0.028- 0.063 0.006- 0.016
HP-ELM [145], 2019	1293	[43], CTU-13	NA	NA	ACC	0.9625- 0.9673
Graf et al. [207], 2019	56392	Private	APK feature	Expert system [208] , Neural networks	TP FP ACC	0.0288 0.82 0.966
MaMaDroid [147], 2019	43940	[44]	Abstracted API calls	Random Forests, 1-NN, 3-NN, SVM	F1-score	0.75- 0.99
APIGRAPH [148], 2020	322,594		API call, Exception, Permissions	NA	NA	NA
DroidSieve [152], 2017	124479	McAfee Goodware, [43], [44], [46], Marvin Goodware	Generated Certificate, Package directory and extension, API calls , Permission	SVM, RF, Extra Trees, XGBoost	ACC F1-score DR FPR	0.9982 0.9981 0.9842 0.9842
MUDFLOW [153], 2015	18204	Google Play, [43], VirusShare	Static Taint	SVM	ACC	0.864
Rathore et al. [133], 2020	11290	Google Play, [44]	Permissions	LR, SVM, DT, RF, ET, AB, GB, DNN	ACC	0.9381

indicate adware. The authors then employed the discrete levels to classify the apps as normal, malware, or unknown. They collected 124,198 apps (84,980 normal apps from Google Play store, 11733 malware apps from [45] and [43]), and extracted 154,939 dynamic features such as those related to file and network operations, phone events, data leaks, dynamically loaded leaks, and dynamically registered broadcast receivers, and 342,004 static features such as Java package name, permissions requested by the app, publisher ID, and publisher's certificate. Among all the extracted features, they applied the fisher score to rank the features and linear classifiers L1 [186], L2 [70], SVM [62], to construct MARVIN. Among 124,189 apps, MARVIN classified them as normal (89980 apps), malware (11733 apps) and unknown (27476 apps). The performance of the MARVIN method was 99.76%, 99.85%, and 99.83% using SVM, L1, and L2, respectively.

Android firmware and pre-installed apps are considered solution to detect Android malware. Therefore, Zheng *et al.* [187] created a scheme combining static and dynamic analysis methods called DroidRay. The authors first created a private dataset consisting of 250 firmware and 24009 apps, extracted various features such as firmware name, product model, and Android version. The authors applied static and dynamic analyses in terms of system signature vulnerability (implementing two signature scenarios), network security analysis (host security and IP tables), and privilege escalation vulnerability detection. The researchers employed the signature message-digest algorithm (MD5) [41] and a package name to recognize the possible vulnerabilities of each app. All analyzed apps were stored in a database to generate the report. DroidRay was evaluated to examine application and system levels. At the application level, they found that 1947 apps were vulnerable to signature attacks, and 19 firmware had pre-installed malware. At the system level, 142 firmware had default vulnerabilities, 5 had malicious host files, and 249 had Java-level privilege escalation vulnerabilities.

Kabakus and Dogru [188] implemented a hybrid Android malware analysis tool called mad4a. The authors applied static analysis using PScout II, finding API calls and organizing the calls into groups based on extracted features, such

TABLE 17. Recent studies using dynamic analysis method, where ACC: Accuracy, DR: Detection rate, FPR: False positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
DAIDS [209] , 2014	NA	NA	Time of installed, and updated apps, Time of requested permissions, Requested features, CPU consumption, Memory level per process, Open connections per package,In & Out SMS messages	NA	CPU and memory Consumption Average power consumed	9.9% - 10% 163 mW
Alzaylaee et al. [157], 2020	31125	NA	Requested Permissions, App attributes, Action event	MLP, naïve Bayes [86], Simple logistic [82], RF [75], SVM with radial base function, SVM linear, J48 [84], PART [102]	TPR TNR FPR FNR Precision Recall	0.9956 0.967 0.033 0.0044 0.980 0.9956
Vidal et al. [158], 2018	2850	[44], [43]	Boot sequences for apps	Sequence alignment algorithm, Needleman–Wunsch algorithm [159]	TPR FPR ACC	0.9861 0.0688 0.958
Alzaylaee et al. [155], 2017	2444	McAfee Labs	Contact list, Downloaded apps, Airplane mode, Battery level, Outgoing calls	SVM [62], naïve Bayes [86], Simple logistic [82], MLP [104], RF [75], J48 [84]	TPR FPR TNR FNR	0.931 0.08 0.92 0.069
DroidDolphin [161], 2014	64000	NA	API calls	LibSVM [123]	ACC F1-score Recall Precision FP FN	0.786 0.787 0.77 0.81 0.19 0.23
DynaLog [154], 2017	2226	McAfee Labs	API calls, Logs, Events	NA	NA	NA
MaMaDroid & AuntieDroid [181], 2018	5068	NA	API calls	RF [75]	F1-score Recall Precision	0.85 - 0.92 0.80 - 0.92 0.88- 0.93
Karami et al. [210], 2013	20	NA	I/O , System calls, Network activities	NA	Classified information recorded Malware did not rely on user interactions	3 malware 17 malware
Risk-Ranker [166], 2012	105914		Tracked Behavior, Encryption and Dynamic code	NA	# of malware # apps had native code zero-day attacks	220 9877 499
Patronus [170], 2014	737	Google play, [43], [45]	Pseudo-random events		Precision F1-score	0.65 - 1 0.69- 1
Yerima et al. [156], 2019	15974	NA	API calls, Intents, Code coverage	SMO [141], naïve Bayes [86], Simple logistic [82], MLP [104], PART [102], RF [75], J48 [84]	F1-score	0.832-0.934% %
DroidEvolver [149], 2019	68016	[48]	API calls	NA	F1-score	0.8717- 0.9615

as location, cameras, calendars, calls, contacts, SMS messages, microphones, and storage, while they used an *aapt* tool to extract permission features. The authors used the monkeyrunner tool to control both the devices and emulator. To evaluate mad4a, the authors used a dataset comprising 5808 samples (2809 normal and 2999 malware samples)

