Hindawi
Security and Communication Networks
Volume 2023, Article ID 9879467, 1 page
https://doi.org/10.1155/2023/9879467

Retraction

Retracted: A Comprehensive Review of Android Security: Threats, Vulnerabilities, Malware Detection, and Analysis

Security and Communication Networks

Received 10 October 2023; Accepted 10 October 2023; Published 11 October 2023

This article has been retracted by Hindawi following an investigation undertaken by the publisher [1]. This investigation has uncovered evidence of one or more of the following indicators of systematic manipulation of the publication process:

(1) Discrepancies in scope

(2) Discrepancies in the description of the research reported

(3) Discrepancies between the availability of data and the research described

(4) Inappropriate citations

(5) Incoherent, meaningless and/or irrelevant content included in the article

(6) Peer-review manipulation

The presence of these indicators undermines our confidence in the integrity of the article's content and we cannot, therefore, vouch for its reliability. Please note that this notice is intended solely to alert readers that the content of this article is unreliable. We have not investigated whether authors were aware of or involved in the systematic manipulation of the publication process.

Wiley and Hindawi regrets that the usual quality checks did not identify these issues before publication and have since put additional measures in place to safeguard research integrity.

We wish to credit our own Research Integrity and Research Publishing teams and anonymous and named external researchers and research integrity experts for contributing to this investigation.

The corresponding author, as the representative of all authors, has been given the opportunity to register their agreement or disagreement to this retraction. We have kept a record of any response received.

References

[1] S. Acharya, U. Rawat, and R. Bhatnagar, "A Comprehensive Review of Android Security: Threats, Vulnerabilities, Malware Detection, and Analysis," *Security and Communication Networks*, vol. 2022, Article ID 7775917, 34 pages, 2022.

Hindawi
Security and Communication Networks
Volume 2022, Article ID 7775917, 34 pages
https://doi.org/10.1155/2022/7775917

Review Article

A Comprehensive Review of Android Security: Threats, Vulnerabilities, Malware Detection, and Analysis

Saket Acharya, Umashankar Rawat, and Roheet Bhatnagar

Manipal University Jaipur, Jaipur, Rajasthan, India

Correspondence should be addressed to Umashankar Rawat; umashankar.rawat@jaipur.manipal.edu

Received 2 April 2022; Revised 9 May 2022; Accepted 23 May 2022; Published 29 June 2022

Academic Editor: Bharat Bhushan

The popularity and open-source nature of Android devices have resulted in a dramatic growth of Android malware. Malware developers are also able to evade the detection methods, reducing the efficiency of malware detection techniques. It is hence desirable that security researchers and experts come up with novel and more efficient methods to analyze existing and zero-day Android malware. Most of the researchers have focused on Android system security. However, to examine Android security, with a specific focus on malware development, investigation of malware prevention techniques and already known malware detection techniques needs a broad inclusion. To overcome the research gaps, this paper provides a broad review of current Android security concerns, security implementation enhancements, significant malware detected during 2017–2021, and stealth procedures used by the malware developers along with the current Android malware detection techniques. A comparative analysis is presented between this article and similar recent survey articles to fill the existing research gaps. In the end, a three-phase model is proposed to efficiently identify and characterize Android malware. In the first phase, a lightweight deep transfer learning approach is used to classify Android applications into benign and malicious. In the second phase, the malicious applications are executed in a virtual emulator to reduce the number of false positives. Finally, the malicious applications having the same characteristic ratio are grouped into their corresponding families using the topic modelling approach. The proposed model can efficiently detect, characterize, and provide a familial classification of Android malware with a good accuracy rate.

1. Introduction

According to Gartner smartphone sale reports, Android mobile phones have seized more than 82% of the market share during 2020–2021, leaving all its contenders, namely, iOS, Windows operating system, and Blackberry, a far way behind [1]. The main reason behind Android's growth is its open-source nature and a huge amount of freely available applications [2].

Google Play (GP), the official Android application store, has more than a billion applications with limitless downloads every day [3]. Initially, Google Play used Bouncer [4] to monitor and secure the play store from malware threats but it failed to identify zero-day malware among published applications [5]. The malware creators exploited this vulnerability and extracted the private data of the users. Nowadays, Google Play uses a security feature called Google Play Protect [6, 7] which provides an option to prevent the device from installing unknown and vulnerable apps. But, this option can be disabled manually at the time of the installation process because the open-source nature of Android gives license to the users to install any type of applications and eventually compromises the security of the Android system [8, 9]. The quantity of malware applications uploaded on VirusTotal (https://www.virustotal.com) is rising drastically. Exponentially increasing malware applications have forced the anti-malware business to identify the strong and effective strategies appropriate for on-device recognition inside the current limitations. The present anti-malware solutions use signature-based malware detection because of its implementation proficiency and simplicity [10]. Signature-based detection can be avoided by reshaping the source code, requiring a new signature for every new malware variant [11]. Due to this, the anti-malware clients

also need to regularly update their signature database. To overcome the problems of constrained processing capability and restricted battery availability, malware detection techniques based on the cloud were introduced for analysis and detection [12]. Manual examination and malware definition extraction need sufficient time and ability. These methodologies can also yield false negatives (FNs) while generating a signature for the variations of known malware families. Since malware growth is increasing rapidly, there is an urgent requirement of malware analysis techniques to efficiently examine novel malware variants.

1.1. Background. Android security issues always remain a major concern for security analysts and researchers. Novel malware attacks are increasing daily. The scholarly world and industry analysts have proposed the following criterion to dissect and recognize the Android malware:

(1) Aim of the proposed solution can be categorized into vulnerability assessment or malicious behavior detection. Vulnerability assessment solutions identify the loopholes which can be exploited by an attacker from a remote location. Solutions based on malicious behavior detection focus on analyzing the malicious behavior and preventing the malicious apps to get installed on the device.

(2) Techniques to achieve the above goals can be divided into the following approaches:

Static approach focuses on examining the static part of the application without actually executing the application. Control flow analysis, data flow analysis, and topic modelling approach are the well-known examples of static approach [13, 14].
In the dynamic approach, the Android apps are emulated in a virtual sandbox environment and their activities are monitored to analyze malicious behavior [15–17]. This approach examines runtime malware actions but it generates an extra computational complexity.
To reduce the complexity and handle new malware variants, the pipeline of API calls and their recurrence of appearance of API calls are passed into machine learning and deep learning algorithms. Machine learning-based approaches can efficiently detect Android malware, but they require detailed knowledge of feature classification and feature extraction. Deep Learning approaches automatically extract useful features, but they require high computational power to train the dataset [18–21].

(3) Implementation of the above-mentioned approaches to identify their advantages and limitations.

Existing Android security surveys are based on popular mobile OS platforms [13, 22]. However, this survey paper emphasizes on Android system by covering fine-grained aspects of device security. La Polla et al. [23] reviewed the smartphone security hazards and their solutions. The authors analyzed the restricted use of explicit Android features influencing the overall device security. They classified the malware based on their attack objectives, malicious conveyance, infection, and privilege escalation. However, the techniques to handle zero-day malware are not focused. Suarez Tangil et al. extended the work of La Polla et al. and analyzed the security of Android devices in terms of software vulnerabilities. The authors worked on application-level vulnerabilities, but system-level security issues, such as system calls-based attacks and privilege escalation attacks remained a major security concern. Faruki et al. [24] provided broad research on the Android malware based on commercial anti-malware industry terminology. The authors demonstrated malware infection coverage from 2010 to 2016. The authors also presented some malware detection schemes; however, the techniques like machine learning and deep learning were not focused on.

Cai [25] undertook research to develop a systematic environment for mining the mobile software ecosystem on a constant basis. The author focused on behavioral evolution and conducted a large-scale ecosystem characterization research. Furthermore, an ecological interaction was explored between three attributes: user app platform, mobile platform, and app users. The outcomes ensure long-term app development and security.

Cai et al. [26] presented a research paper on Android application execution approach. The authors looked at how malware behaved in Android apps in terms of execution pathways, structures, methodological scopes, and callbacks. In terms of the security platform, they observed the app execution structure. Furthermore, the scientists tracked the ICCs and methods of over 30,000 apps from 2010 to 2017, including 15,451 benign and 15,183 malicious apps. To identify the differences between these apps, the behavioral structure and similarities were determined.

Liu et al. [27] introduced MR-Droid, a MapReduce-based computing platform for Android inter-app ICC analysis that is accurate and scalable. To create a large-scale ICC graph, MR-Droid gathers data flow properties from several communicating apps. To provide exact alerts and prioritize risk assessments, the authors used the ICC graph to give context for inter-app conversations. This technique necessitates processing a huge number of app-pairs quickly. Extensive tests on 11,996 apps from 24 categories (13 million pairs) showed that their risk prioritizing scheme is successful.

Cai et al. [28] demonstrated DroidCat, a new dynamic and robust Android app classification system based on dynamic method calls and ICC Intents. Without relying on system calls or permissions, the suggested technique efficiently handled reflection. Furthermore, when compared to other similar state-of-the-art static and dynamic malware detection algorithms, the technique was more resilient. In addition, the examination employed 34,343 apps from various sources over the course of nine years, and a performance measure was generated. When compared to two state-of-the- art methodologies, the authors attained a 97% F1 score accuracy.

Fu and Cai [29] investigated the degrading issues in Android malware detectors. The authors looked at four

cutting-edge detectors and found that the current solutions' performance declines over time. Furthermore, the authors developed a novel method based on a longitudinal characterization study of application runtime characteristics. To examine the deteriorating problem, a comparison was made between the proposed strategy and four state-of-the-art techniques.

Ficco [30] proposed an approach based on ensemble detection. The author utilized a blend of generic and specialized detectors during the analysis process to enhance the detection randomness and to improve the overall detection rate. Moreover, an alpha-count mechanism is also presented to differentiate the speed of various detectors. This mechanism provides the observation time window length which can affect the detection accuracy.

D'Angelo et al. [31] proposed an approach based on the exploitation of API transitions in the call sequence. The extraction of a subsequence of API calls resulted in a malware classification resistant to evasion techniques. The authors compared the detectors using Markov chain and call sequence algorithms. The study outcomes outperformed various malware detection techniques.

Malware detection techniques based on deep learning require a large amount of labeled data points to identify malicious threats with optimum accuracy. In most cases, the size of the dataset for detecting new malware threats is not always large, and to collect a new dataset, the search time also increases. Moreover, to identify a new malware threat, the deep learning models need to be trained again for a new dataset from scratch, which is not only resource-consuming but also time-consuming. One efficient solution to overcome the issue of high computational complexity and model retraining is to use deep transfer learning technique. The major aim of utilizing the transfer learning approach in our study is to reduce the computational complexity by transferring well-known feature sets from a trained base model to a destination model with very less training data.

1.2. Motivation. The primary motivation of this review article is to present detailed aspects of Android security issues in terms of device vulnerabilities and Android malware detection techniques. Security issues lead to several loopholes which become an entry gate for the attackers to exploit the device. The scholarly world has proposed several detection schemes to identify Android malware, but the malware developers remain successful in attacking the device with more novel malware variants. The research gaps which are lacking in the existing surveys are covered in this article to help researchers to identify and detect fine-grained Android security loopholes. Moreover, the gaps in Android malware detection and analysis techniques are identified and a lightweight and efficient model is proposed which can fill the current gaps and also provides family-wise classification of novel malware variants.

1.3. Contribution. This paper provides a broad survey of Android architecture, Android security enhancements, malware detection events that occurred during 2017–2021, and quality and constraints of outstanding malware examination and recognition approaches. Specifically, this paper comprehensively covers malware analysis techniques and also the stealth methods used by malware developers to escape the detection by producing variations of the previously known malware. The systematic comparative summary of related Android malware detection surveys is presented in Table 1.

According to Table 1, the current study differs from previous surveys done in the related area as it provides collective information of Android malware detection approaches and techniques. For example, the authors in [33, 34, 39] presented a survey on Android malware detection which is purely based on machine learning methods. The authors in [35] focused only on static analysis methods, and dynamic analysis techniques are not covered. Similarly, familial clustering is not very much focused, and transfer learning technique is not covered at all in the related surveys as per Table 1.

The rest of the survey article is arranged as follows.

Section 2 summarizes the Android application design and security components required to debilitate the attack areas. Section 3 discusses the Android security flaws regardless of existing implementations examined in Section 2 and important security upgrades in the Android versions to handle the specified flaws. Section 4 presents the timeline of important Android malware variants developed during 2017–2021 and different stealth techniques used by the Android malware. Sections 5 and 6 classify the conspicuous evaluation, investigation, and identification strategies along with their implementation solutions, respectively, and compare the most commonly used tools proposed by the scholarly world and anti-malware industry as indicated by their usefulness canvassed in Section 4. In Section 7, a model is proposed to detect and characterize Android malware using a blend of deep transfer learning and topic modelling approaches. In the end, Section 8 concludes this paper and discusses further work.

2. Android Architecture

Android is being drafted by Google as an open-source project. It comprises the original equipment manufacturers (OEMs), chip producers, data carriers, and app developers. Android applications are compiled in Java but the original libraries are made in C/C++ language. The Android architecture is shown in Figure 1. Android is created over the Linux kernel because of its productive driver model, efficient process and memory management, and system administration support for central services. Different Android devices use different CPUs and instruction sets. Every mix of CPU and instruction set has its own application binary interface (ABI) [43]. These ABIs contain information on the instruction sets, the endianness of memory stores, data passing conventions, the format of shared libraries, and so on. The ABIs and their corresponding instruction sets are shown in Table 2.

Android user applications can be divided into native and third-party applications. The application framework consists of several managers as shown in Figure 1.

Table 1: Comparison of current reviews having extending scope with this study (✓ = with content, ★ = with less content, and ✕ = with no content).

Survey authors	Year	Ref.	Static analysis approach	Dynamic analysis approach	Machine learning method	Deep learning method	Training speedup	Transfer learning technique	Familial clustering
Saket et al.	2022	This article	✓	✓	✓	✓	✓	✓	★
Razgallah et al.	2021	[32]	✓	✓	✓	★	★	✓	✕
Liu et al.	2020	[33]	✓	✓	✓	★	✕	✕	★
Calik et al.	2020	[34]	✓	✓	✓	✕	✕	✕	✕
Pan et al.	2020	[35]	✓	✕	✓	★	✕	✕	✕
Qamar et al.	2019	[36]	✓	✓	✓	★	✕	✕	★
Prerna et al.	2019	[37]	✓	✓	✕	✕	✕	✕	✕
Aiman et al.	2019	[38]	✓	★	✕	✕	✕	✕	✕
Ebtesam et al.	2019	[39]	★	★	✓	★	✕	✕	✕
Raima et al.	2017	[40]	✓	✓	★	★	✕	✕	✕
Muttoo et al.	2017	[41]	✓	✓	✓	★	✕	✕	✕
Alireza et al.	2016	[42]	✓	✓	★	★	✕	✕	✕

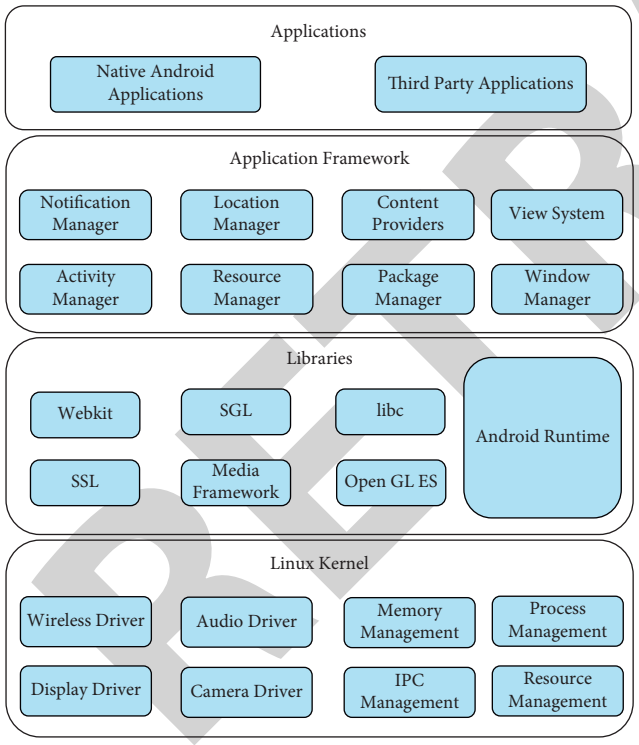

Figure 1: Android architecture.

Every manager is responsible for providing the important services like

Resource managing service: it is used to provide access to static manifest resources such as user layouts, strings, and other settings.

Notification managing service: this service is used by Android applications to display notifications on the screen. This service can be disabled manually by the users in the settings option.

Content providers: they are used for data sharing among applications. If an application wants data of other application, its content provider service communicates with the other application through interprocess communication.

Activity managing service: it is used to control all the activities of an application. It maintains an activity log file to record the activities. This log file can be used to debug the application in case of app crash.

Location managing service: it is used to provide access to location of the device. An application may request location access during or after installation. The request can be allowed or denied using location managing service. By default, location service is disabled due to privacy concern. Users can enable the service from the setting menu of the device.

After the booting process, a zygote process starts the DVM by preprocessing the native code. The zygote process speeds up the application stacking instances of libraries to be imparted to the newly stacked user applications [46]. The system framework provides a unique instance of Java system libraries to the app developer. The key core libraries which are provided to Android developers are described in Table 3.

2.1. Android Package Kit (APK) Design. Android package kit (APK) is a zip file consisting of a few files and directories as shown in Figure 2. The major design components of an APK file are as follows:

Manifest file: the *AndroidManifest.xml* file contains the metadata information, for example, APK name, application permissions, activities, services, content

Table 2: Android application binary interface [44, 45].

ABI	Instruction sets supported	Brief description
armeabi-v7a	*armeabiVFPV3-D16Thumb-2*	32 bit ARM-based processors use this type of ABI. VFP hardware floating point instructions and Thumb-2 are used as registers.
arm64-v8a	*AArch64*	It is used by ARMv8-A processors, which are built onAArch64 framework. It also contains the single instruction multiple data (SIMD) architecture.
x86	*MMXx86 (IA-32) SSE/2/3*	This ABI is used by the processors that support "x86," "i386," or "IA-32" instruction sets.
x86 64	*x86-64POPCNT MMX SSSE3 SSE4.X*	This ABI is used by the processors that support "x86-64" instruction sets. The instruction codes are generated by Gradle and corresponding interrupts are generated according to the type of unconditional instruction sets.

Table 3: Android key core libraries.

Core library	Functionality	Key APIs
android.content	Provides classes to access and store device data	Package management, resource management, and content management
android.opengl	Provides interface to access OpenGL ES	2-D and 3-D graphics management
android.text	Provides text and input manipulation service	Drawing management, text management, input validation
android.os	Provides access to operating system services	System calls, process management, memory management
android.webkit	Provides access to browse webpages on the web browser	Plugins, cookies management, session management

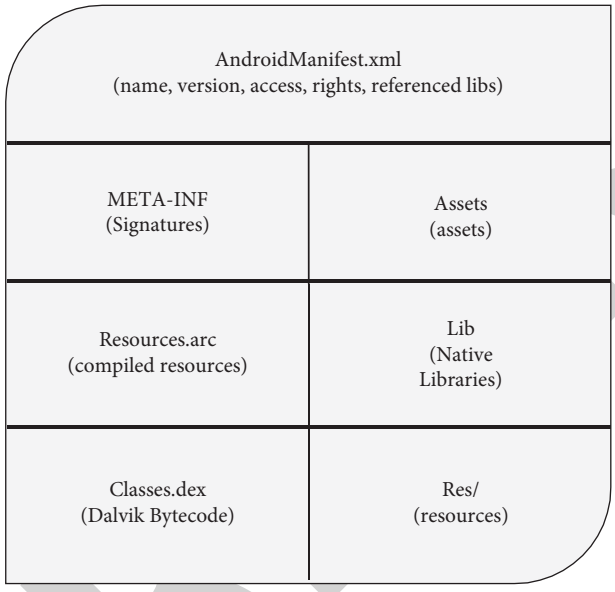

Figure 2: APK design components.

providers, old and new version support, and libraries to be attached. The major tasks performed by this file are as follows:

META-INF: it contains the signature of the application developer authentication to authenticate the external developer.

Assets: they are directories that consist of static application resources which cannot be linked to *resources.arsc* file.

Resources.arsc folder: this folder stores the symbols, pictures, alphanumeric constants, user interfaces, menus, and images incorporated into the parallel. Folder resources contain static resources.

Native libraries: these libraries are stored in *Native Development Kit (NDK)*. The Native Development Kit contains a set of APIs that enables an Android user to use C and C++ code with Android and provides platform libraries you can use to manage native activities and access physical device components, such as sensors and touch input. Reuse your own or other developers' C or C++ libraries.

Dex Files: they are executable files that store the Dalvik bytecode to be implemented on the Dalvik Virtual Machine (DVM).

The app development procedure is represented in Figure 3. Following are the important steps involved in Android application development:

(1) Step 1: the compiled Java files are converted to bytecode. During conversion, all the required resources are fetched from the res folder.

(2) Step 2: the converted bytecode is allowed to execute on Dalvik Virtual Machine (DVM). The output of this step is dex file.

(3) Step 3: the dex file is linked to resources.arsc folder in order to use the inbuilt packages, classes, and functions.

(4) Step 4: the dex file and resources are built to generate the APK file.

(5) Step 5: the generated APK file is signed with a valid certificate to provide authentication and data integrity.

(6) Step 6: the APK file is ready for publication and installation process. The APK is verified during the time of installation and in case of certificate mismatch, the error is displayed on the screen.

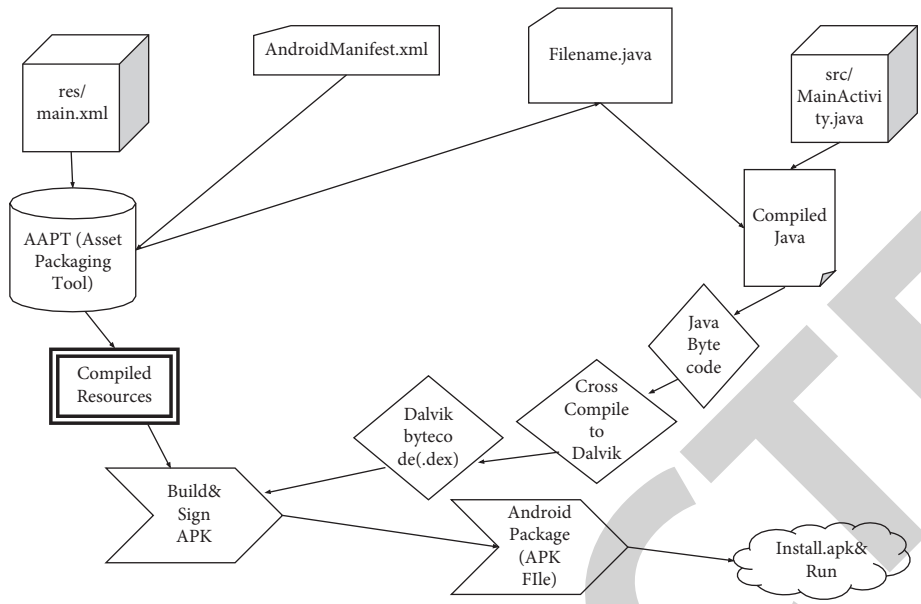

Figure 3: App development procedure.

2.2. Android Application Components. The main components of an Android app which are shown in Figure 4 are described below:

Activity: it is the user interface component of an Android application. An activity is declared in the AndroidManifest.xml file. Activities are initialized using the Intents.

Service: service component is responsible for performing background tasks, for example, playing music or downloading something on the device. Services are launched using *Intents* described below.

Intent: Intent is a messaging component through which application components send messages to each other. Android Intent can be used to instantiate new activities or get output from other activities. When an application component queries for an event, Intents pass the query to receiving application component. Services are initialized bypassing the Intent to perform a specific task.

Content provider: content provider is used to provide an interface to system and user applications for accessing the metadata information. The content providers act as the data stores. These data stores are available through the application-characterized Uniform Resource Identifiers (URIs).

Broadcast receivers: they respond to the messages which are received from external applications and system applications. The applications can also broadcast message to other applications to inform them about newly downloaded data available for them to use.

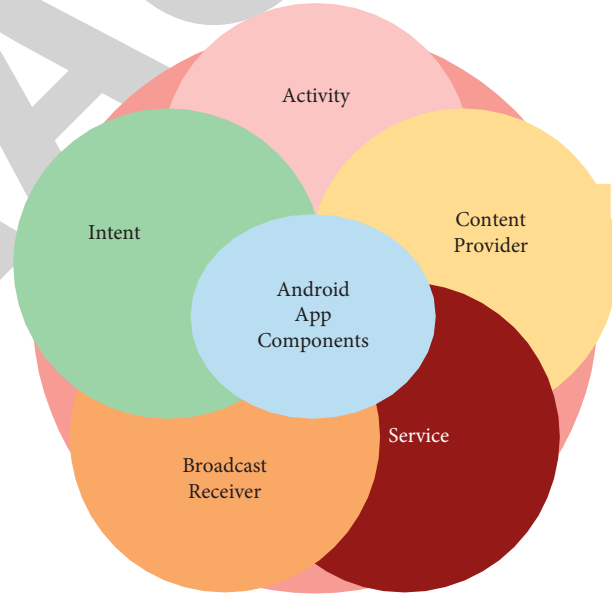

Figure 4: Android application components.

2.3. APK Sandboxing. Android is developed as a secure operating system with a thought process to secure user information, developer applications, and system network [47]. However, the security relies upon the developer's ability to follow the best and most secure improvement practices. Likewise, the user must know about the consequences an application may have on the information and device security. Anti-malware applications can skip vigorous malware checks due to enforced OS security model and hence they have limited filtering options. Android kernel executes the Linux discretionary access control (DAC). Each application is secured with a unique id (UID) inside a separated sandbox. Sandboxing hides the applications and their framework administrations from the interference of other applications. Android system ensures network security by using a feature paranoid network security, a component to control Bluetooth and Internet connectivity [48]. The

process of sandboxing an Android application is shown in Figure 5.

Application signature provides data integrity to secure app data. The application signing process puts the Android app into a virtual sandbox environment after assigning it a unique UID. The hash value of UID is matched with the hash value of a valid app certificate during the installation process. In case of a forged signature or invalid certificate, the application installation is aborted by the Android system.

2.4. APK Permissions. Android is based on a permission-based safety model to restrict the applications from accessing confidential information such as system network, GPS, and contact information. These permissions must be declared by the developer with the help of <*uses permissions*>tag in *AndroidManifest.xml*. The restrictions are imposed during the app installation. Android permissions are classified into the access levels described in Table 4. The normal permissions do not pose critical threats to users, system applications, or the device. They are granted automatically during the app installation time. They can be changed later from the application settings. Some common examples include INSTALL SHORTCUT, SET WALLPAPER, SET ALARM, etc. In contrast, the dangerous permissions belong to the high-risk category because they provide a path to access the private data and important APIs of the device. The user's consent is taken for accepting these permissions during installation time. READ SMS, SEND SMS, GET LOCATION, etc. are some common examples of dangerous permissions.

2.5. Android Secure System Partitioning. Android device partition is made from the kernel and hence the partitioning is read-only to protect the system from unauthorized modifications. The system cache and memory card space are also secured with the necessary privileges to avoid any data tampering by the attacker. Dynamic partitioning is performed with the help of the mapper module. This module resides in the kernel. The metadata containing block ranges are stored in the super partition. Hence, the super partition resides above the dynamic partition and defines block values for the dynamic partition. During boot-up init, metadata are verified and virtual blocks are made to display each dynamic partition. The layout of Android system partitioning is depicted in Figure 6. While building *Digi Embedded*, the following build artifacts are generated:

system.img: it is the system image that contains runtime libraries, essential APIs, and default system applications that are preinstalled in the device. System calls are used to fetch the required libraries at runtime.

vendor.img: this partition has all the binary files which cannot be shared publicly with the Android open-source development team. It contains all the copyrighted binaries and product-specific files.

boot.img: it contains the configuration files which belong to the kernel. It is initiated by boot loader *U-Boot*, and it mounts the necessary partitions required after boot.

recovery.img: it performs the same task as *boot.img* but in recovery mode. When the recovery mode is required, *U-Boot* executes this image in order to install any software update packages or to format the data.

u-boot-file.imx: it contains the bootloader known as *U-Boot*. It is executed when the device boots. It loads the device and starts the operating system.

2.6. Google Play Protection. Android comes with the secure app market called Google Play store to publish the applications so that the user can find, download, and install the Android apps. However, it still permits the user to install the applications through other unofficial markets. Google restricts the tertiary applications with Google Play Protect (https://developers.google.com/android/play-protect), a dynamic investigation sandboxed environment to stop any threat from accessing Google Play. Play Protect is a machine learning-based security method. Google Play is equipped for remote uninstallation if it discovers suspicious conduct. Android has the option of executing a security check while introducing applications from other commercial markets. The security check can be performed in two ways:

On-device scanning: Google Play Protect frequently scans all the applications which are installed on the device. If any suspicious application is detected, the user is notified to remove that application immediately. Hence, the user's data remain safe.

Cloud-based protection: rigorous security testing is performed by Google Play Protect to scan billions of applications. Only safe applications are allowed in the play store and suspicious applications are removed from getting downloaded. Cloud scanning protection ensures the user that the downloaded application is trusted and safe.

The internal working mechanism of Google Play Protect is demonstrated in Figure 7. In the initial step, the developer signs the developed APK file using a valid certificate and sign-key. The signature is then verified by the Google security team. For valid signatures, the developed app is scanned by Google Play Protect. On successful scan, the app is ready to get published on the play store.

3. Security Issues in Android

This section provides a detailed description of the user and device security concerns. Google releases security fixes frequently to resolve device bugs and to improve device security, but the malware developers still find detection evading mechanisms. The first step in dealing with a security bug is to identify the seriousness of the bug and the resources which can be affected with the bug. The seriousness level helps the researchers and security team to prioritize the issue so that the required bug fixes are deployed to the users [52]. The seriousness levels are briefly described below:

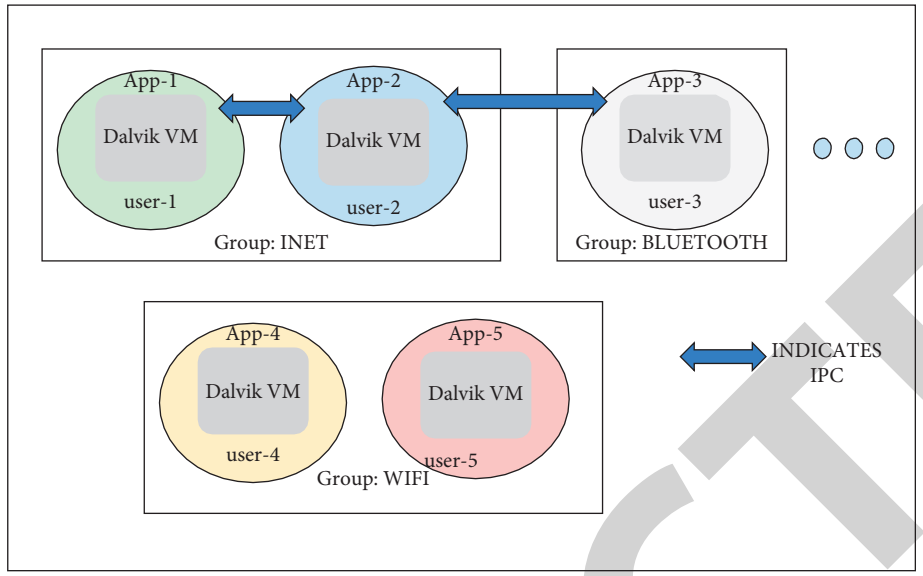

Figure 5: APK sandboxing in virtual environment.

Table 4: Levels of APK permissions in Android applications [49–51].