TABLE 18. Continued: Recent studies using dynamic analysis method, where ACC: Accuracy, DR: Detection rate, FPR: False positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
Droidcat [171], 2018	34343	Private	Method calls, Declaring classes, Callback, Intent, Carry data through URI only, Distributions of sources, Sinks	NA	Precision Recall F1-score	0.9753- 0.9796 0.9734 - 0.97.91 0.9739 - 0.9784
DroidEye [173] , 2018	14804	Comodo Cloud Security Center	Permissions, Intents, App attributes, API calls, Exceptions	SVM	TPR FPR	0.9210 0.525
Chen et al . [176] , 2019	11439	[44]	Sequence of API calls, Permissions, Activities, API calls	C&W, JSMA, Neural Networks	F1-score	0.88- 0.96
MalDy [178], 2019	NA	[44], [43], [48], Maldozer and PlayDrone, Threat-Analyzer sandbox	file, network, register access records	CART, RF, Extremely Randomized Trees, KNN, SVM and XG-Boost	F1-score	0.9486- 0.9543
SaaS [180], 2019	1495	NA	App certification, Instruction sequences , Opcode n-gram, App's GLCM-6, APIs, Permission	J48, RF, SMO, naïve Bayes, Bagging	ACC TPR FPR ROC	0.995987 0.963 0.038 0.992
Droidscribe [167], 2016	5560	[44]	System calls & their arguments	SVM, CP, Hybrid method	ACC Precision Recall	72%- 94% 92% 96%
Cai and Jenkins [165], 2018	6432	NA	System calls (extent, frequency, distribution of the apps)	RF	ACC	82% - 93%
DroidSpan [172], 2020	26382	NA	behavioral profiling	RF	F-1 score	0.7181- 0.9388

from Google Play Store, ASHISHB malware,[7] [43], [44], and [45]. Based on API calls, the authors found that the contacts (2790 apps), calendar (10 apps), and location (3 apps) for malware apps requested permissions, while for the normal apps, they found contacts (2994 apps), location (3 apps), and camera (2 apps) requested permissions. They compared normal and malware apps using dynamic analysis and found that the average number of incoming and outgoing connections was 87 for malware apps and 233 for normal apps. The average uploaded size was 2 MB for malware and 18 MB for normal apps, and the average downloaded size was 5 MB for abnormal and 671 MB for normal apps. The average number of INTERNET _CLOSE was 519 for abnormal and 464 for normal apps.

Fu and Cai [189] developed a classification approach, and manually selected a subset of metrics that adopted by [171], [189], and [165]. The authors employed RF [75] as the machine learning classifier. and compared their approach with four approaches: MamaDroid [147], Droid-Sieve [152], Afonso [190], and RevealDroid [191], based on same-period and over-time settings. The authors used a dataset consisting of 24,780 apps (13,627 normal apps and 11,153 malware apps). For the same-period setting, they combined normal and malware apps, and obtained average accuracies of 93.75%, 88.05%, 82.36%, 86.71%, and 85.02% for [147], [152], [190], and [191], respectively. For the over-time setting, they achieved average accuracies of 71.43%, 63.67%, 29.87%, 49.91%, and 45.94%, for [147], [152], [190], and [191], respectively.

Vinayakumar *et al.* [192] employed a text-mining approach to detect Android malware using long short-term memory (LSTM), which aims to convert extracted permission features into vectors/words. The authors generated two dictionaries, mapping words to words with a total size of 500, and then built two matrices for training and testing, with fixed sizes for both of $1785 * 300$. These matrixes passed into embedding layers to construct a new matrix with a size of $500 * 128$, to be used in the LSTM layer. The performance was evaluated using the developed approach and achieved accuracy, precision, and recall of 89.7%, 91%, and 96%, respectively. Subsequently, the authors [193] applied LSTM using permission features as a static analysis with a size

[7]https://github.com/ashishb/android-malware

TABLE 19. Recent studies using hybrid feature analysis method, where ACC: Accuracy, DR: Detection rate, FPR: False positive rate, TPR: True positive rate, NA: Not available.

Approach & Year	# of samples	Source of samples	Extracted Features	Utilized Algorithm	Performance	
					Metric	Results
Tong and Yan [184], 2017	2000	[43]	System call, Sequential system calls, Network accesses, Runtime activities	NA	ACC	0.88 - 0.92
MARVIN [185], 2015	124198	NA	Developer's certificate, API calls,Java package name, Permissions requested, Publisher ID, File and Network operations, Phone events, Data leaks, Dynamically loaded leaks & registered broadcast receivers	, L1 [186], L2 [70], SVM [62]	ACC	0.9976- 0.9985
DroidRay [187],2014	24009	NA	Firmware name, Product model, Android version, System signature, Permission, Sending an SMS message, Silent patterns	NA	Signature vulnerabilities Pre-installed malware Default vulnerabilities Malicious host files Java-level privilege escalation vulnerabilities	8.1 7.6% 56.8% 2% 99.6%
mad4a [188], 2018	5808	Google Play Store, ASHISHB malware, [43], [44], [45]	Title and package name, API calls, Permission	NA	NA	NA
Fu and Cai [189], 2019	24,780	Private	NA	NA	F1-score	0.90 - 0.98
Vinayakumar et al. [192], 2017	558	[43]	Battery, Binder, Memory, Permission Requested	LSTM, RNN	ACC	0.874 - 0.939
Rad-HGC [194], 2019	1402537	Tencent Security Lab , Private	API call sequence , downloaded app, certified signature for the app, package name	SVM	ACC TPR	0.9631 0.9812

of 558 apps (279 normal apps, 279 malware apps from [43]), and dynamic analysis features using battery, binder, memory, and permission requested using the Monkey tool, using 1738 apps (408 normal apps, 1330 malware apps), and 6832 feature vectors. The Vinayakumar *et al.* listed different feature sets: battery + permission, binder + permission, Memory + CPU + permission, and network + permission, and then evaluated their approach using LSTM and RNN. The best accuracies using LSTM and RNN were 94.7% and 93.7%, respectively, using binder + permission feature se for both classifiers.

Hou *et al.* [194] applied a heterogeneous graph (HG) to detect Android malware by implementing two attack models: an adversarial attack model called HG-Attack and a defense attack model called Rad-HGC. These models rely on the capability of the attackers; for instance, how to compromise the devices, and the knowledge of how to use samples (available information) to compromise the devices. To construct their model, the authors used a public dataset from Tencent Security Lab with a sample size of 1,389,408 (547366 normal apps, 217107 malware apps, and 624,935 unclassified apps), and a private dataset with size of 13,129 apps. Based on the dataset, Hou *et al.* extracted API call sequences from the execution of each Android app, and then extracted semantic relations such as whether the app is downloaded on the device or not, the certified signature for the app, and the package name. The authors used the heterogeneous graph as the node classification, and employed meta-path-based random walks and heterogeneous skip-grams for detecting the malware. SVM downstream classifier after the learning representations of apps in HG were also utilized. The authors evaluated both attack and defense models, and found that HG-Attack was able to evade 5 devices out of 10 and injected 298 nodes out of 400. In contrast, Rad-HGC was able to detect attacks with an average accuracy of 96.31% and showed stability over time, with TPR of 98.12%.

Table 19 summarizes the hybrid feature analysis approaches.

V. OPEN PROBLEMS, LESSONS LEARNED, AND FUTURE TRENDS

As the number of smartphone users has risen recently, the potential for malware threats has increased, too. This section highlights open problems that might interest researchers, summarizes lessons learned from published papers, and provides a brief section on future trends.

A. OPEN PROBLEMS

The identification and detection of Android malware are considered hot topics in mobile security; however, there are problems available for consideration in both research and technical fields, which are listed below.

1) STEGANOGRAPHY IN MALWARE

Steganography is a technique that can be employed by attackers to hide files and extend the time before malware is recognized. The time required to detect and identify the intrusion is important because the risk of stealing information increases with time. Various existing studies have not considered the time required to detect malware; therefore, there is a need to develop an effective method that balances the detection rate and the time required to identify malware.

2) PRESERVING PRIVACY

Several frameworks have used cloud-based distribution methods to collect, analyze, and examine the validity of malware detection; however, they did not discuss how to protect the privacy of the collected data, which can lead to privacy leakages because gathering and storing data will take place in the cloud. Using a third party to perform additional experiments could also put cloud-based data at risk. Therefore, it is important to consider how to secure the developed frameworks that rely on cloud-based storage.

3) PUBLIC DATASETS

Although the number of smartphone users has increased, possible threats and attacks rise annually. It is important to deal with the patterns of developed malware; therefore, authors tend to use publicly available datasets, which have various weaknesses, for example, some of them are out of date, some of them do not consider different types of malware, and the sample size might not be sufficient. Looking at recent research, there are many approaches that utilize relatively small sample sets, that is, several hundred to a few thousand samples. With such sample sets, performance results may be impressive; however, the evaluation might not reflect the detection results in real-world situations. Performing an experiment with a large number of samples and updating samples could be one possible way to improve detection rates.

4) FAMILY CLASSIFICATION

Most existing studies have focused on detecting Android malware. However, they ignored family classification. In addition, classifying malicious code into families of related malware is an important step in forensic analysis and threat assessment. It allows analysts to identify malware that comes from the same source and likely follows similar malicious intent.

B. LESSONS LEARNED

Android malware detection has gained considerable attention; therefore, this paper has discussed more than one hundred papers with respect to the analysis of feature patterns. Based on this study, there are several lessons learned from recent approaches, which are listed below.

1) The constructing of Android malware detection relies on collecting the appropriate samples. Most of the current studies have used samples gathered since 2010 and employed unbalanced datasets, which can lead to unreliable results. Therefore, it is a crucial to collect instances that reflect malware behavior using a suitable sample size to evaluate the developed model.
2) Most existing studies have employed machine-learning algorithms. However, there are several additional available techniques that can be used to solve this problem, such as natural language processing, image processing, and deep learning algorithms, which can improve the performance of implemented frameworks.
3) Comparing and evaluating projects would be more effective and instructive if a common evaluation metric was to be adopted.
4) It is important to evaluate and compare works using the same dataset. There is a need to build a dataset and make it available for research purposes; therefore, it is essential to construct a public corpus that can be updated regularly, which contains normal and malware apps.

C. FUTURE TRENDS

Because the number of malware that is uploaded into official markets has been raised annually, attackers tend to employ innovative techniques to make the detection difficult in a short period of time. Therefore, the direction of the near future in this field should consider how to recognize the malware promptly, no need to retrain samples and must to be sustained. Employing different techniques such as deep learning algorithms, image processing and natural language processing may also be considered in the near future.

VI. CONCLUSION

Over the last decade, the number of smartphone owners has increased globally. Users can use these devices for various personal and professional operations and activities. Various activities have increased the amount of classified and valuable information that can be targeted by malicious actors. In turn, the techniques and methods used to infiltrate these devices have become more sophisticated. To overcome such attacks, researchers in both academic and industrial fields have developed methods for detecting malware apps.

This paper focuses on the risks associated with malware that targets mobile devices, and considers current approaches and mechanisms used to detect malware with respect to methodology, associated datasets, and evaluation approaches for studies in the mobile malware field published since 2010. Finally, we considered possible open problems that could be addressed in future studies, highlighted what we have learned, and introduced possible future trends in this field.

ACKNOWLEDGMENT

The author would like to thank the Deanship of Scientific Research at Umm Al-Qura University for supporting this work by Grant Code: 18-COM-1-01-0007. The author wishes to thank the anonymous reviewers for their helpful and valuable comments that greatly contributed to improving the manuscript.

REFERENCES

[1] Statcounter. (Mar. 2020). *Desktop vs Mobile vs Tablet Market Share Worldwide*. Accessed: Oct. 20, 2021. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide

[2] Statista. (Mar. 2020). *Computing Device Shipments Forecast Worldwide From 2013 to 2022, by Segment Type*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.statista.com/statistics/265878/global-shipments-of-pcs-tablets-ultra-mobiles-mobile-phones/

[3] Statcounter. (Mar. 2020). *Mobile Operating System Market Share Worldwide*. Accessed: Oct. 20, 2021. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[4] Statista. (Feb. 2020). *Mobile Operating Systems' Market Share Worldwide From Jan. 2012 to December 2019*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[5] A. Charland and B. Leroux, "Mobile application development: Web vs. native," *Commun. ACM*, vol. 54, no. 5, pp. 49–53, May 2011.

[6] Statista. (Jan. 2020). *Number of Apps Available in Leading App Stores as of 4th Quarter 2019*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[7] Statista. (Jan. 2020). *Number of iOS and Google Play Mobile App Downloads Worldwide From 3rd Quarter 2016 to 4th Quarter 2019*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.statista.com/statistics/695094/quarterly-number-of-mobile-app-downloads-store/ Jan. 2020,

[8] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices (SPSM)*, 2011, pp. 3–14.

[9] Statista. (Feb. 2020). *Mobile Malware Evolution 2019*. Accessed: Oct. 20, 2021. [Online]. Available: https://securelist.com/mobile-malware-evolution-2019/96280/

[10] V. Chebyshev. (Feb. 2020). *Mobile Malware Evolution 2019*. Accessed: Oct. 20, 2021. [Online]. Available: https://securelist.com/mobile-malware-evolution-2019/96280/

[11] M. LABS. *Mobile Signatures*. Accessed: Oct. 20, 2021. [Online]. Available: https://blog.malwarebytes.com/glossary/stalkerware/

[12] V. Kouliaridis and G. Kambourakis, "A comprehensive survey on machine learning techniques for Android malware detection," *Information*, vol. 12, no. 5, p. 185, Apr. 2021.

[13] S. Selvaganapathy, S. Sadasivam, and V. Ravi, "A review on Android malware: Attacks, countermeasures and challenges ahead," *J. Cyber Secur. Mobility*, vol. 10, no. 1, pp. 177–230, Mar. 2021.

[14] 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.

[15] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of Android malware detection with deep neural models," *ACM Comput. Surveys*, vol. 53, no. 6, pp. 1–36, Feb. 2021.

[16] V. Kouliaridis, K. Barmpatsalou, G. Kambourakis, and S. Chen, "A survey on mobile malware detection techniques," *IEICE Trans. Inf. Syst.*, vol. 103-D, no. 2, pp. 204–211, 2020.