Levels	Description	Examples
Normal permissions	These permissions are less risky to the user, system applications, or the device. They are granted automatically during the app installation time. They can be changed later from the application settings.	INSTALL SHORTCUT, SET WALLPAPER, SET ALARM, VIBRATE, INTERNET, BLUETOOTH, ACCESS WIFI, etc.
Dangerous permissions	These permissions belong to high-risk category because of their ability to access the private data and important APIs of the device. User's consent is taken for accepting these permissions during installation time.	READ SMS, SEND SMS, WRITE SMS, GET ACCOUNTS, READ CONTACTS, READ LOGS, etc.
Signature-based permissions	These permissions are granted only if the signature of app maker certificate and the requesting app are matching. They are granted implicitly during the install time if the above criterion is fulfilled.	APP CERTIFICATE, APP SIGNATURE

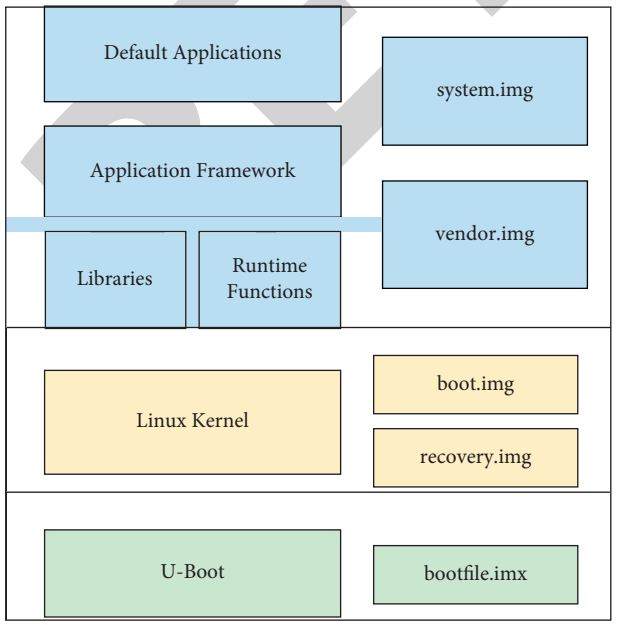

Figure 6: Android secure system partitioning.

Critical level: this is the most dangerous severity level and it must be resolved immediately in order to secure the device. The results of successful exploitation of a critical bug are listed below:

Unauthorized data access.
Remote code execution.
Remote boot bypass.
Remote denial of service attacks.
Remote data wipe.

High level: bugs with *high* level of severity have vicious outcomes. The results of successful exploitation of a high severity bug are listed below:

Remote access to authorized information.
Remote script execution to acquire sensitive data.
Bypassing the lock screen and security codes.
Exploitation of cryptographic vulnerabilities.
Remote ransomware attack.

Medium level: bugs belonging to this category are less harmful as compared to *critical-level* and *high-level*

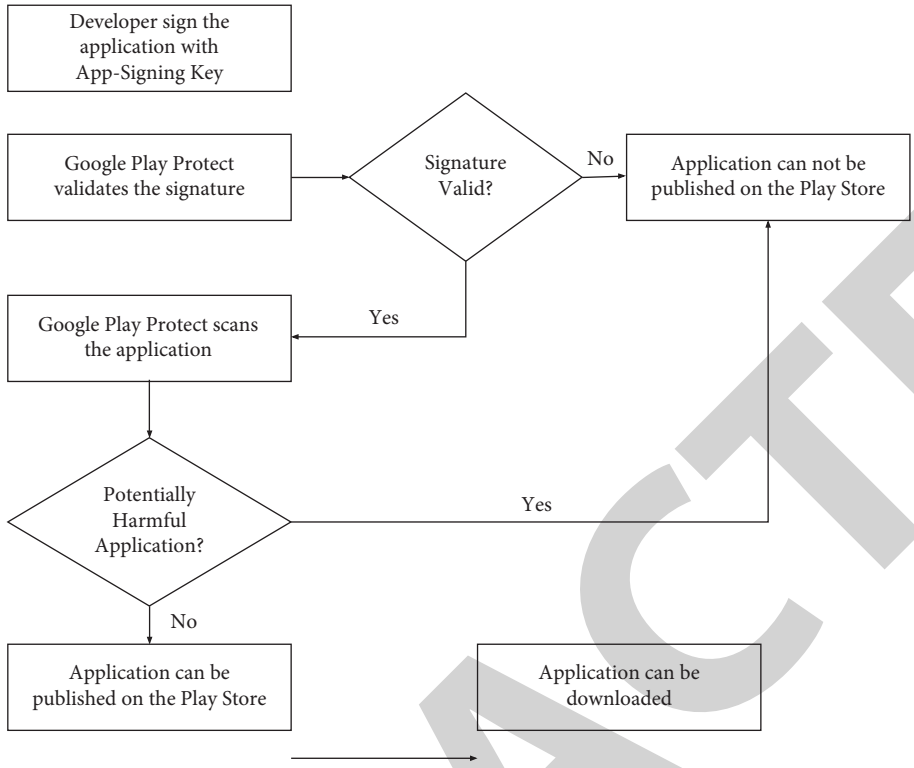

Figure 7: Google Play Protect working mechanism.

bugs, but they can damage device data on successful exploitation by an attacker. Following are the outcomes of successful exploitation of *medium-level* bugs:

Bypassing the permissions and gaining access to root permissions.
Bypassing Wi-Fi encryption.
Local code injection.
Local script execution.

Low level: bugs which have low level of seriousness do not harm the devices much but still they need to be patched properly in order to secure device data. These bugs can have following outcomes if they are exploited successfully:

Removing user applications.
Random spam pop-up notifications.
Unexpected closure of system applications.

The major categories of the attacks, issues related to device updates, and the security fixes provided by Google to enhance the device security are discussed in the following sections.

Cai [53] proposed a viable malware detection that is self-sustaining and does not require retraining. The author looked into the long-term viability of learning-based classifiers. The author began by defining sustainability indicators and developing the DroidSpan classification system, which focuses on capturing sensitive access distribution. Furthermore, the author analyzed and contrasted DroidSpan's long-term viability with five cutting-edge detectors that had 13,627 benign apps and 12,755 malware apps.

3.1. Android Vulnerabilities and Attacks. Though Android OS is secure, it is vulnerable to various kinds of attacks. When a malicious app is installed on the device, it creates undesirable threats to the device security. According to *Mobile Security Threats and Vulnerability Report 2019-20* (https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/), around 50% high-risk vulnerabilities are identified in Android operating systems as compared to its contender iOS, in which the percentage of high-risk vulnerabilities was 40%. The amount of vulnerabilities reported in Android and iOS is depicted in Figure 8. Elish et al. [54] offered a flow analysis for app pairings that calculates the level of risk associated with prospective communications. Based on inter-component communication (ICC) between interacting apps, their approach statically evaluates the sensitivity and context of each inter-app transaction and provides fine-grained security policies for inter-app ICC risk assessment. The authors conducted an empirical investigation on 7,251 apps from the Google Play store to determine which apps connect via ICC channels.

The most critical types of Android system vulnerabilities are as follows:

(1) Flaw in security mechanism implementation: during the time of implementing security mechanisms, rigorous testing plays an essential role to detect and

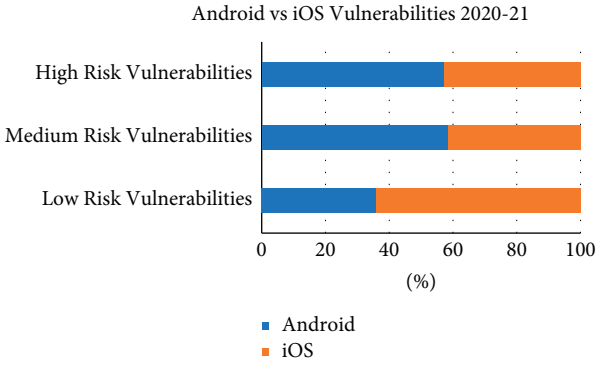

Figure 8: Android vs iOS vulnerabilities.

analyze security bugs. If these bugs remain undetected, the attacker can remotely exploit them and gain access to the root level of the system.

(2) Application source code vulnerabilities: these vulnerabilities are associated with the native source code of Android applications. They can be exploited using a code regeneration technique in which an attacker can inject a malicious payload in the original APK file.

(3) Misconfiguration: they are the insecure vulnerabilities that are present in Android applications due to improper security configurations. They can be exploited easily by modifying the contents of *config* files.

During 2019-2020, the percentages of security mechanism vulnerability and application source code vulnerability were very high as depicted in Figure 9. These vulnerabilities pose more risk to device security and their exploitation may lead to loss of data including sensitive user data.

Significant malicious attacks along with their consequences are described in Table 5. The consequences of these attacks can range from small data loss to major financial damage. The most common examples of critical data loss include password captures, ransom payments, stealing of login credentials, and hijacking device administration.

3.2. Software Update Issues. Android software update takes months before reaching the end-users. This principle is known as *fragmentation*, in which several Android versions remain vulnerable due to late or unavailability of software updates [58, 59]. Software updates are sent to the devices in batches, in which 1% of the major devices receive the update in one or two days. Security checks and validations are performed by the Google security team to check for any errors or bugs. After successful validation, the updates are rolled out to the next batch, which contains 25% of the devices. Within a few weeks, 100% of the devices receive the update. Due to this fragmentation, the devices which are not receiving software updates on time remain vulnerable to various attacks. Also, during the software update, Android does not verify the embedded privileges in the updated apps

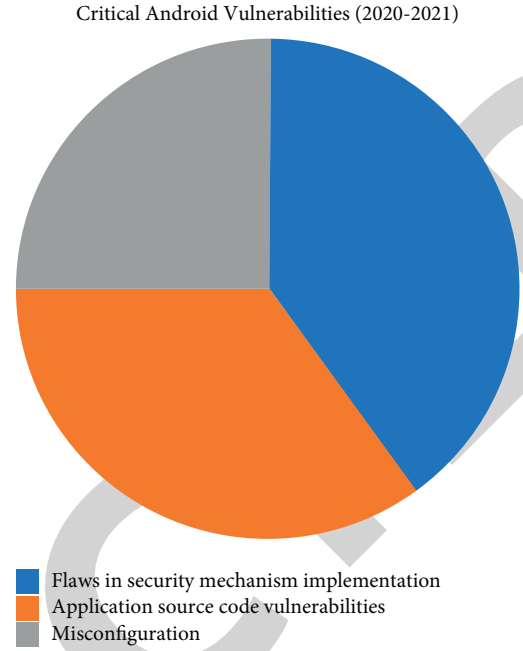

Figure 9: Significant Android vulnerabilities detected during 2020-2021.

and hence it compromises the device security [60]. The software update rollout process is demonstrated in Figure 10.

3.3. Android Mobile Network Quality of Service. Most of the Android users access the Internet data using 3 G/4G networks. According to Mobile Insight report 2020 (https://www.inmobi.com/insights/reports), the total network data volume has contributed nearly 75% of global network traffic since 2016. Android applications access Internet data with the help of socket API calls. Figure 11 shows Android cellular network stack architecture. Internet connectivity is done using cellular interface and base stations.

The protocol stack contains several layers to provide specific functionalities. These layers are briefly described below:

Session control: it is responsible for initializing and maintaining session between Android device and cellular network. It also handles voice sessions.

Mobility control: it handles the location-based device services. Android device's real-time location is offered by this layer.

Resource management: this layer is responsible for allocating and managing resources. Mobile data used by Android applications can be checked using device settings option.

Data link layer: it provides error checking mechanism. Device hardware address is contained in this layer.

Low-level layer: this layer provides the wireless communication standards which are used to access connectionless data.

Table 5: Significant Android attacks [55–57].

Attack name	Brief description	Consequences
Privilege escalation	Privilege escalation attacks exploit the kernel-level vulnerabilities to get the root privilege of the device. Publicly available device modules can be intercepted by the attacker to gain access to the critical permissions.	Remote access to root-level privileges which may result in getting complete control of the Android system.
Personal identity theft	This attack happens when the user grants critical permissions to unknown malware apps and unintentionally allows access to private data.	Unauthorized access to sensitive user data, which an attacker may use to hide its identity while executing an attack.
Monetary threat	Malicious applications can gain financial advantage by placing calls or making the user to subscribe to premium services without user's consent.	This attack leads to financial damage like money loss and online banking data capture.
Malicious ad campaign	Malicious ads may attract users to download malware apps. These ads can redirect the application to remote target attack server and hence the device security can be compromised.	Loss of user's credentials like password and saved card information. The attacker can silently monitor online transactions and steal private information.
Denial of service (DoS)	This attack takes place when an app exhausts the already limited resources like memory, battery consumption, bandwidth, and so on and restricts the normal users to execute their functions.	Unavailability of important resources and services to genuine users.

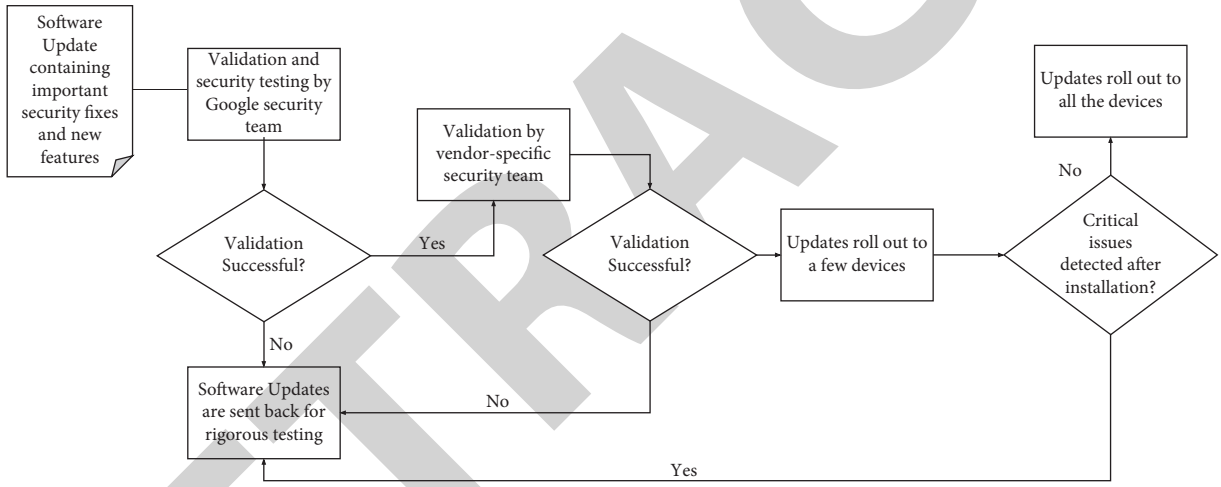

Figure 10: Android software update rollout process.

3.4. Android Security Fixes. Android frequently releases various security fixes and novel enhancements to improve device security. These fixes are important to fix several bugs present in the Android operating system. Before releasing important fixes, Google security team validates and tests the patches to ensure device safety. All security fixes and new features are rolled out in the form of software updates. The important security fixes introduced in the overall Android versions including latest *Android 11* are briefly described in Table 6.

The different Android versions along with their release times are depicted in Figure 12.

4. Android Malware Variants and Malware Embedding Techniques

This segment discusses significant malware variants developed during 2017–2021 and the stealth methods used by this malware to hide from detection. The most common strategy used by the attackers is to regenerate the Android app after inserting the malignant source code inside the *manifest* file. Furthermore, clone applications demand elevated level permissions from the user and compromise the framework by exploiting the approved permission, which prompts expending more resources or performing pointless tasks. At the point when the user installs this regenerated malevolent APK downloaded from an unreliable source, the security of the device gets traded off. Apart from regenerating malicious applications, the attackers also inject malicious payloads in the app resources at runtime. Bytecode can be analyzed to separate the applications containing malicious payload from genuine applications by doing an investigation on their conduct and acquiring data from control flow and data flow charts. However, this technique has the drawback of requiring more storage and power.

Cai and Jenkins [63] presented a long-term Android malware detector capable of detecting new infections without the need for retraining. The authors looked into application runtime activities and used behavior analysis to distinguish between benign and malicious programs.