[17] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surv. Tuts.*, vol. 17, no. 2, pp. 998–1022, 2nd Quart., 2014.

[18] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *Proc. Int. Conf. Inf. Syst. Secur.* Berlin, Germany: Springer, 2011, pp. 49–70.

[19] M. L. Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 1, pp. 446–471, 1st Quart., 2013.

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

[21] P. Yan and Z. J. S. Q. J. Yan, "A survey on dynamic mobile malware detection," *Softw. Qual. J.*, vol. 26, no. 3, pp. 891–919, 2018.

[22] A. Eshmawi and S. Nair, "Smartphone applications security: Survey of new vectors and solutions," in *Proc. ACS Int. Conf. Comput. Syst. Appl. (AICCSA)*, May 2013, pp. 1–4.

[23] A. Amamra, C. Talhi, and J.-M. Robert, "Smartphone malware detection: From a survey towards taxonomy," in *Proc. 7th Int. Conf. Malicious Unwanted Softw.*, Oct. 2012, pp. 79–86.

[24] S. Arshad, A. Khan, M. A. Shah, and M. Ahmed, "Android malware detection & protection: A survey," *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, no. 2, pp. 463–475, 2016.

[25] R. Zachariah, K. Akash, M. S. Yousef, and A. M. Chacko, "Android malware detection a survey," in *Proc. IEEE Int. Conf. Circuits Syst. (ICCS)*, Dec. 2017, pp. 238–244.

[26] M. Odusami, O. Abayomi-Alli, S. Misra, O. Shobayo, R. Damasevicius, and R. Maskeliunas, "Android malware detection: A survey," in *Proc. Int. Conf. Appl. Informat.* Cham, Switzerland: Springer, 2018, pp. 255–266.

[27] N. K. Gyamfi and E. Owusu, "Survey of mobile malware analysis, detection techniques and tool," in *Proc. IEEE 9th Annu. Inf. Technol., Electron. Mobile Commun. Conf. (IEMCON)*, Nov. 2018, pp. 1101–1107.

[28] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Hum.-Centric Comput. Inf. Sci.*, vol. 8, no. 1, pp. 1–22, Dec. 2018.

[29] P. Agrawal and B. Trivedi, "A survey on Android malware and their detection techniques," in *Proc. IEEE Int. Conf. Electr., Comput. Commun. Technol. (ICECCT)*, Feb. 2019, pp. 1–6.

[30] M. LABS. *Android/Adware.MobiDash*. Accessed: Oct. 20, 2021. [Online]. Available: https://blog.malwarebytes.com/detections/android-adware-mobidash/

[31] E. Kabirova. (2014). *10 Years Since the First Smartphone Malware–to the Day*. Accessed: Oct. 20, 2021. [Online]. Available: https://eugene.kaspersky.com/2014/06/15/10-years-since-the-first-smartphone-malware-to-the-minute/

[32] M. LABS. *Spyware.InfoStealer*. Accessed: Apr. 29, 2021. [Online]. Available: https://blog.malwarebytes.com/detections/spyware-infostealer/

[33] D. Maslennikov. (2011). *ZeuS-in-the-Mobile—Facts and Theories*. Accessed: Oct. 20, 2021. [Online]. Available: https://securelist.com/zeus-in-the-mobile-facts-and-theories/36424/

[34] D. Maslennikov. *Hummer*. Accessed: Oct. 20, 2021. [Online]. Available: https://malware.wikia.org/wiki/Hummer

[35] P. Shoshin. (2020). *PhantomLance Android Backdoor Discovered on Google Play*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.kaspersky.com/blog/phantomlance-android-backdoor-trojan/35234/

[36] J. Cannell. (2013). *Cryptolocker Ransomware: What You Need to Know*. Accessed: Oct. 20, 2021. [Online]. Available: https://blog.malwarebytes.com/101/2013/10/cryptolocker-ransomware-what-you-need-to-know/

[37] F. Tchakounté and F. Hayata, "Supervised learning based detection of malware on Android," in *Mobile Security and Privacy*. Amsterdam, The Netherlands: Elsevier, 2017, pp. 101–154.

[38] X. Jiang. (2011). *Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html

[39] M. Zaman, T. Siddiqui, M. R. Amin, and M. S. Hossain, "Malware detection in Android by network traffic analysis," in *Proc. Int. Conf. Netw. Syst. Secur. (NSysS)*, Jan. 2015, pp. 1–5.

[40] uniper Networks Mobile Threat Center (MTC). *Mobile Signatures*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.juniper.net/us/en/security/mobile-threat-center/

[41] R. Rivest and S. Dusse, "The MD5 message-digest algorithm," MIT Lab. Comput. Sci. RSA Data Secur., Cambridge, MA, USA, Tech. Rep. RFC 1321, 1992.

[42] D. Eastlake and P. Jones, "US secure hash algorithm 1 (SHA1)," Internet Soc., VA, USA, Tech. Rep. RFC 3174, 2001.

[43] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.

[44] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[45] M. Parkour. (2008). *Mobile Signatures*. Accessed: Oct. 20, 2021. [Online]. Available: http://contagiodump.blogspot.com/