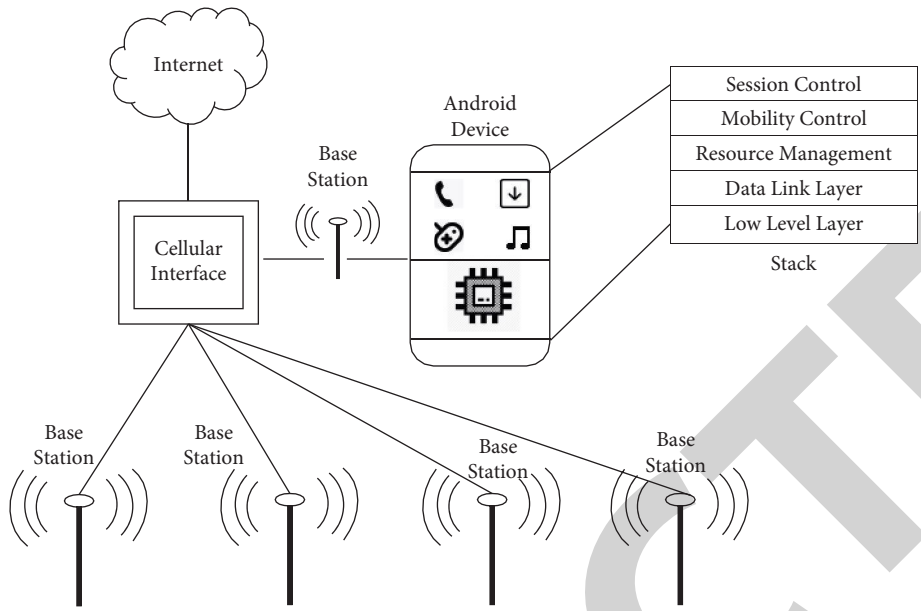

Figure 11: Android cellular network stack architecture.

Table 6: Android security fixes [61, 62].

Android version	Version name	Release year	Key feature	Important security fixes
Android 1.5	Cupcake	2009	First on-screen keyboard introduced	Security patches to remove chunk consolidation attacks were rolled out
Android 2.3	Gingerbread	2010	Interactive user design interface	Patches were rolled to fix format string vulnerabilities
Android 4.0	Ice Cream Sandwich	2011	Refined visual concepts and improved GUI	Security fixes to remove address space related vulnerabilities were rolled out
Android 4.1 to 4.3	Jelly Bean	2012-2013	Improved notifications and voice search facility	Various fixes rolled to patch Android-specific flaws
Android 4.4	KitKat	2014	Novel UI and improved status bar	Fixes to prevent SQL injection attacks were rolled out
Android 5.0, 5.1	Lollipop	2014-2015	Notifications on lock screen and smart lock system	Full disk encryption and smart lock to prevent authentication-based attacks
Android 6.0	Marshmallow	2015	Fingerprint lock was introduced	Boot-level encryption patches were rolled out
Android 7.0, 7.1	Nougat	2016	Split screen mode	Security fixes to prevent buffer overflow and stack overflow attacks were rolled out
Android 8.0, 8.1	Oreo	2017	Native picture in picture mode	User-level permissions were improved
Android 9.0	Pie	2018	Navigation using gestures and multifunctional UI	File-level encryption and protection against ransomware attacks
Android 10.0	Android 10	2019	Novel GUI with tapped buttons on the screen	Face authentication and physical device security fixes
Android 11.0 direct file access	Android 11	2020	Priority-based conversation notifications	Implementation of scoped storage to prevent buffer overflow attacks and race condition attacks
Android 12.0	Android 12	2021	Improved customized widgets, dynamic colors, and resolution	Implementation of hardware-based security with trusted platform

Without the need for continuous retraining, the proposed approach can last for five years.

4.1. Android Malware Variants. After every four months, the anti-malware solutions report a significant increase in the new malware families [64–66]. Most of the malware behave like legitimate application and silently enable critical hidden permissions especially *message permissions* and send the device data to the distant server, while some malware variants aim to drain the device resources, for example, high amount of battery consumption, memory, and processor utilization. Table 7 describes the characteristics of popular Android malware. Some of these malware have also infected

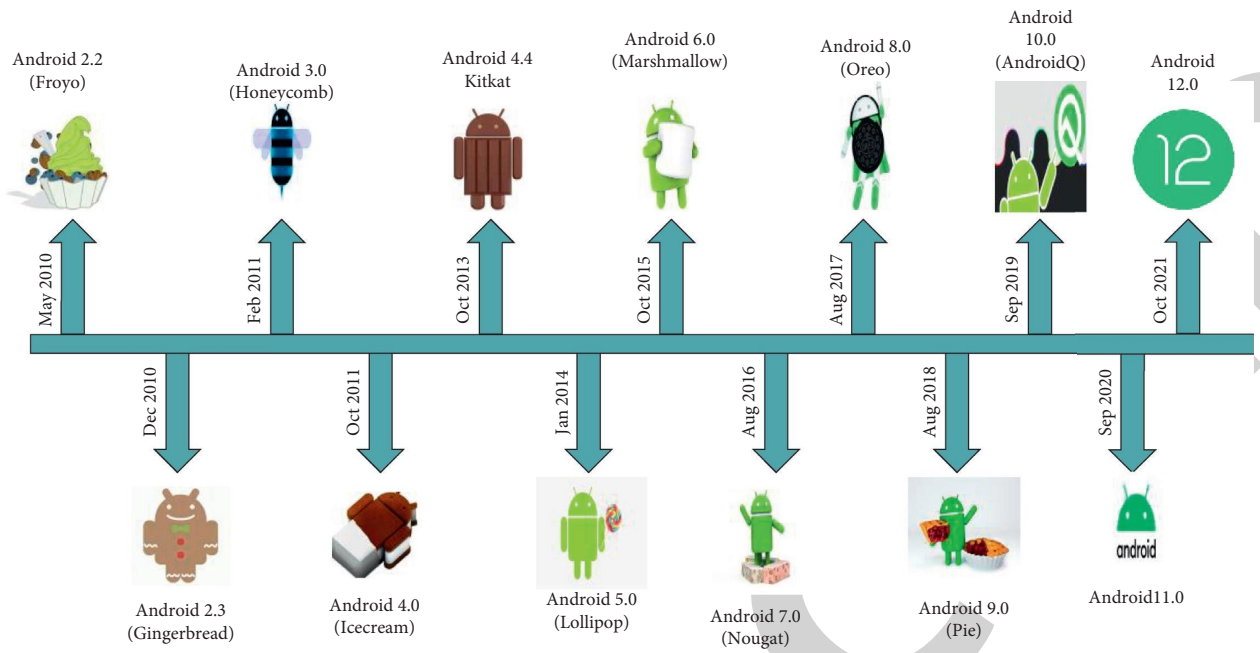

Figure 12: Android version timeline.

official play store [82]. Among all the malware types, *virus*, *ransomware*, and *Trojan* are famous for doing severe damages. The *jisut* [83] ransomware family affected millions of devices during the past few years. The device data affected by this type of ransomware can be saved after providing the ransom amount. However, this does not guarantee that the attacker will not erase user's data even after getting the amount. Figure 13 represents the timeline of Android malware variants evolved during 2017–2021. The primary motivation behind the malware which were developed in 2017 is to fool the users into tapping on web advertisements, which generates income for the parent companies. Besides, the malware can also gain root access and remotely send device data to the malware developer's server. The famous malware variant belonging to this category was the *HummingBad* malware [90]. More than ten million Android users got infected with this malware after they installed some vulnerable applications from third-party markets. The most common malware symptoms were unusual advertisements pop ups, unexpected software update pop ups, and high battery drainage in less time. In late January 2018, Google uncovered and brought down more than 0.7 million malignant applications from Google Play, an expansion of around 65% contrasted with the earlier year. According to Kaspersky security report [91], more than five million malicious APKs and one million new mobile banking Trojans and ransomware Trojans were detected in 2019. Banking Trojan malware resides in the mobile phone memory and waits for the starting of a mobile banking application. If this event occurs, the malware gets the details of the running application and shows notifications for fake login and password form. When user enters genuine login credentials, it is immediately sent to the remote attacker's system. The malware figures expanded exponentially in 2020 and more than twenty-five million Android devices got infected by malware [92]. The expansion of these numbers clearly specifies that the malware developers were successful in getting escaped from the malware detection mechanisms.

In 2021, most of the malware applications were supplied from new Coronavirus (COVID-19) based domains [93, 94]. These domains were created with the goal of misleading the users. Most of the malware apps discovered were group malware that range from spyware to ransomware, and even trojans intended to wipe out the user data for individual or money related information. In January 2021, more than 600 malicious applications containing coronavirus-based keywords in their manifest files were detected. Malware belonging to this category target the users who used to install third-party applications. These malware hide themselves in another application which looks genuine. Majority of applications infected by this malware have a label named *covid information provider*. On executing these apps, they were detected as *Android.Trojan.Information.Stealer* and *Android.Trojan.Fake.Installer*. These figures expanded exponentially in the subsequent months and more than fifty thousand COVID-19-based malware applications were detected till October 2021. Figure 14 shows an exponential growth of COVID-19 themed Android malware detected from January 2021 to October 2021.

4.2. Regenerating Android Applications. Attackers can regenerate or repack the Android apps by decompiling the mainstream free/paid applications from the well-known markets, embed, add the malware payload, recompile the Trojan applications, and convey them by means of the unsecure market stores. An application can be recompiled and regenerated using reverse engineering process.

Table 7: Characteristics of popular Android malware.

Malware type	Famous examples	Self replication ability	Stealth execution ability	Espionage ability	Host requirement
Virus	Universal cross-site scripting (UXSS) attack, command and control (C and C) [67], CardBlock, CardTrap, Android installer hijacking [68], and crossover [69] are most prominent Android viruses.	Yes	No	Yes	Yes
Trojan	Foney [70] Fakeplayer [71], and Zsone [72] are a few examples of Android trojans.	No	Yes	Yes	Yes
Spyware	GPSSpy and Nickyspy [73] are some well-known spyware applications.	No	Yes	Yes	No
Ransomware	FakeDefender.B [74] behaves like Avast anti-malware and generates fake malware alerts.	No	Yes	No	Yes
Botnet	Beanbot [75], Geinimi [76], and Anserver-bot [77, 78] are some well-known Android botnets.	No	Yes	Yes	Yes
Backdoor	Basebridge, KMin, and Obad are eminent examples of backdoors [79].	No	Yes	Yes	Yes
Keylogger	FlexiSpy [80] and mSpy [81] are some common examples of Android keyloggers.	No	Yes	No	Yes

Regenerating applications is one of the famous malicious app generation methods. More than 85% instances from the Malicious Genome Dataset are regenerated malware app variants [95, 96]. The flowchart for regenerating an Android app is shown in Figure 15. The steps to repackage an original APK into a malicious application are as follows:

Step 1: the source APK file is decompiled and dex files are extracted using apktool. The extracted files are stored into a folder having same name as source application's name.

Step 2: malicious payload is created using Java and compiled class files are embed into benign application's bytecode.

Step 3: the modified application is regenerated and recompiled using apktool.

Step 4: modified APK is signed using apk-jarsigner, a utility which takes jar files and generates signature for the application.

4.3. Stealth Malware Techniques. The anti-malware apps installed in the Android device cannot perform the deep analysis as compared to the desktop anti-malware apps. Malicious app developers take advantage of these vulnerabilities and obscure the malware payloads to hide against the professional anti-malware solutions. Stealth malware techniques like code encryption, reflection code, key permutations, runtime loading, and native code execution [97, 98] remain a serious concern for behavior-based malware detection applications. Reverse engineering can be done on Dalvik bytecode due to the easy accessibility of type safe data. Code obfuscation techniques can be easily executed on the bytecode with the help of several optimization tools available online. Dexguard [99, 100] is a commercial optimization tool used to ease the task of code transformation. The genuine APK file is extracted and all the important methods, local variables, and global variables are substituted with unreadable code to strengthen the reverse processing. Faruki et al. proposed an automatic Dalvik bytecode obfuscation technique to obtain new malware variants.

Cai and Ryder [101] presented a longitudinal characterization study on Android apps to identify and observe the build and execution nature of the apps. The researchers used a lightweight static technique to examine the execution code of 17,664 programs over the course of eight years. They also discovered that the functions of applications are heavily reliant on the Android system design, and that this reliance is growing over time. In addition, while the Activity components invoke lifecycle callbacks, the event callbacks are based on the user interface rather than the system interface. Some significant code modification techniques which are used to hinder malware detection strategies are as follows:

(1) Dummy code insertion: dummy code insertion is a famous code insertion technique to change the executable size and escape the signature-based anti-malware solutions. Dummy code insertion maintains the syntax of the initial app, but it modifies the instruction opcode sequence to change the malicious app signature. "*Goto*" instructions can be used to alter and reorder the opcode maintaining the original semantics. These techniques can be used to escape the signature-based anti-malware solutions [102, 103]. Figure 16 shows the process of injecting dummy code into an original APK file.

(2) Package retitling: Android packages uniquely identify Android applications. Many anti-malware programs use basic package or class names of a known malicious app to detect signature [104]. Such simple mappings can be used to escape the signature-based anti-malware. Figure 17 shows how a malware can be escaped from signature-based detection technique.

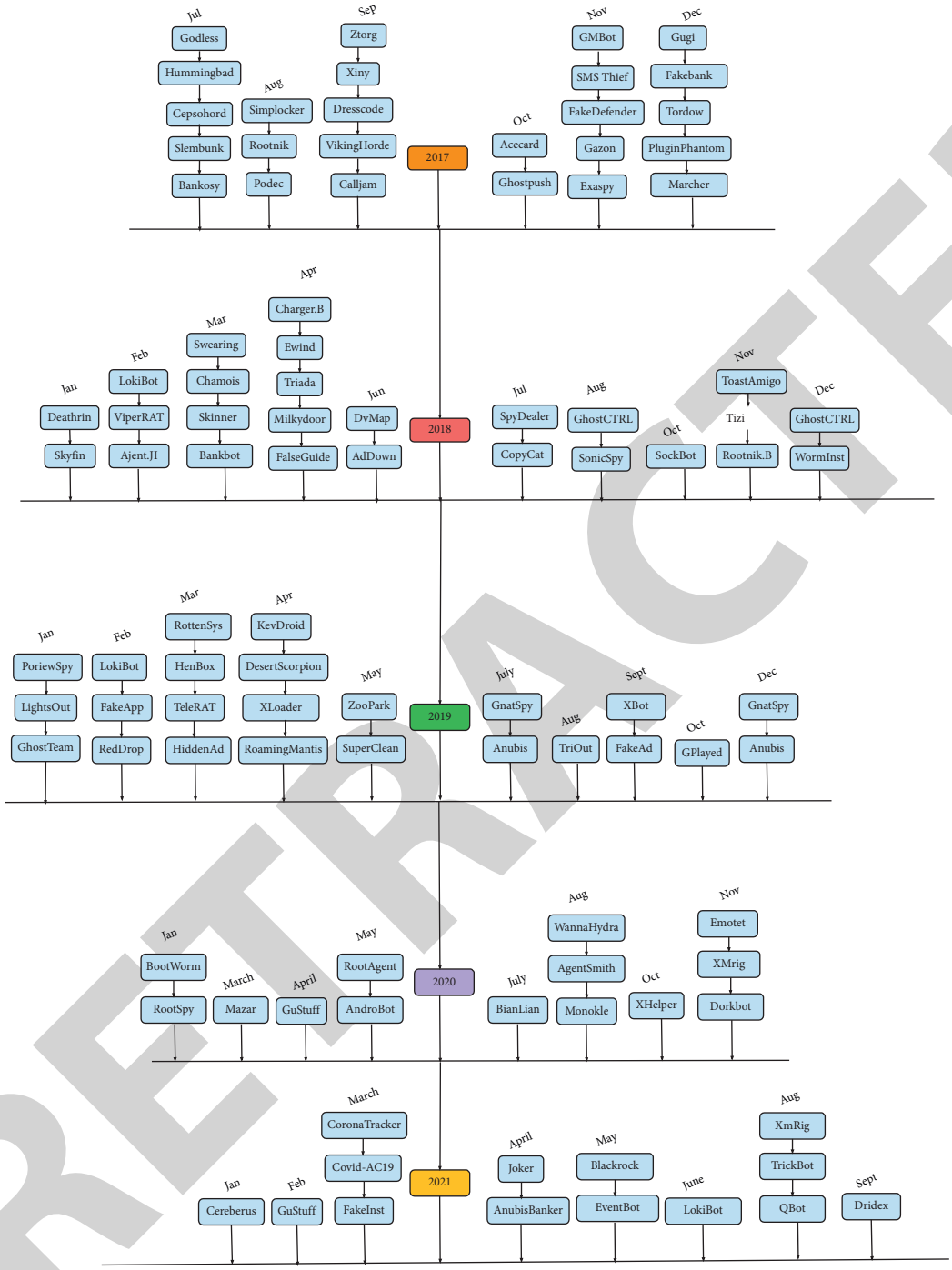

Figure 13: Timeline of significant Android malware variants detected during 2017–2021 [3, 13, 84–89].