[46] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on Android malware," *Comput. Secur.*, vol. 51, pp. 16–31, Jun. 2015.
[47] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2017, pp. 252–276.
[48] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, May 2016, pp. 468–471.
[49] W. Li, X. Fu, and H. Cai, "AndroCT: Ten years of app call traces in Android," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR)*, May 2021, pp. 570–574.
[50] H. Cai and B. G. Ryder, "Artifacts for dynamic analysis of Android apps," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, p. 659.
[51] H. Cai, X. Fu, and A. Hamou-Lhadj, "A study of run-time behavioral evolution of benign versus malicious apps in Android," *Inf. Softw. Technol.*, vol. 122, Jun. 2020, Art. no. 106291.
[52] H. Cai and B. G. Ryder, "A longitudinal study of application structure and behaviors in Android," *IEEE Trans. Softw. Eng.*, early access, Feb. 19, 2020, doi: 10.1109/TSE.2020.2975176.
[53] J.-F. Lalande, V. V. T. Tong, M. Leslous, and P. Graux, "Challenges for reliable and large scale evaluation of Android malware analysis," in *Proc. Int. Conf. High Perform. Comput. Simul. (HPCS)*, Jul. 2018, pp. 1068–1070.
[54] E. Rahm and H. H. Do, "Data cleaning: Problems and current approaches," *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, Dec. 2000.
[55] S. García, J. Luengo, and F. Herrera, *Data Preprocessing in Data Mining*, vol. 72. Cham, Switzerland: Springer, 2015.
[56] P. A. Estévez, M. Tesmer, C. A. Perez, and J. M. Zurada, "Normalized mutual information feature selection," *IEEE Trans. Neural Netw.*, vol. 20, no. 2, pp. 189–201, Feb. 2009.
[57] M. L. McHugh, "The chi-square test of independence," *Biochemia Medica*, vol. 23, no. 2, pp. 143–149, 2013.
[58] S. Akaho, "A kernel method for canonical correlation analysis," *arXiv preprint cs/0609071*, 2006.
[59] Q. Tao, G. Wu, F. Wang, and J. Wang, "Posterior probability support vector machines for unbalanced data," *IEEE Trans. Neural Netw.*, vol. 16, no. 6, pp. 1561–1573, Jun. 2015.
[60] A. Alzubaidi and J. Kalita, "Authentication of smartphone users using behavioral biometrics," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 1998–2026, 3rd Quart., 2016.
[61] A. S. Shamili, C. Bauckhage, and T. Alpcan, "Malware detection on mobile devices using distributed machine learning," in *Proc. 20th Int. Conf. Pattern Recognit.*, Aug. 2010, pp. 4348–4351.
[62] J. A. K. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Process. Lett.*, vol. 9, no. 3, pp. 293–300, Jun. 1999.
[63] N. Eagle and A. Pentland, "Reality mining: Sensing complex social systems," *Pers. Ubiquitous Comput.*, vol. 10, no. 4, pp. 255–268, 2005.
[64] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices (SPSM)*, 2011, pp. 15–26.
[65] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern Recognit.*, vol. 36, no. 2, pp. 451–461, Feb. 2003.
[66] A. Shabtai, U. Kanonov, and Y. Elovici, "Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method," *J. Syst. Softw.*, vol. 83, no. 8, pp. 1524–1537, Aug. 2010.
[67] A. Shabtai, Y. Fledel, Y. Elovici, and Y. Shahar, "Using the KBTA method for inferring computer and network security alerts from time-stamped, raw system metrics," *J. Comput. Virol.*, vol. 6, no. 3, pp. 239–259, Aug. 2010.
[68] T. Alpcan, C. Bauckhage, and A.-D. Schmidt, "A probabilistic diffusion scheme for anomaly detection on smartphones," in *Proc. IFIP Int. Workshop Inf. Secur. Theory Practices*. Berlin, Germany: Springer, 2010, pp. 31–46.
[69] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, no. 4, pp. 225–231, 1973.
[70] L. Rüschendorf and S. T. Rachev, "A characterization of random variables with minimum L^2-distance," *J. Multivariate Anal.*, vol. 32, no. 1, pp. 48–54, Jan. 1990.
[71] R. De Maesschalck, D. Jouan-Rimbaud, and D. L. Massart, "The Mahalanobis distance," *Chemometrics Intell. Lab. Syst.*, vol. 50, no. 1, pp. 1–18, 2000.
[72] T. Kohonen, "Exploration of very large databases by self-organizing maps," in *Proc. Int. Conf. Neural Netw. (ICNN)*, vol. 1, 1997, pp. PL1–PL6.
[73] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Comput. Secur.*, vol. 37, pp. 215–227, Sep. 2013.
[74] M. S. Alam and S. T. Vuong, "Random forest classification for detecting Android malware," in *Proc. IEEE Int. Conf. Green Comput. Commun., IEEE Internet Things IEEE Cyber, Phys. Social Comput.*, Aug. 2013, pp. 663–669.
[75] M. Pal, "Random forest classifier for remote sensing classification," *Int. J. Remote Sens.*, vol. 26, no. 1, pp. 217–222, 2005.
[76] B. Amos. (May 2013). *Android Antimalware*. Accessed: Oct. 20, 2021. [Online]. Available: https://github.com/VT-Magnum-Research/antimalware
[77] D. Damopoulos, G. Kambourakis, and G. Portokalidis, "The best of both worlds: A framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones," in *Proc. 7th Eur. Workshop Syst. Secur. (EuroSec)*, 2014, pp. 1–6.
[78] K. Riad and L. Ke, "RoughDroid: Operative scheme for functional Android malware detection," *Secur. Commun. Netw.*, vol. 2018, pp. 1–10, Sep. 2018.
[79] A. Shabtai and Y. Elovici, "Applying behavioral detection on Android-based devices," in *Proc. Int. Conf. Mobile Wireless Middleware, Operating Syst., Appl.* Berlin, Germany: Springer, 2010, pp. 235–249.
[80] C. Stachniss, G. Grisetti, and W. Burgard, "Information gain-based exploration using rao-blackwellized particle filters," in *Robotics: Science and Systems*, vol. 2. Cambridge, MA, USA: Massachusetts Institute of Technology, 2005, pp. 65–72.
[81] Q. Gu, Z. Li, and J. Han, "Generalized Fisher score for feature selection," 2012, *arXiv:1202.3725*.
[82] F. Y. Hsieh, D. A. Bloch, and M. D. Larsen, "A simple method of sample size calculation for linear and logistic regression," *Statist. Med.*, vol. 17, no. 14, pp. 1623–1634, Jul. 1998.
[83] T. Penna and H. Herrmann, "Broad histogram method," 1996, *arXiv:cond-mat/9610041*.
[84] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.
[85] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Mach. Learn.*, vol. 29, no. 2, pp. 131–163, Nov. 1997.
[86] Q. Wang, G. M. Garrity, J. M. Tiedje, and J. R. Cole, "Naïve Bayesian classifier for rapid assignment of rRNA sequences into the new bacterial taxonomy," *Appl. Environ. Microbiol.*, vol. 73, no. 16, pp. 5261–5267, Aug. 2007.
[87] M. Curti, A. Merlo, M. Migliardi, and S. Schiappacasse, "Towards energy-aware intrusion detection systems on mobile devices," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2013, pp. 289–296.
[88] M. Damshenas, A. Dehghantanha, K.-K. R. Choo, and R. Mahmud, "M0Droid: An Android behavioral-based malware detection model," *J. Inf. Privacy Secur.*, vol. 11, no. 3, pp. 141–157, Sep. 2015.
[89] A. Merlo, M. Migliardi, and P. Fontanelli, "On energy-based profiling of malware in Android," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2014, pp. 535–542.
[90] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardw., Softw. Codesign Syst. Synth. (CODES/ISSS)*, 2010, pp. 105–114.
[91] F. Yuan, L. Zhai, Y. Cao, and L. Guo, "Research of intrusion detection system on Android," in *Proc. IEEE 9th World Congr. Services*, Jun. 2013, pp. 312–316.
[92] R. S. R. James, A. Albasir, K. Naik, M. Y. Dabbagh, P. Dash, M. Zamani, and N. Goel, "Detection of anomalous behavior of smartphones using signal processing and machine learning techniques," in *Proc. IEEE 28th Annu. Int. Symp. Pers., Indoor, Mobile Radio Commun. (PIMRC)*, Oct. 2017, pp. 1–7.
[93] X. Kou and Q. Wen, "Intrusion detection model based on Android," in *Proc. 4th IEEE Int. Conf. Broadband Netw. Multimedia Technol.*, Oct. 2011, pp. 624–628.

[94] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile protection for smartphones," in *Proc. 26th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2010, pp. 347–356.
[95] T.-E. Wei, C.-H. Mao, A. B. Jeng, H.-M. Lee, H.-T. Wang, and D.-J. Wu, "Android malware detection via a latent network behavior analysis," in *Proc. IEEE 11th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Jun. 2012, pp. 1251–1258.
[96] S. Kumar, A. Viinikainen, and T. Hamalainen, "Evaluation of ensemble machine learning methods in mobile threat detection," in *Proc. 12th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, 2017, pp. 261–268.
[97] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. V. D. Veen, and C. Platzer, "ANDRUBIS–1,000,000 apps later: A view on current Android malware behaviors," in *Proc. 3rd Int. Workshop Building Anal. Datasets Gathering Exper. Returns Secur. (BADGERS)*, Sep. 2014, pp. 3–17.
[98] D. Oktavianto and I. Muhardianto, *Cuckoo Malware Analysis*. Birmingham, U.K.: Packt, 2013.
[99] B. Trammell and E. Boschi, *Bidirectional Flow Export Using IP Flow Information Export (IPFIX)*, document RFC 5103, Jan. 2008.
[100] A. Rajput, R. P. Aharwal, M. Dubey, S. Saxena, and M. Raghuvanshi, "J48 and JRIP rules for e-governance data," *Int. J. Comput. Sci. Secur.*, vol. 5, no. 2, p. 201, 2011.
[101] V. Veeralakshmi and D. Ramyachitra, "Ripple Down Rule learner (RIDOR) classifier for IRIS dataset," *Issues*, vol. 1, no. 1, pp. 79–85, 2015.
[102] E. Frank and I. H. Witten, "Generating accurate rule sets without global optimization," in *Proc. 15th Int. Conf. Mach. Learn.*, J. Shavlik, Ed. San Mateo, CA, USA: Morgan Kaufmann, 1998, pp. 144–151.
[103] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Comput.*, vol. 20, no. 1, pp. 343–357, 2016.
[104] E. A. Zanaty, "Support vector machines (SVMs) versus multilayer perception (MLP) in data classification," *Egyptian Informat. J.*, vol. 13, no. 3, pp. 177–183, Nov. 2012.
[105] P. Soucy and G. W. Mineau, "A simple KNN algorithm for text categorization," in *Proc. IEEE Int. Conf. Data Mining*, Nov. 2001, pp. 647–648.
[106] J. Ribeiro, F. B. Saghezchi, G. Mantas, J. Rodriguez, S. J. Shepherd, and R. A. Abd-Alhameed, "An autonomous host-based intrusion detection system for Android mobile devices," *Mobile Netw. Appl.*, vol. 25, no. 1, pp. 164–172, Feb. 2020.
[107] S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "DroidLight: Lightweight anomaly-based intrusion detection system for smartphone devices," in *Proc. 21st Int. Conf. Distrib. Comput. Netw.*, Jan. 2020, pp. 1–10.
[108] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.
[109] G. J. McLachlan and T. Krishnan, *The EM Algorithm and Extensions*, vol. 382. Hoboken, NJ, USA: Wiley, 2007.
[110] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "DroidDet: Effective and robust detection of Android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, Jan. 2018.
[111] A. A. Hakim, A. Erwin, K. I. Eng, M. Galinium, and W. Muliady, "Automated document classification for news article in Bahasa Indonesia based on term frequency inverse document frequency (TF-IDF) approach," in *Proc. 6th Int. Conf. Inf. Technol. Electr. Eng. (ICITEE)*, Oct. 2014, pp. 1–4.
[112] S. Tata and J. M. Patel, "Estimating the selectivity of $tf-idf$ based cosine similarity predicates," *ACM SIGMOD Rec.*, vol. 36, no. 2, pp. 7–12, Jun. 2007.
[113] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics Intell. Lab. Syst.*, vol. 2, nos. 1–3, pp. 37–52, 1987.
[114] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining api-level features for robust malware detection in Android," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2013, pp. 86–103.
[115] J. R. Quinlan, *C4. 5: Programs for Machine Learning*. Amsterdam, The Netherlands: Elsevier, 2014.
[116] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided Android malware classification," *Comput. Elect. Eng.*, vol. 61, pp. 266–274, Jul. 2017.
[117] Y. Zhang, R. Jin, and Z. Zhou, "Understanding bag-of-words model: A statistical framework," *Int. J. Mach. Learn. Cybern.*, vol. 1, nos. 1–4, pp. 43–52, Dec. 2010.
[118] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018.
[119] S. Chen and X. Yang, "Alternative linear discriminant classifier," *Pattern Recognit.*, vol. 37, no. 7, pp. 1545–1547, Jul. 2004.
[120] S. Srivastava, M. R. Gupta, and B. A. Frigyik, "Bayesian quadratic discriminant analysis," *J. Mach. Learn. Res.*, vol. 8, pp. 1277–1305, Jun. 2007.
[121] D. K. Wedding and K. J. Cios, "Certainty factors versus Parzen windows as reliability measures in RBF networks," *Neurocomputing*, vol. 19, nos. 1–3, pp. 151–165, Apr. 1998.
[122] Y.-F. Lu, C.-F. Kuo, H.-Y. Chen, C.-W. Chen, and S.-C. Chou, "A SVM-based malware detection mechanism for Android devices," in *Proc. Int. Conf. Syst. Sci. Eng. (ICSSE)*, Jun. 2018, pp. 1–6.
[123] C. C. Chang and C. J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, 2011.
[124] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new Android malware detection approach using Bayesian classification," in *Proc. IEEE 27th Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Mar. 2013, pp. 121–128.
[125] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy Android malware detection using ensemble learning," *IET Inf. Secur.*, vol. 9, no. 6, pp. 313–320, 2015.
[126] A. Merlo, M. Migliardi, and P. Fontanelli, "Measuring and estimating power consumption in Android to support energy-based intrusion detection," *J. Comput. Secur.*, vol. 23, no. 5, pp. 611–637, 2015.
[127] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Comput. Secur.*, vol. 43, no. 6, pp. 1–18, 2014.
[128] G. A. Seber and A. J. Lee, *Linear Regression Analysis*, vol. 329. Hoboken, NJ, USA: Wiley, 2012.
[129] O. Stegle, S. V. Fallert, D. J. C. MacKay, and S. Brage, "Gaussian process robust regression for noisy heart rate data," *IEEE Trans. Biomed. Eng.*, vol. 55, no. 9, pp. 2143–2151, Sep. 2008.
[130] R. E. Barlow and H. D. Brunk, "The isotonic regression problem and its dual," *J. Amer. Stat. Assoc.*, vol. 67, no. 337, pp. 140–147, Mar. 1972.
[131] W.-Y. Loh, "Fifty years of classification and regression trees," *Int. Stat. Rev.*, vol. 82, no. 3, pp. 329–348, 2014.
[132] K. A. Talha, D. I. Alper, and C. Aydin, "APK Auditor: Permission-based Android malware detection system," *Digital Invest.*, vol. 13, pp. 1–14, Jun. 2015.
[133] H. Rathore, S. K. Sahay, P. Nikam, and M. Sewak, "Robust Android malware detection system against adversarial attacks using Q-learning," *Inf. Syst. Frontiers*, vol. 23, no. 4, pp. 1–16, 2020.
[134] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.
[135] Z. Wang, C. Li, Z. Yuan, Y. Guan, and Y. Xue, "DroidChain: A novel Android malware detection method based on behavior chains," *Pervas. Mobile Comput.*, vol. 32, pp. 3–14, Oct. 2016.
[136] A. Desnos and G. Gueguen. (2013). *Androguard-Reverse Engineering, Malware and Goodware Analysis of Android Applications*. [Online]. Available: htpp://www.code.google.com/p/androguard
[137] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Comput. Secur.*, vol. 49, pp. 255–273, Mar. 2015.
[138] J. Ching, "Nondeterministic parallel control-flow/definition-use nets and their applications," in *Parallel Computing: Trends and Applications*. Amsterdam, The Netherlands: Elsevier, 1994, pp. 589–592.
[139] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "Pindroid: A novel Android malware detection system using ensemble learning methods," *Comput. Secur.*, vol. 68, pp. 36–46, Jul. 2017.
[140] G.-Y. Wang, H. Yu, and D.-C. Yang, "Decision table reduction based on conditional information entropy," *Chin. J. Comput.*, vol. 25, no. 7, pp. 759–766, 2002.
[141] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," Microsoft Res., Tech. Rep. MSR-TR-98-14, 1998.
[142] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for Android malware detection," in *Proc. 7th Int. Conf. Comput. Intell. Secur.*, Dec. 2011, pp. 1011–1015.