The static features are extracted and filtered using bloom filter. When signature is generated using extracted features and standard packages, an attacker generates a different signature with similar standard packages and malicious payload. Since the attacker uses same standard package to regenerate malicious application, the application remains undetected by signature-based detection mechanisms.

Suarez-Tangil and Stringhini [105] conducted a deep analysis of Android malicious app behavior consisting of more than 1.2 million malware samples with 1.28 K families during a period from 2010 to 2017. The authors aimed to understand the evolution of repackaged malware, separated the components that were unrelated to malware, and analyzed the behavior of malicious riders using differential analysis. The samples were collected from distinct antivirus vendors.

(3) Modification of control flow path: control flow path of a code can be changed using the goto instructions.

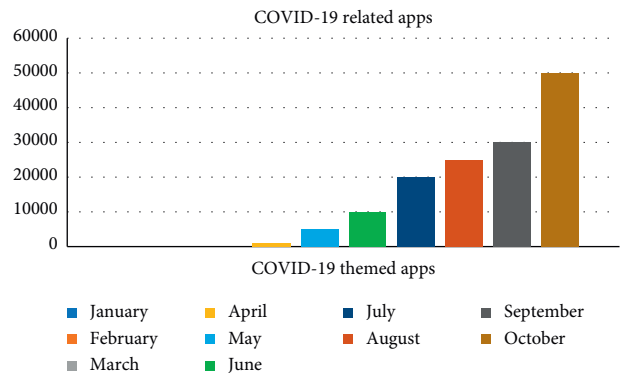

Figure 14: COVID-19 themed Android malware applications.

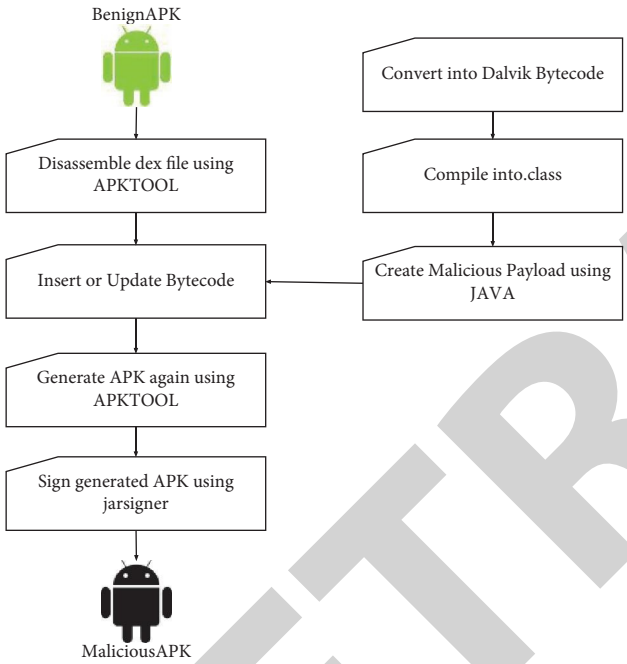

Figure 15: Regenerating malicious APK.

Though simple, such techniques escape the professional anti-malware solutions [102, 106]. Figure 18 shows the difference between control flow paths of two source codes with no modifications in Figure 18(a) and with *goto* instructions in Figure 18(b). A malicious code string is injected in the original code snippet as shown in Figure 18(b). This string can read SMS and account details of the device.

Similar malicious strings are embedded by the attackers to hack the device. String injection is achieved by using *apktool*, which decompiles and recompiles the application. After injecting the string, the new APK file is recreated. This malicious file can be downloaded from third-party app stores.

(4) Java code reflection: Java bytecode can be manipulated by using *reflection API*. The API provides an interface to access the strings, interfaces, and classes which are preloaded in JVM (Java virtual machine). User applications allow Java reflection to create program class instances or method calling. To obtain the exact class name, implementation of data flow analysis [107] can be performed. However, the native strings can be encoded to harden the reflection API searching process. Such approaches can easily escape static analysis methods. Java code reflection is depicted in Figure 19.

5. Android Malware Analysis Techniques

Android malware analysis techniques are broadly classified into "*static*," "*dynamic*," and "*hybrid*" analyses. Static analysis methods check the code without executing it, so they are fast but can contain false positives. Dynamic approach examines the implemented code and identifies its behavior. Thus, these techniques are effective against malicious code transformations, but they are time-consuming. Hybrid approaches combine the advantages of both the static and dynamic approaches.

The above-mentioned techniques are explained in below sections.

5.1. Static Analysis. Static analysis techniques disassemble and decompile the Android app without actually executing it. These techniques focus on static information which can be obtained from *manifest* file data. The manifest file contains information about Android version, configuration, compatible versions, permissions, string values, static variables, and activity files. Some important static analysis techniques are mentioned below:

(1) Pattern-based malware detection: almost every anti-malware solutions use pattern or definition-based malware detection. In this technique, the interesting syntactic pattern or feature is extracted and a unique signature is created matching the particular malware [108]. Figure 20 shows the steps to detect Android malware using signature matching technique. Static features are obtained from manifest file data and signature is generated using the obtained features. The generated signature is matched with the signatures of existing Android malware variants. On successful match, the test APK file is considered malicious.

Pattern-based techniques are not effective against unseen and already known malware variants. Faruki et al. proposed *AndroSimilar* [109], a feature extraction technique to detect unseen and new malware variants. The authors extracted static uncertain features from the APK files and stored the features in bloom filters. These filters are used to identify the presence or absence of an element in a given set of elements. Further, they obtained a unique signature from bloom filter sequences and matched the signature with malware database signatures. *AndroSimilar* technique was efficiently able to detect repackaged APK files, but this technique did not perform well against obfuscated malware.

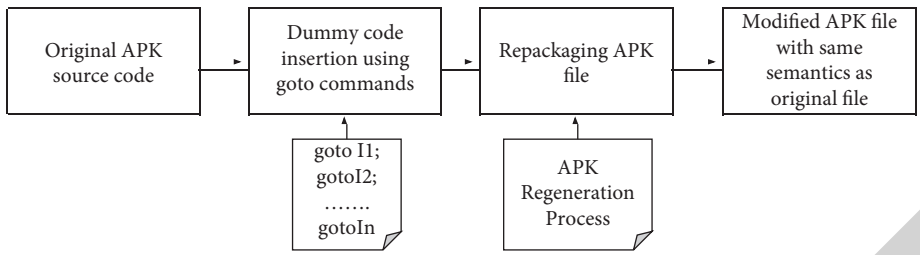

Figure 16: Dummy code insertion.

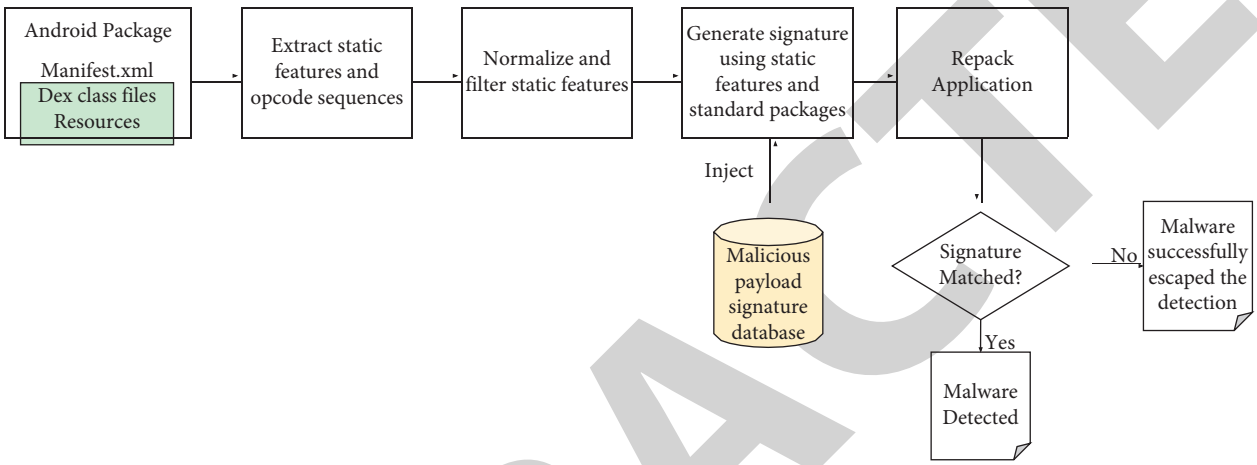

Figure 17: Package retitling.

(2) Required permission analysis: to access sensitive data, permissions are required in the Android system. This is the main principle of the Android security system. By default, no application has permissions which can compromise user security. The unknown applications which require critical permissions can be considered malicious [110, 111]. Sanz et al. [112] utilized <*get permissions*> and <*get features*> keywords residing in AndroidManifest.xml to search malicious applications. The authors used MLAs (machine learning algorithms) on a dataset of 250 malicious and 360 genuine apps. They matched the requested and critical permissions from the manifest and their similar APIs in the bytecode. Further, they mapped the reduced features on the MLA's on a dataset of around 125,000 malicious and genuine app dataset. The proposed methodology used by the authors can determine malicious applications, but their technique only used two features <*get permissions*> and <*get features*>. More number of features can be used to improve the overall detection. Enck et al. [113] created a tool, *Kirin*, to identify the composition of specific risky permissions to detect malicious attributes prior to the app installation on the device. The authors used certain security rules to identify malicious conduct of applications. *Kirin* can be improved by adding more novel security fixes into it to detect new malware variants.

(3) Analysis of bytecode: the Dalvik bytecode contains the information such as methods, classes, and interfaces. This information can be used to validate app behavior. Bytecode analysis is broadly categorized into data flow and control flow analysis [114, 115]. The control flow analysis searches the probable paths an app can take during its execution, whereas the data flow analysis searches the probable values during various points of execution. The possible execution paths can be traversed using control flow graph (CFG) in order to predict control and data dependency. An attacker can embed malicious code in the original source code and repack the APK file. Figure 21(a) shows control flow graph of malicious source code. Figure 21(b) shows control flow graph made by embedding the malicious instructions of Figure 21(a). The embedded instructions change the flow of control and when the repacked malicious APK is installed, the malicious event triggers.

Atici et al. [115] proposed an approach to analyze malicious Android applications using CFG and machine learning. The approach works by taking the CFG instructions as input vector to the machine learning algorithms and application behavior is classified into malicious and benign. The approach has good detection rate, but the overall detection time can be reduced by using string similarity algorithms.

(4) Decompiling Java bytecode: the process of decompiling the bytecode into source code is known as

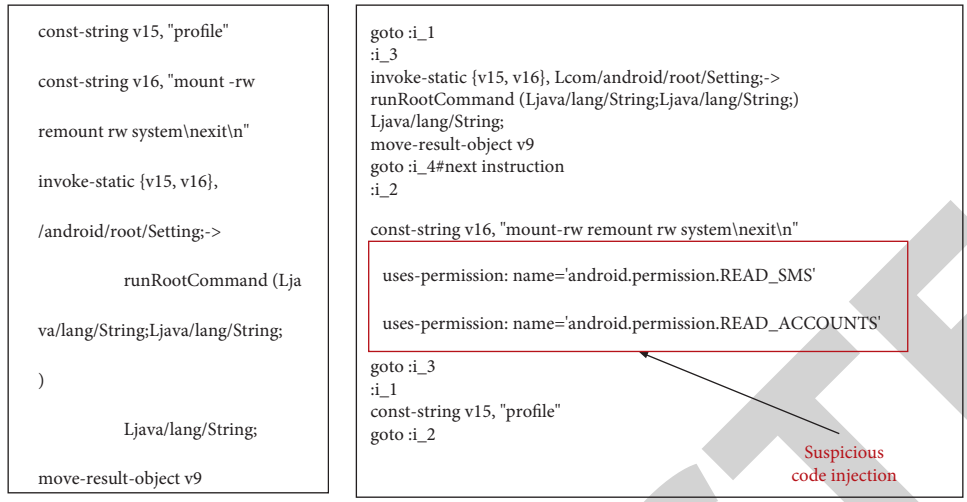

Figure 18: Modification of control flow path. (a) Code snippet without control flow modification. (b) Code snippet with modification.

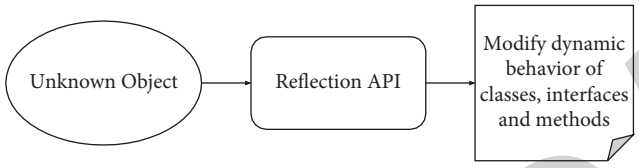

Figure 19: Java code reflection.

reverse engineering. The APK file can be decompiled using decompiler tools. Several decompiler tools are available online to decompile the Dalvik bytecode into source code. Octeau et al. created the *dare* tool for converting Dalvik bytecode to Java source code. Further, they performed the static control flow and data flow analysis on the source code with the help of Fortify Static Code Analyzer (SCA) framework [116]. The framework identifies various vulnerabilities that can be exploited by an attacker. The authors reverse engineered several applications obtained from third-party market stores and analyzed static source code path modifications. Bartel et al. [117] created the *Dexpler* plugin for generating source code from APK files. The tool can easily decompile several APK files, but it requires a good knowledge of APK reverse engineering. Crussell et al. implemented *AnDarwin* tool which contains *Watson Libraries for Analysis (WALA)* to detect the security leakage inside the apps on a dataset. The only drawback with the tool is that it can only detect the applications which have similar semantics.

5.2. Dynamic Analysis. Static analysis techniques are fast, but they do not work with transformed malware. Dynamic analysis methods run the app in a safe virtual environment and identify the malicious activities happening during the runtime execution of the application, but these methods may fail if any of the malicious execution paths get missed due to the triggering of non-trivial events. For example, a malicious activity did not happen during the execution of the malicious app but the event happens afterwards. Thus, the dynamic analysis can be evaded by delaying the malware execution. Some major categories of dynamic analysis technique are mentioned below:

(1) Overutilization of resources: Android mobile devices are small in size and they have a limited resource constraint. Denial of service (DoS) attacks can be performed by the attacker to overutilize the limited hardware resources. Parameters like CPU utilization, memory usage, battery utilization, network behavior, and system calls are greatly affected by this type of attack. Burguera et al. proposed *CrowDroid* [108] which monitors system resources and device behavior to detect malware. The detector collects traces from a huge number of real users with the help of crowd-sourcing and then the collected data are aggregated using two-means clustering algorithm. Since, the proposed framework analyzes the user data, it can also access confidential data and this is the major challenge to secure data privacy.

(2) System call tracing: specific malicious activities like sending SMS/emails, critical data leakage, and making fake voice calls with no user consents can be effectively detected by analyzing the system call trace [118, 119]. Figure 22 represents the method of accessing core libraries using system calls. The core libraries are demonstrated in Table 2.