[143] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "PBMDS: A behavior-based malware detection system for cellphone devices," in *Proc. 3rd ACM Conf. Wireless Netw. Secur. (WiSec)*, 2010, pp. 37–48.
[144] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden Markov model: Analysis and applications," *Mach. Learn.*, vol. 32, no. 1, pp. 41–62, 1998.
[145] S. Shamshirband and A. T. Chronopoulos, "A new malware detection system using a high performance-ELM method," in *Proc. 23rd Int. Database Appl. Eng. Symp. (IDEAS)*, 2019, pp. 1–10.
[146] K. Polat and S. Güneş, "A new feature selection method on classification of medical datasets: Kernel F-score feature selection," *Expert Syst. Appl.*, vol. 36, no. 7, pp. 10367–10373, Sep. 2009.
[147] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version)," *ACM Trans. Privacy Secur.*, vol. 22, no. 2, pp. 1–34, 2019.
[148] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with API semantics to detect evolved Android malware," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 757–770.
[149] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "DroidEvolver: Self-evolving Android malware detection system," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Jun. 2019, pp. 47–62.
[150] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," in *Proc. IEEE Secur. Privacy Workshops (SPW)*. San Francisco, CA, USA: Springer, 2019, pp. 8–14.
[151] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 729–746.
[152] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: Fast and accurate classification of obfuscated Android malware," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 309–320.
[153] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 426–436.
[154] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DynaLog: An automated dynamic analysis framework for characterizing Android applications," in *Proc. Int. Conf. Cyber Secur. Protection Digit. Services (Cyber Secur.)*, 2016, pp. 1–8.
[155] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "EMULATOR vs REAL PHONE: Android malware detection using machine learning," in *Proc. 3rd ACM Int. Workshop Secur. Privacy Analytics*, Mar. 2017, pp. 65–72.
[156] S. Y. Yerima, M. K. Alzaylaee, and S. Sezer, "Machine learning-based dynamic analysis of Android apps with improved code coverage," *EURASIP J. Inf. Secur.*, vol. 2019, no. 1, p. 4, Dec. 2019.
[157] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DL-droid: Deep learning based Android malware detection using real devices," *Comput. Secur.*, vol. 89, Feb. 2020, Art. no. 101663.
[158] J. M. Vidal, M. A. S. Monge, and L. J. García-Villalba, "A novel pattern recognition system for detecting Android malware by analyzing suspicious boot sequences," *Knowl.-Based Syst.*, vol. 150, pp. 198–217, Jun. 2018.
[159] A. Apvrille, "The evolution of mobile malware," *Comput. Fraud Secur.*, vol. 2014, no. 8, pp. 18–20, Aug. 2014.
[160] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Molecular Biol.*, vol. 48, no. 3, pp. 443–453, 1970.
[161] W.-C. Wu and S.-H. Hung, "DroidDolphin: A dynamic Android malware detection framework using big data and machine learning," in *Proc. Conf. Res. Adapt. Convergent Syst. (RACS)*, 2014, pp. 247–252.
[162] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *J. Comput. Linguistics*, vol. 18, no. 4, pp. 467–479, 1992.
[163] H. Cai and B. G. Ryder, "Understanding Android application programming and security: A dynamic study," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 364–375.
[164] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*, vol. 1, May 2015, pp. 77–88.
[165] H. Cai and J. Jenkins, "Towards sustainable Android malware detection," in *Proc. 40th Int. Conf. Softw. Eng., Companion*, May 2018, pp. 350–351.
[166] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, 2012, pp. 281–294.
[167] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "DroidScribe: Classifying Android malware based on runtime behavior," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2016, pp. 252–261.
[168] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.
[169] V. Vovk, A. Gammerman, and G. Shafer, *Algorithmic Learning in a Random World*. Berlin, Germany: Springer, 2005.
[170] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang, "Design and implementation of an Android host-based intrusion prevention system," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, Dec. 2014, pp. 226–235.
[171] H. Cai, N. Meng, B. G. Ryder, and D. Yao, "DroidCat: Effective Android malware detection and categorization via app-level profiling," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 6, pp. 1455–1470, Nov. 2018.
[172] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 2, pp. 1–28, Apr. 2020.
[173] L. Chen, S. Hou, Y. Ye, and S. Xu, "DroidEye: Fortifying security of learning-based classifier against adversarial Android malware attacks," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining (ASONAM)*, Aug. 2018, pp. 782–789.
[174] L. Chen and Y. Ye, "Sec*MD*: Make machine learning more secure against adversarial malware attacks," in *Proc. Australas. Joint Conf. Artif. Intell.* Cham, Switzerland: Springer, 2017, pp. 76–89.
[175] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014, *arXiv:1412.6572*.
[176] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Security*, vol. 15, no. 1, pp. 987–1001, Jul. 2019.
[177] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google play," in *Proc. ACM Int. Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, 2014, pp. 221–233.
[178] E. B. Karbab and M. Debbabi, "MalDy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports," *Digit. Invest.*, vol. 28, pp. S77–S87, Apr. 2019.
[179] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digit. Invest.*, vol. 24, pp. S48–S59, Mar. 2018.
[180] Y. Zhang, W. Ren, T. Zhu, and Y. Ren, "SaaS: A situational awareness and analysis system for massive Android malware detection," *Future Gener. Comput. Syst.*, vol. 95, pp. 548–559, Jun. 2019.
[181] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro, "A family of droids-Android malware detection via behavioral modeling: Static vs dynamic analysis," 2018, *arXiv:1803.03448*.
[182] A. developer. (2017). *Inspect Trace Logs With Traceview*. Accessed: Oct. 20, 2021. [Online]. Available: https://developer.android.com/studio/profile/traceview.html
[183] C. J. Geyer, "Practical Markov chain Monte Carlo," *Stat. Sci.*, vol. 7, no. 4, pp. 473–483, Nov. 1992.
[184] F. Tong and Z. Yan, "A hybrid approach of mobile malware detection in Android," *J. Parallel Distrib. Comput.*, vol. 103, pp. 22–31, May 2017.
[185] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. COMPSAC*, vol. 2, Jul. 2015, pp. 422–433.
[186] S. Rane, W. Sun, and A. Vetro, "Privacy-preserving approximation of L1 distance for multimedia applications," in *Proc. IEEE Int. Conf. Multimedia Expo*, Jul. 2010, pp. 492–497.
[187] M. Zheng, M. Sun, and J. C. S. Lui, "DroidRay: A security evaluation system for customized Android firmwares," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur.*, Jun. 2014, pp. 471–482.
[188] A. T. Kabakus and I. A. Dogru, "An in-depth analysis of Android malware using hybrid techniques," *Digit. Invest.*, vol. 24, pp. 25–33, Mar. 2018.
[189] X. Fu and H. Cai, "On the deterioration of learning-based malware detectors for Android," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion Proc. (ICSE-Companion)*, May 2019, pp. 272–273.
[190] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *J. Comput. Virol. Hacking Techn.*, vol. 11, no. 1, pp. 9–17, 2015.