Da et al. [120] proposed a model to detect Android malware with the help of system call tracing. The authors preprocessed normalized system calls to improve detection rate. The applications which do not require any critical permissions but access root-level system calls are classified as malicious. The proposed model can be improved further by incorporating dynamic libraries. Dimjasevic et al. [121]

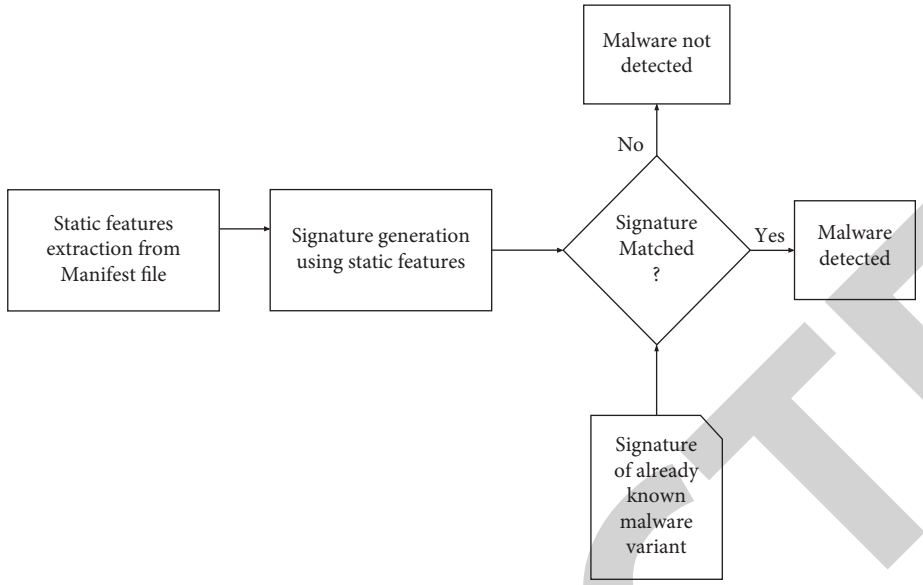

Figure 20: Android malware detection using signature matching.

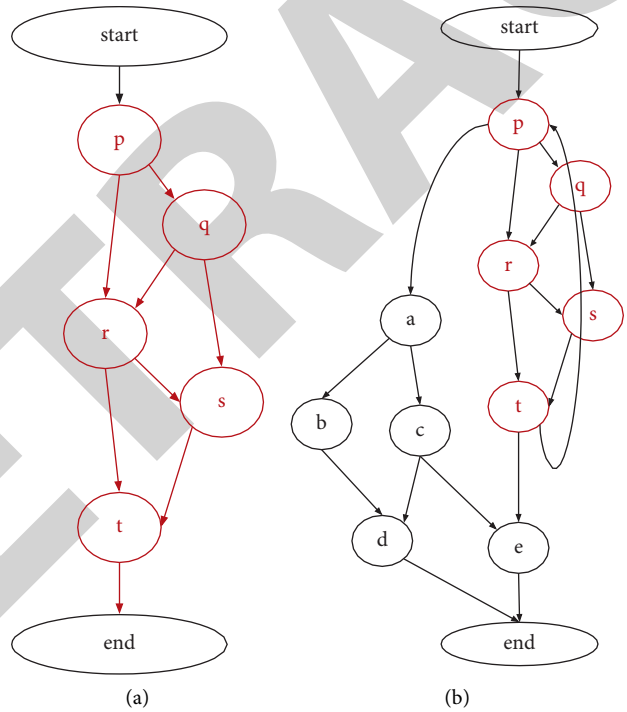

Figure 21: Execution control flow analysis. (a) CFG of malicious source code instructions. (b) Embedded CFG with benign and malicious instructions.

proposed *Maline* tool which works in three stages. During the first stage, it takes APK dataset and extracts log file information. In the next stage, feature extraction is done using system calls and dependency graphs are made. Finally, machine learning methods are used to train the feature set. The proposed method can be further improved by using deep learning methods to reduce feature extraction and processing time.

(3) Observation of virtual machine: the drawback of emulating an app in a virtual environment is that the virtual machine itself is vulnerable against unknown malware threats which weakens the analysis motive. To prevent this, virtual machine observation techniques can be used to detect app behavior by monitoring the behavior out of the virtual machine. *DroidScope* by Yan and Yin [122] reconstructs both device-level and Java-level semantics. The authors

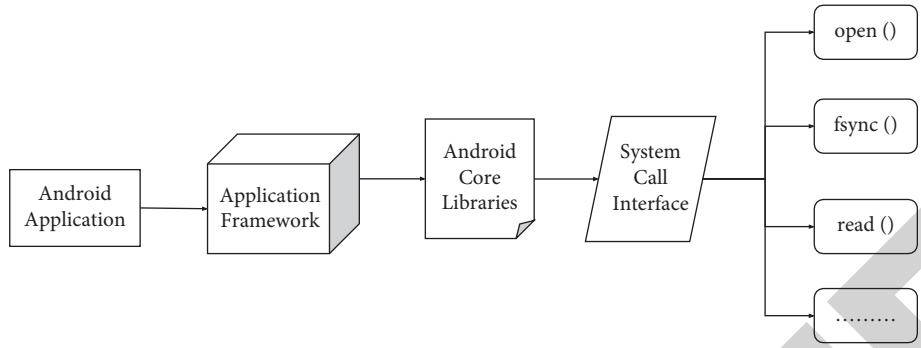

Figure 22: Accessing core Android libraries using system calls.

reconstructed the information about operating system and Java and used this information to depict malicious behavior of applications. In this manner, they analyzed both the hardware and software-level information. The proposed method can efficiently detect significant malware, but it requires dealing with the operating system kernel semantics.

6. Android Malware Classification and Detection Techniques

Malware safety solutions can be classified as detection based on feature derivation-based machine learning models and detection based on deep learning models [123]. The popular machine learning-based malicious app detection methods depend on attribute engineering and learning the features representing the features with the techniques which require a deep research knowledge [124–126]. Moreover, once the assaulter identifies the features, the malware detectors can be bypassed easily [127]. Deep learning [128] methods are famous for automatic useful feature extraction and thus outperformed the classical MLAs (machine learning algorithms) in several fields of system vision, natural language processing (NLP), and voice recognition.

6.1. Android Malware Classification and Identification Using Machine Learning. Several studies conducted in [129–132] showed that most of the malware types are similar in design and hence they can be classified using various machine learning algorithms. Some famous supervised machine learning algorithms used for classifying malware are support vector machine [133], Naive Bayes [134], decision tree [135], hidden Markov models [136, 137], J48 [138], random forest [139], AdaBoost [140], and instance-based learning algorithms [127]. Also, reduction algorithms can be combined with supervised learning for feature selection and extraction. Feature extraction is done to create an effective training model. Feature selection techniques like single variable removal, hierarchical, goodness evaluator, and weighted term frequency give better representative features that can be used during feature selection stage. However, improper feature extraction can lead to lesser accuracy and huge number of false positives. Figure 23 shows general model of identifying Android malware using machine learning.

Every machine learning algorithm requires feature set to train the models. In Figure 23, the Android application features can be obtained from manifest file, permissions, source code, API calls [141, 142], and opcode instruction sequences. These important features can be normalized and used as an input vector to various machine learning algorithms. The efficiency and performance of various machine learning algorithms depend on the underlying classification methodology and size of input vector dataset. The most common machine learning algorithms are briefly explained below:

Logistic regression algorithm: this algorithm utilizes the factual ideas and models a connection between the information and output numerical values. The framework is designed with the help of a logistic equation. The equation helps to yield the output values for a particular set of input values. The logistic model can be represented with the help of following equation:

$$P = \frac{1}{e - \left(b + c_1 x_1 + c_2 x_2 + \cdots + c_p x_p\right)}. \quad (1)$$

In equation (1), "p" denotes logistic output probability. "c" values are learned weights and "b" denotes the bias.

Linear discriminant analysis: linear discriminant analysis (LDA) is an extremely basic technique for reducing dimensions of large datasets. This algorithm acts as a prehandling step for machine learning-based applications. This technique is created to change the features into a lower dimensional space, which amplifies the proportion of the outside-class variance to the inside-class variance, hence providing the maximum separation among various classes.

K-nearest neighbor (KNN): this algorithm is mainly used to solve problems based on classification. The algorithm requires exceptionally less execution time, and in this way, it is a widely used algorithm. The "K" in the algorithm denotes the count of neighbors which are defined by the users. Euclidean distance is used to measure the K closest neighbors of the dataset and anticipate the yield as per its neighbors.

If a and b are two coordinate points in Euclidean m-space, the function defining the Euclidean distance between two points a and b is represented by

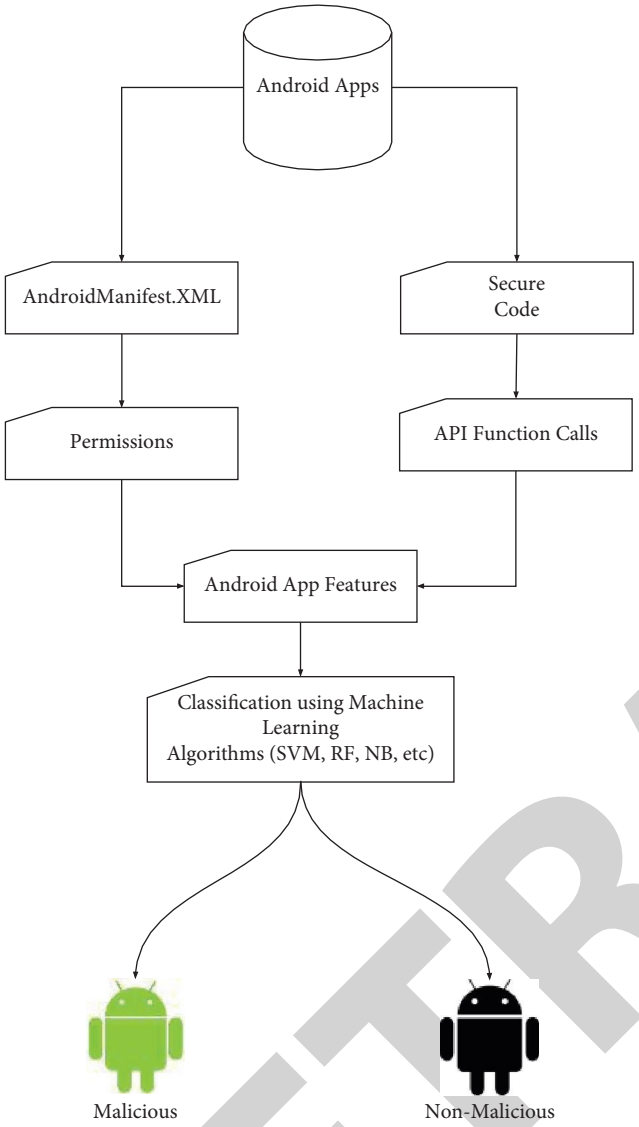

Figure 23: Detecting Android malware using machine learning.

$$D(a,b) = d(b,a), \\ = \sum_{i=1}^{m} (b_i - a_i)^2. \quad (2)$$

Decision tree: in this algorithm, *tree* data structure is used to solve classification problems. The leaf nodes depict the class label and the non-leaf nodes denote the attributes of the binary tree. In order to categorize an attribute as leaf node or non-leaf node, statistical methods are used. The features can be extracted by analyzing network traffic of malicious and benign applications. Decision tree is constructed using the feature set. The outcome of decision tree can be divided into malicious and non-malicious depending on the probability values. Figure 24 shows the model of detecting Android malware using decision tree classifier.

Gaussian Naive Bayes: this algorithm can be used for both twofold and manifold class classification problems. The model is developed on the basis of Bayes theorem [143]. The data are captured in the form of probabilities. This algorithm can be enhanced by using Gaussian standard deviation from the training information. The standard probability equation based on *Bayes' theorem* can be written as

$$P\left(\frac{H_t}{D_t}\right) = P\frac{(D_t/H_t) * P(H_t)}{P(D_t)}, \quad (3)$$

where $P(H_t/D_t)$ is posteriori probability of D_t, i.e., probability of occurrence of event D_t after event H_t happened.

Support vector classifier: this algorithm is used for both relapse and classification problems. The data items are sketched in n-dimensional curve, where textitn denotes the total number of attributes that can be plotted in the multidimensional space. Further, a distinct space is drawn to differentiate the classes from each other. For this purpose, hyperplane is used. Hyperplane is the plane having maximum distance between data points. The size of hyperplane depends on number of features. The hyperplane will be a two-dimensional plane if the number of features is three in number. For features greater than three, hyperplane will be multidimensional.

Figure 25 shows two-class datapoints plotted in two dimensions. Figure 25(a) shows two classes with multiple hyperplanes in between. Figure 25(b) shows insertion of hyperplane between the two classes. Using the hyperplane, classification of datapoints can be done to identify support vectors. The datapoints which are closest to hyperplane are known as support vectors.

Several studies have been done in detecting Android malware using machine learning. Some of the recent studies are mentioned below.

Wei et al. [143] proposed a malicious app detection tool called *Androidetect* to monitor instantaneous attacks. The authors analyzed the relationship among system functions, critical permissions, and critical APIs. Further, they combined the system functions to depict the application behaviors and created eigenvectors. Finally, using these eigenvectors, they compared the techniques of Naive Bayesian, J48 decision tree, and application function decision algorithm for efficient identification of malware apps. The proposed tool gives better detection results when compared with the related work of other similar studies, but it also generates sufficient number of false positives.

Kouliaridis and Kambourakis [144] provided a comprehensive survey on Android malware detection using machine learning techniques. The authors gave a detailed overview of the research work done over the past seven years in malware detection using machine learning. Moreover, a comparative analysis of various techniques is also provided in the research work.

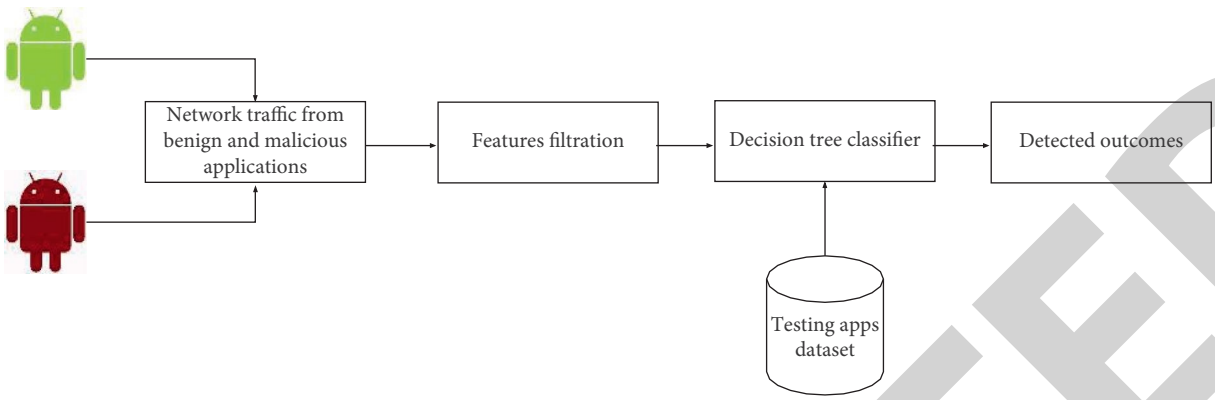

Figure 24: Model of detecting Android malware using decision tree.

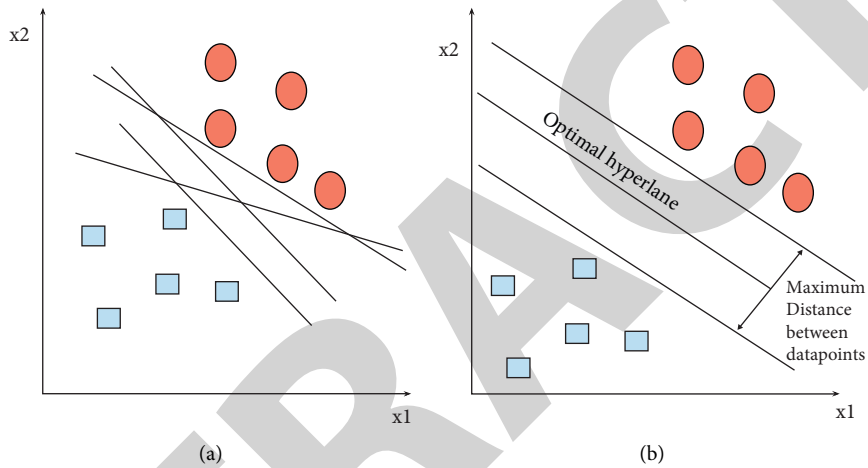

Figure 25: Datapoint separation in support vector classifier hyperplanes. (a) Separation with multiple hyperplanes. (b) Datapoint separation with single hyperplane.

Mahindru and Sangal [145] presented a machine learning-based framework MLDroid to detect Android malware. The authors experimented on more than five million applications to extract various important features. Further, the machine learning-based models were trained and results were presented.

Li et al. [146] introduced *SIGPID*, a malicious app detection system based on permission utilization to deal with the substantial growth of Android malware. The authors developed three levels of pruning by intercepting the permissions data to detect the most useful permissions that can be helpful in differentiating benign and malicious apps. Further, they utilized machine learning-based classification models to group distinct families of malicious and benign apps.

D'Angelo et.al. [31] proposed an approach based on the exploitation of API transitions in the call sequence. The extraction of a subsequence of API calls resulted in a malware classification resistant to evasion techniques. The authors compared the detectors using Markov chain and call sequence algorithms. The study outcomes outperformed various malware detection techniques.

Ficco [30] proposed an approach based on ensemble detection. The author utilized a blend of generic and specialized detectors during the analysis process to enhance the detection randomness and to improve the overall detection rate. Moreover, an alpha-count mechanism is also presented to differentiate the speed of various detectors. This mechanism provides the observation time window length which can affect the detection accuracy.

Narayanan et al. [147] proposed a novel framework named *Context-aware, Adaptive, and Scalable Android Malware Detector (CASANDRA)* to address the problem of high population of Android malware. Besides being scalable and consistent, the proposed framework is adaptive to the evolution of zero-day malware, and with the help of significant features, the apps can be classified as malicious or benign.

Wang et al. [148] implemented an approach named *Mlifdect* to detect Android malware using a blend of information fusion and machine learning. The authors extracted static features from several Android apps and constructed two feature sets. The feature sets served as input vector for the machine learning model. Further, they used

Dempster–Shafer theory-based approach for effective detection. *Mlifdect* produced a good accuracy rate; however, more features can be incorporated to enhance overall detection rate.

Liu et al. [149] proposed a novel feature generation approach to detect malicious applications. The approach uses a blend of context-based and control graph-based feature production techniques. The authors combined features from various applications and generated a vector space. Finally, they trained the malware detector using the random forest technique from the obtained input vector space. The proposed approach is limited to classify malicious and benign applications and does not provide familial classification.

Li et al. [150] proposed an Android threat identification system based on feature fusion. The authors used feature subset selection and principle component analysis for feature dimensionality reduction and random forest algorithm for feature classification. The proposed system produced good detection rate, but more number of features can be used to reduce false positive rate.

Bakour and Ünver [151] presented an image-based malware detection method called VisDroid. Five image-based datasets, each of them consisting of 4850 samples, were constructed and six different machine learning-based models were trained on these datasets. Moreover, the authors also compared the results with similar malware detection methods and observed that their results provided a better accuracy.

6.2. Android Malware Classification and Identification Using Deep Learning. Detection of malware using machine learning techniques needs three basic things such as feature engineering, feature learning, and feature representation. To achieve these, a detailed domain-level knowledge is required [124, 125, 152]. In the worst case, if an attacker finds the information about the features, the malware detector can be exploited easily [127]. Deep learning overcomes these issues as deep learning algorithms automatically extract features from a particular dataset. Selecting the proper classification algorithm for detection purpose is important considering its impact on identification performance and efficiency. In Table 8, the commonly used deep learning algorithms for detecting Android malware are briefly introduced. Figure 26 shows the general architecture of detecting Android using deep learning.

The numerous strengths of various deep learning algorithms discussed in Table 8 motivated the researchers to analyze and detect Android malware using deep learning techniques.

Bakour and Ünver [156] proposed a novel model DeepVisDroid based on hybrid deep learning to detect Android malware samples. The authors converted four dataset application files into grayscale images. Further, the local and global features were extracted from the image regions to feed them as an input to the training model. The proposed model achieved a classification accuracy of more than 98% with less overhead.

Haq et al. [157] suggested a multivector malware detection mechanism based on hybrid deep learning to detect malicious applications. The authors utilized bi-directional LSTM and CNN approaches to detect malware. The proposed model was trained using publicly available datasets.

Millar et al. [158] proposed a new multiview approach based on deep learning to detect Android malware. The proposed approach does not require deep knowledge of malware domain. The authors performed a benchmark comparison of the state-of-the-art models using publicly available Android malware datasets. The experimental results provide a reduction of 77% in false positive rate as compared to other similar malware detectors.

Yuan et al. [159] proposed *DroidDetector*, which automatically detects whether an Android app is malignant or not. The authors used DNN for the learning process. Further, they attributed the malicious apps into their corresponding families. *DroidDetector* provides a good detection accuracy, but it also requires high amount of computational power.

Li et al. [160] proposed a fine-grained deep learning model for detecting Android malware. The proposed methodology can detect fine-grained malware families with 0.1% false positive rate (FPR). The model has very low FPR, but computational complexity and cost of implementation are high.

Zhu et al. [161] proposed *deep flow*, a deep learning technique to detect Android malware on the basis of flow of metadata information in the Android application. Firstly, the authors extracted the features from the APK file and then they examined the flow of critical information in the application. Finally, they constructed a flow graph from the obtained metadata. The proposed technique efficiently detects various malware variants, but it needs more features to detect novel malware variants.

Karbab et al. [162] proposed Maldozer, a deep learning tool which automatically detects Android malware and classifies the malware into their corresponding families. The authors evaluated the proposed tool on a large number of datasets consisting of around 35 K malware apps and 40 K genuine apps. The tool's performance can be enhanced further by reducing the false positive rate.

Kim et al. [163] proposed a multimodal deep learning system to detect Android malware. The authors refined the features using similarity-based feature extraction method [164] for effective malware detection. They implemented the proposed technique on a big dataset of around 40K samples. The model outperformed several traditional models, but it needs complex hardware resources to perform well.

Zhang et al. [165] proposed the *DeepClassifyDroid* model to detect Android malware. The model works by extracting features in the first stage. In the next stage, the features are embedded into feature vector set. Finally, the vector set is used as an input to the deep learning model. The proposed model executes faster than traditional machine learning models. The accuracy can be improved further by

Table 8: Deep learning algorithms in a nutshell [153–155].

Algorithm	Description	Strengths	Weakness
Deep neural network (DNN)	Deep neural networks have the ability to understand the importance of data especially when the data are in bulk amount and then naturally determine the derived importance with new dataset.	(i) Achieved success in various applications. (ii) No requirement of extensive domain-level knowledge.	(i) Learning process sometimes consumes a lot of time.
Convolutional neural network (CNN)	CNNs consist of three layers, i.e., convolutional layer, subsampling layer, and fully connected layer [19]. The convolutional layer uses the convolution method to perform the weight distribution. The subsampling layer is responsible for feature map reduction. Further, multiple fully connected layers and a softmax layer reside on the upper layer to recognize and classify the dataset.	(i) Less number of neuron connections are required as compared to standard neural networks. (ii) Several modifications to CNN have been developed to increase the overall efficiency.	(i) Normally, it requires multiple layers to identify the complete feature hierarchy.
Recurrent neural network (RNN)	RNNs are convenient for data flow handling. RNN deals with the positional memory. The output layers are pipelined to the subsequent input layers.	(i) Modelling timing constraints. (ii) Capability of remembering multiple events.	(i) Vanishing gradient issue affects the overall learning process.
Restricted Boltzmann machine (RBM)	RBMs provide a similar variation over a specific ordering of its inputs. They are generally used to show big dimensional temporal streams like audio or video streams.	(i) Make the samples look like they appear from the data distribution. (ii) Can be used to extract features for training other network models on the top of it.	(i) Difficult to train properly. (ii) Computing similar model is time-consuming.
Deep belief network (DBN)	DBNs consist of different hidden variables and stochastic layers. The top two layers have symmetric relations between them. The bottom layers are directly related to the upper layers.	(i) Provides layer to layer learning approach.	(i) The training stage consumes a huge amount of resources.

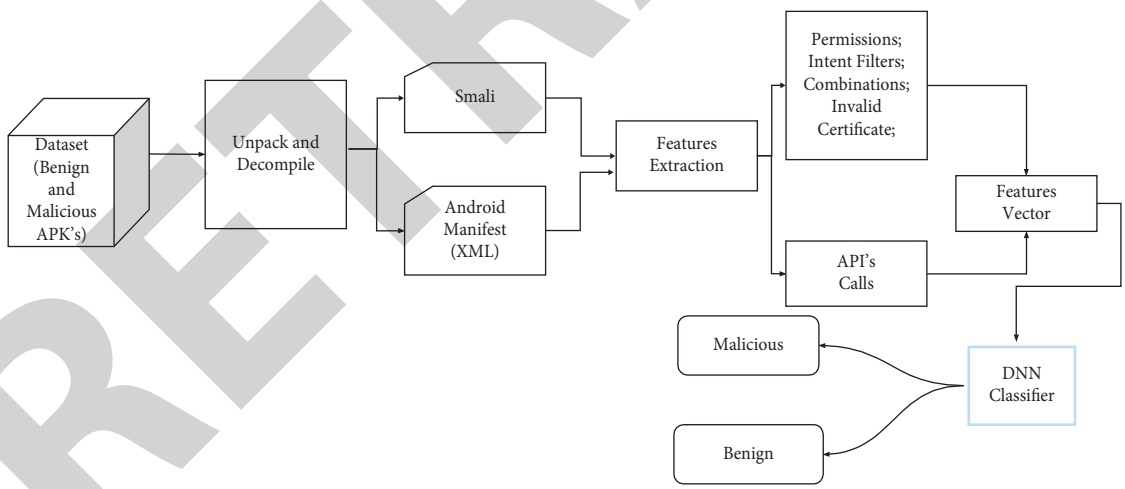

Figure 26: Detecting Android malware using deep learning.

incorporating dynamic feature sets such as dynamic system calls and variable network traffic.

Lu et al. [166] proposed a hybrid malware detection model which is a blend of deep belief network (DBN) and gated recurrent unit (GRU). The authors processed static and dynamic APK features using DBN and GRU, respectively. Finally, the outcomes of DBN and GRU served as an input for neural network to classify the applications. The proposed hybrid approach has a higher detection accuracy when compared with other similar studies, but high computational power is required to implement the model. For this purpose, graphical processing units and cloud instances can be used. Zhang and Li [167] proposed a code semantic feature-based malware detection approach. The authors utilized graph convolutional network (GCN) to extract code semantic features. GCN provides high-level semantic information to classify the applications. The proposed approach can provide better accuracy if more number of features are integrated with the semantic features. Zhou et al. [168] proposed a static

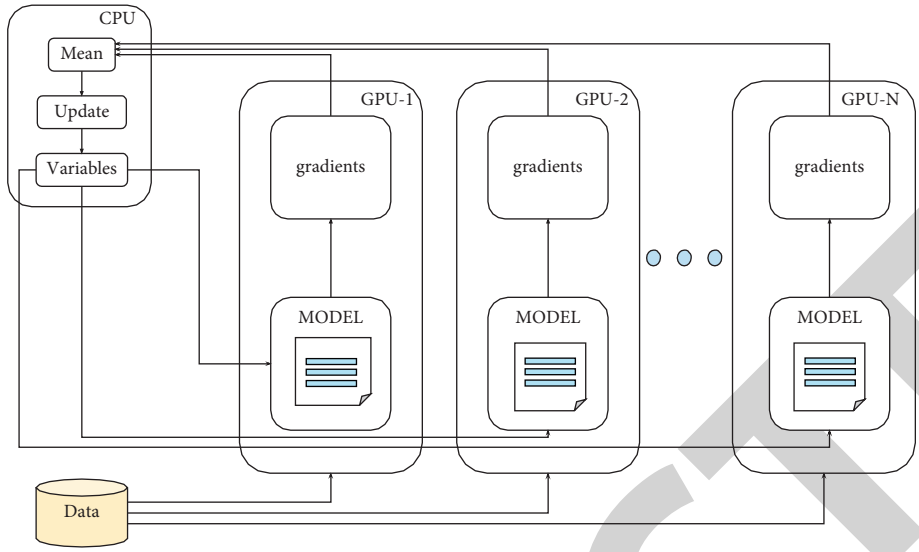

Figure 27: Training phase speedup using multiple GPU instances.

detection approach based on *SIMGRU*. The authors utilized similarity analysis to enhance gated recurrent unit (GRU). The detection model has a better accuracy when compared with other similar models. However, the computational complexity is high and cloud integration is required.

The training algorithm establishes the primary part of deep learning. The quantity of layers separates the deep neural networks from facile ones. The higher the quantity of layers is, the more denser it becomes. Each layer can particularly recognize a specific behavior. As the training algorithms are the basic building blocks for every deep neural networks, these algorithms need to be optimized to provide better accuracy results. The training phase can be accelerated using graphical processing units (GPUs) and cloud instances.

6.2.1. Accelerating the Training Phase Using GPU and Cloud Processing. Training time is the significant performance aspect of machine learning. Distributed computing and GPUs can be used to accelerate the training process. Cloud gives enormous measures of computational strength, and now all significant cloud merchants incorporate GPU-based servers that can effectively be utilized for preparing neural networks on request at reasonable costs. GPUs can process several computations in parallel as they contain huge number of cores. The complex computations can be done in less amount of time, and hence huge dataset models can be trained effectively with the help of multiple GPUs. Figure 27 demonstrates the usage of multiple GPU instances to speed up the training process.

Cloud merchants can provide more than thousands of GPU cores running in parallel and their GPU instances can be additionally utilized for advanced learning phases. Since GPUs are little expensive, cloud hosting organizations provide an option of using GPU services on their cloud instances. These are less expensive than purchasing multiple GPUs. Figure 28 shows multiple GPUs deployed on different cloud instances. The cloud instance manager manages the multiple cloud instance workers. The GPU cores deployed at several cloud instances have supreme computational power which is required to train big datasets.

6.3. Comparison of Android Malware Detection and Analysis Techniques Based on Objectives and Methodology. Several research scholars and industry people have proposed different tools and techniques for detecting and analyzing Android malware. In this section, a summary of some of the famous detection and analysis tools is demonstrated in Table 9. Detection methods have been categorized according to the following parameters:

(1) Objective, which can be detection, analysis, and/or estimation.

(2) Methodology, which can be static, dynamic, signature based, system call based, virtual emulation based, machine learning based, and deep learning based.

Some of the tools or techniques combine two or more methodology parameters to enhance the detection rate. However, the time taken in detecting and analyzing phases will also increase significantly. Therefore, the parameters need to be chosen carefully in order to balance the performance and time trade-off.

7. Proposed Model for Android Malware Detection and Familial Classification

This paper proposes a model which comprises three primary modules, including static, dynamic, and result investigations, as shown in Figure 29. Data are preprocessed in order to remove the unnecessary features. After getting the useful data, deep transfer learning can be applied in the first phase to classify benign and malicious applications. Deep transfer

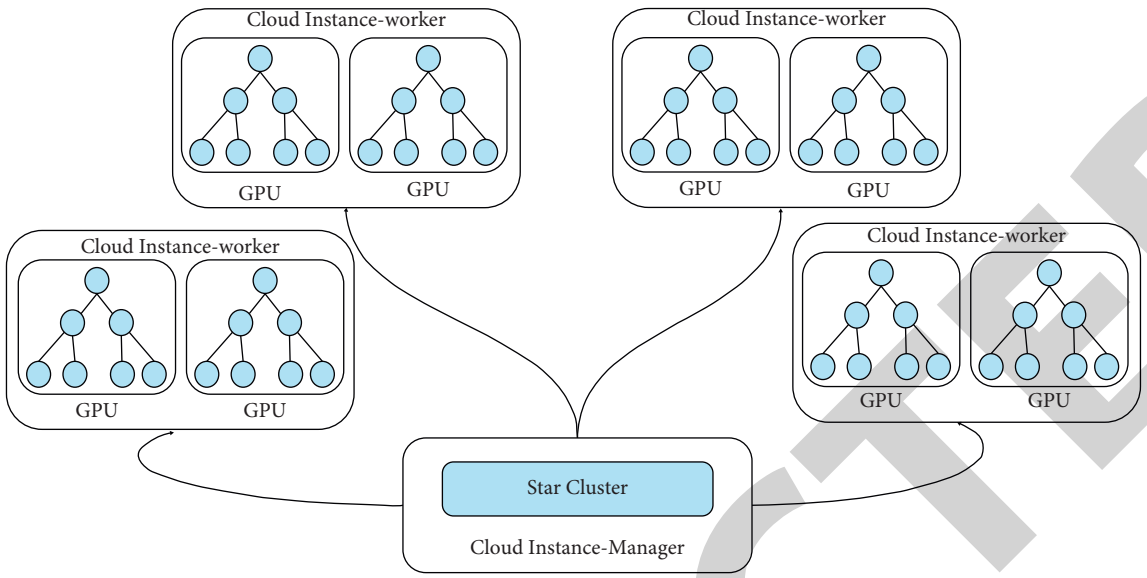

Figure 28: Deploying GPU cores on multiple cloud instances.