[191] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of Android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 1–29, Jan. 2018.
[192] R. Vinayakumar, K. P. Soman, and P. Poornachandran, "Deep Android malware detection and classification," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2017, pp. 1677–1683.
[193] R. Vinayakumar, K. P. Soman, P. Poornachandran, and S. S. Kumar, "Detecting Android malware using long short-term memory (LSTM)," *J. Intell. Fuzzy Syst.*, vol. 34, no. 3, pp. 1277–1288, Mar. 2018.
[194] S. Hou, Y. Fan, Y. Zhang, Y. Ye, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "αCyber: Enhancing robustness of Android malware detection system against adversarial attacks on heterogeneous graph based model," in *Proc. 28th ACM Int. Conf. Inf. Knowl. Manage.*, Nov. 2019, pp. 609–618.
[195] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic Android malware detection at scale," in *Proc. 9th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, 2013, pp. 1666–1671.
[196] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—A review of applications in the atmospheric sciences," *Atmos. Environ.*, vol. 32, nos. 14–15, pp. 2627–2636, 1998.
[197] P. S. Chen, S.-C. Lin, and C.-H. Sun, "Simple and effective method for detecting abnormal internet behaviors of mobile devices," *Inf. Sci.*, vol. 321, pp. 193–204, Nov. 2015.
[198] H. Cuntz, F. Forstner, A. Borst, and M. Häusser, "One rule to grow them all: A general theory of neuronal branching and its practical application," *PLoS Comput. Biol.*, vol. 6, no. 8, Aug. 2010, Art. no. e1000877.
[199] J. J. Gómez-Hernández and X.-H. Wen, "To be or not to be multi-Gaussian? A reflection on stochastic hydrogeology," *Adv. Water Resour.*, vol. 21, no. 1, pp. 47–61, Feb. 1998.
[200] M. Anderson, T. Adalı, and X.-L. Li, "Joint blind source separation with multivariate Gaussian model: Algorithms and performance analysis," *IEEE Trans. Signal Process.*, vol. 60, no. 4, pp. 1672–1683, Apr. 2011.
[201] J. Sahs and L. Khan, "A machine learning approach to Android malware detection," in *Proc. Eur. Intell. Secur. Informat. Conf.*, Aug. 2012, pp. 141–147.
[202] Z. Aung and W. Zaw, "Permission-based Android malware detection," *Int. J. Sci. Technol. Res.*, vol. 2, no. 3, pp. 228–234, 2013
[203] P.-E. Danielsson, "Euclidean distance mapping," *Comput. Graph. Image Process.*, vol. 14, no. 3, pp. 227–248, 1980.
[204] R. J. Lewis, "An introduction to classification and regression tree (CART) analysis," in *Proc. Annu. Meeting Soc. Academic Emergency Med.*, San Francisco, CA, USA, vol. 14, 2000, pp. 1–14.
[205] D. Li, Z. Wang, and Y. Xue, "Fine-grained Android malware detection based on deep learning," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, May 2018, pp. 1–2.
[206] M. Z. Mas'ud, S. Sahib, M. F. Abdollah, S. R. Selamat, and R. Yusof, "Analysis of features selection and machine learning classifier in Android malware detection," in *Proc. Int. Conf. Inf. Sci. Appl. (ICISA)*, May 2014, pp. 1–5.
[207] R. Graf, L. A. Kaplan, and R. King, "Neural network-based technique for Android smartphone applications classification," in *Proc. 11th Int. Conf. Cyber Conflict (CyCon)*, May 2019, pp. 1–17.
[208] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert System*. Reading, MA, USA: Addison-Wesley, 1983.
[209] A. Salman, I. H. Elhajj, A. Chehab, and A. Kayssi, "DAIDS: An architecture for modular mobile IDS," in *Proc. 28th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, May 2014, pp. 328–333.
[210] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral analysis of Android applications using automated instrumentation," in *Proc. IEEE 7th Int. Conf. Softw. Secur. Rel. Companion*, Jun. 2013, pp. 182–187.

ABDULAZIZ ALZUBAIDI received the bachelor's degree in computer science from the University College, King Abdulaziz University, Saudi Arabia, in 2001, the Master of Science degree from Jordan University, and the Ph.D. degree from the University of Colorado at Colorado Springs, USA, in 2017. His research interests include intrusion detection, computer vision, human activity recognition, cyber-security awareness's, and smart-devices security.