Table 9: Comparative summary of Android malware identification and analysis tools/techniques on the basis of objectives and methodologies.

Tool/technique	Objective					Methodology				
	Detection	Analysis	Estimation	Static	Dynamic	Signature based	System calls/API based	Virtual machine emulation	Machine learning based	Deep learning based
APKInspector [169]		✓		✓						
DexRay [170]	✓	✓	✓		✓	✓				✓
PetaDroid [171]	✓		✓		✓	✓	✓			✓
Gsdroid [172]	✓	✓	✓		✓		✓	✓	✓	
SEDMDroid [168]	✓	✓			✓		✓	✓	✓	✓
Famd [173]		✓	✓			✓	✓	✓		
Maldozer [162]	✓	✓			✓					✓
AdDroid [174]	✓	✓		✓			✓			
JOWMDroid [175]	✓	✓				✓	✓	✓	✓	
DeepAMD [176]	✓	✓		✓			✓		✓	✓
DAMBA [177]	✓	✓	✓	✓	✓	✓	✓		✓	
AdMAt [178]	✓	✓		✓	✓			✓	✓	
GDroid [179]	✓	✓	✓	✓				✓	✓	
KronoDroid [180]	✓	✓	✓		✓	✓		✓		
ProDroid [181]	✓	✓		✓				✓	✓	
Hawk [182]	✓	✓	✓	✓	✓			✓	✓	
LinRegDroid [183]	✓	✓		✓	✓			✓	✓	✓
FAM [184]	✓	✓	✓	✓				✓	✓	✓
MDTA [185]	✓	✓	✓					✓	✓	
DroidFax [186]	✓	✓	✓		✓	✓	✓	✓	✓	
MamaDroid [187]	✓	✓	✓		✓	✓	✓			
DroidEvolver [188]	✓	✓	✓	✓	✓	✓	✓			
AndroCT [189]	✓	✓	✓	✓	✓	✓	✓			

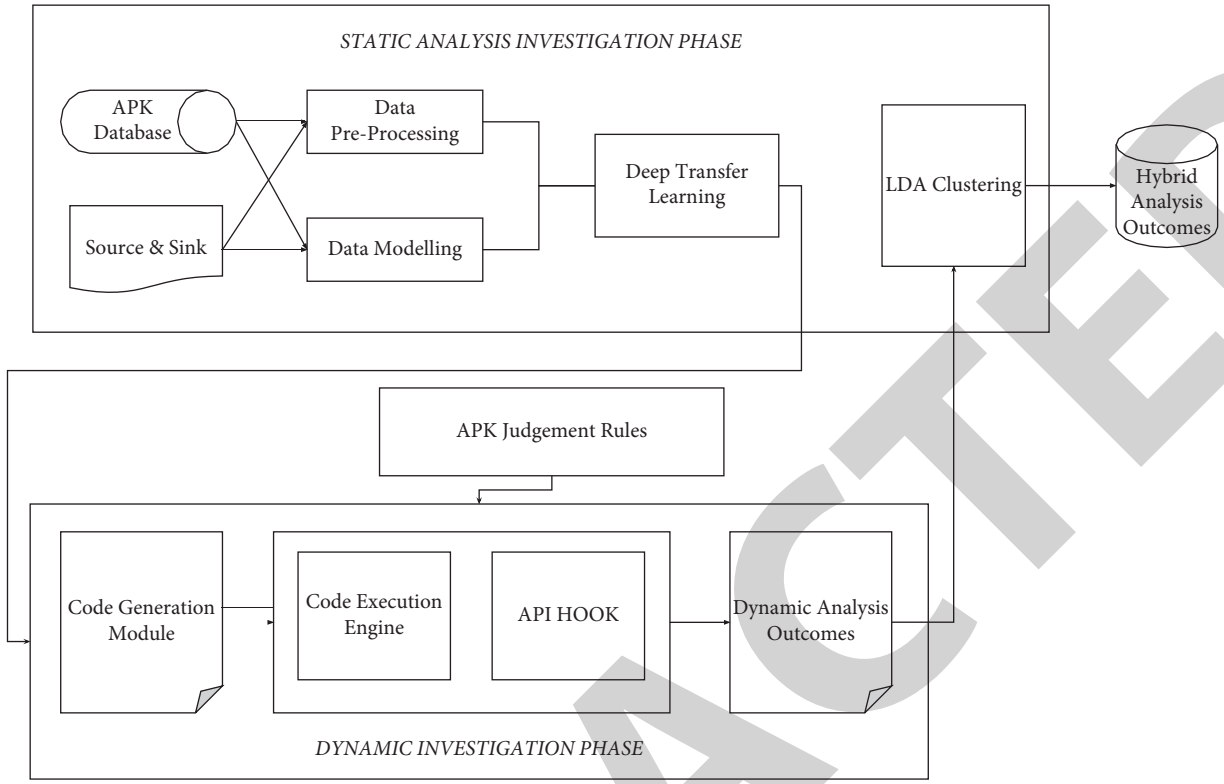

Figure 29: Proposed model to detect and characterize Android malware.

learning provides good accuracy and does not require high computational power to train complex dataset comprising millions of features. In the second phase, dynamic investigation can be done to remove any false positives. In this phase, the runtime behavior of detected applications is monitored.

Finally, after getting the unique malicious samples, LDA algorithm can be used to group the common Android malware into their respective families according to their behavior and keywords. The resulting hybrid outcome process is the blend of static and dynamic investigation results. The proposed model can train any type of dataset including complex dataset because lightweight version of deep learning, i.e., deep transfer learning technique, is used for classification.

8. Conclusion and Future Work

A comprehensive survey of Android security architecture, core features and security aspects of upcoming 6G mobile networks, stealth techniques employed by various Android malware, and several approaches used to detect and analyze these malware is presented in this paper. There are various security issues present in the current mobile network generations, and since the future Android devices will include 6G network for mobile data communication, the researchers need to come up with more advanced and secure 6G network features. The core features and security concerns of current mobile networks and 6G mobile networks are demonstrated in Section 3. The current approaches used to detect the Android malware are described in Section 4. Each approach can be used individually, but they have their own limitations. The static analysis approach is inefficient against code encryption and obfuscation techniques. Dynamic analysis can be escaped by using various anti-emulation methods. Detection based on machine learning is effective but requires domain-level knowledge. Moreover, the malware can escape the detection mechanism if the malware developer obtains the feature sets. Deep learning models are efficient, but they require lot of computational power. Hence, no single technique or approach is sufficient to completely secure the Android devices from malicious attacks. A hybrid model is proposed in this paper which uses lightweight deep transfer learning technique to overcome the problem of high computational cost. The proposed model can detect and characterize Android malware with a good accuracy rate. However, it also requires overcoming several new challenges such as dynamic feature inclusion and optimal layer management during transfer learning phase. Our further study will look at these challenges and expand our study to explore novel methods to fulfill current challenges and maintain resource and computational requirement balance. Some of the major focus areas will be efficient implementation of Android hardware security to detect malicious codes embedded in the 32 bit, 64 bit ARM, and x86-64 architectures.

Data Availability

The data used to support the findings of this study can be accessed from the following websites: https://www.sec.cs.tu-bs.de/~danarp/drebin/; https://www.unb.ca/cic/datasets/andmal2017.html.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] "Gartner Smartphone Reports ," Gartner, 2019, https://www.gartner.com/en/newsroom/press-releases/2019-11-26-gartner-says-global-smartphone-demand-was-weak-in-thi.

[2] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces, 2012 marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pp. 317–326, ACM, San Antonio, TX, USA, February 2012.

[3] R. Potharaju, M. Rahman, and B. Carbunar, "A longitudinal study of google play," *IEEE Transactions on computational social systems*, vol. 4, no. 3, pp. 135–149, 2017.

[4] "A Look at Google Bouncer." Trendlabs-Security-Intelligence," 2019, https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/ accessed on.

[5] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012*, vol. 95, p. 110, New York, 2012.

[6] E. Cunningham, "Keeping You Safe with Google Play Protect," 2017, https://blog.google/products/android/google-play-protect/.

[7] T. Cho, H. Kim, and J. H. Yi, "Security assessment of code obfuscation based on dynamic monitoring in android things," *IEEE Access*, vol. 5, pp. 6361–6371, 2017.

[8] R. Binns, U. Lyngs, M. Van Kleek, J. Zhao, T. Libert, and N. Shadbolt, "Third party tracking in the mobile ecosystem," in *Proceedings of the 10th ACM Conference on Web Science*, pp. 23–31, ACM, Amsterdam, Netherlands, May 2018.

[9] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, pp. 653–656, ACM, Austin, TX, USA, May 2016.

[10] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, pp. 1011–1015, IEEE, Sanya, China, December 2011.

[11] M. Zheng, M. Sun, and J. C. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," in *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 163–171, IEEE, Melbourne, VIC, Australia, July 2013.

[12] C. Jarabek, D. Barrera, and J. Aycock, "Thinav: truly lightweight mobile cloud-based anti-malware," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 209–218, ACM, Orlando, FL, USA, December 2012.

[13] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security," *USENIX security symposium*, vol. 2, p. 2, 2011.

[14] E. Medvet and F. Mercaldo, "Exploring the usage of topic modeling for android malware static analysis," in *Proceedings of the 2016 11th International Conference on Availability, Reliability and Security (ARES)*, pp. 609–617, IEEE, Salzburg, Austria, August 2016.

[15] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the 7th European Workshop on System Security*, vol. 5, ACM, Amsterdam, Netherlands, April 2014.

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

[17] S. K. Dash, G. Suarez-Tangil, S. Khan et al., "Droidscribe: classifying android malware based on runtime behavior," in *Proceedings of the 2016 IEEE Security and Privacy Workshops (SPW)*, pp. 252–261, IEEE, San Jose, CA, USA, May 2016.

[18] N. Zhang, Y.-a. Tan, C. Yang, and Y. Li, "Deep learning feature exploration for android malware detection," *Applied Soft Computing*, vol. 102, Article ID 107069, 2021.

[19] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, Article ID 46717, 2019.

[20] A. Alzubaidi, "Recent advances in android mobile malware detection: a systematic literature review," *IEEE Access*, vol. 9, 2021.

[21] 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 *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, pp. 309–320, Arizona, USE, March 2017.

[22] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2013.

[23] M. La Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE communications surveys & tutorials*, vol. 15, no. 1, pp. 446–471, 2012.

[24] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[25] H. Cai, "Embracing mobile app evolution via continuous ecosystem mining and characterization," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 31–35, Madrid, Spain, July 2020.

[26] H. Cai, X. Fu, and A. Hamou-Lhadj, "A study of run-time behavioral evolution of benign versus malicious apps in android," *Information and Software Technology*, vol. 122, Article ID 106291, 2020.

[27] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "Mr-droid: a scalable and prioritized analysis of inter-app communication risks," in *Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW)*, pp. 189–198, IEEE, San Jose, CA, USA, May 2017.

[28] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.

[29] X. Fu and H. Cai, "On the deterioration of learning-based malware detectors for android," in *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 272-273, IEEE, Montreal, QC, Canada, May 2019.

[30] M. Ficco, "Malware analysis by combining multiple detectors and observation windows," *IEEE Transactions on Computers*, vol. 71, 2021.

[31] G. D'Angelo, M. Ficco, and F. Palmieri, "Association rule-based malware classification using common subse- quences of api calls," *Applied Soft Computing*, vol. 105, Article ID 107234, 2021.

[32] A. Razgallah, R. Khoury, S. Hallé, and K. Khanmohammadi, "A survey of malware detection in android apps: recommendations and perspectives for future research," *Computer Science Review*, vol. 39, Article ID 100358, 2021.

[33] 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, Article ID 124579, 2020.

[34] E. C. Bayazit, O. K. Sahingoz, and B. Dogan, "Malware detection in android systems with traditional machine learning models: a survey," in *Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pp. 1–8, IEEE, Ankara, Turkey, June 2020.

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

[36] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: review, taxonomy & future directions," *Future Generation Computer Systems*, vol. 97, pp. 887–909, 2019.

[37] P. Agrawal and B. Trivedi, "A survey on android malware and their detection techniques," in *Proceedings of the 2019 IEEE In- ternational Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pp. 1–6, IEEE, Coimbatore, India, February 2019.

[38] A. A. A. Samra, H. N. Qunoo, F. Al-Rubaie, and H. El-Talli, "A survey of static android malware detection techniques," in *Proceedings of the 2019 IEEE 7th Palestinian International Conference on Electrical and Computer Engineering (PICECE)*, pp. 1–6, IEEE, Gaza, Palestine, March 2019.

[39] E. J. Alqahtani, R. Zagrouba, and A. Almuhaideb, "A survey on android malware detection techniques using machine learning algorithms," in *Proceedings of the 2019 6th International Conference on Software Defined Systems (SDS)*, pp. 110–117, IEEE, Rome, Italy, June 2019.

[40] R. Zachariah, K. Akash, M. S. Yousef, and A. M. Chacko, "Android malware detection a survey," in *Proceedings of the 2017 IEEE international conference on circuits and systems (ICCS)*, pp. 238–244, IEEE, Thiruvananthapuram, India, December 2017.

[41] S. K. Muttoo and S. Badhani, "Android malware detection: state of the art," *International Journal of Information Technology*, vol. 9, no. 1, pp. 111–117, 2017.

[42] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, 2016.

[43] R. J. Blainey, M. K. Gschwind, J. L. McInnes, and S. J. Munroe, "Compiling code for an enhanced application binary interface (abi) with decode time instruction optimization," *US Patent*, vol. 8, 2014.

[44] *Android Application Binary Interface*, Android ABI Guide, 2019, https://developer.android.com/ndk/guides/abis.

[45] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh, "Cider," *ACM SIGARCH - Computer Architecture News*, vol. 42, no. 1, pp. 367–382, 2014.

[46] O. Cinar, *Android Apps with Eclipse*, Apress, NY, USA, 2012.

[47] Android Security, "Android Security Overview," 2019, https://security.googleblog.com/search/label/android%20security.

[48] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smart- phones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 347–356, 2010.

[49] A. Permissions, "Android Application Permissions Overview," 2019, https://developer.android.com/guide/topics/permissions/overview.

[50] J. Jenkins and H. Cai, "Icc-inspect: supporting runtime inspection of android inter-component communications," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pp. 80–83, Gothenburg, Sweden, June 2018.

[51] M. Dilhara, H. Cai, and J. Jenkins, "Automated detection and repair of incompatible uses of runtime permissions in android apps," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pp. 67–71, Gothenburg, Sweden, May 2018.

[52] S. Karthick and S. Binu, "Android security issues and solutions," in *Proceedings of the 2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, pp. 686–689, IEEE, Bengaluru, India, February 2017.

[53] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, pp. 1–28, 2020.

[54] K. O. Elish, H. Cai, D. Barton, D. Yao, and B. G. Ryder, "Identifying mobile inter-app communication risks," *IEEE Transactions on Mobile Computing*, vol. 19, no. 1, pp. 90–102, 2018.

[55] Y. Kouraogo, K. Zkik, G. Orhanou et al., "Attacks on android banking applications," in *Proceedings of the 2016 International Conference on Engineering & MIS (ICEMIS)*, pp. 1–6, IEEE, Agadir, Morocco, September 2016.

[56] S. Acharya, U. Rawat, and R. Bhatnagar, "Android rogue application detection using image resemblance and reduced lda," *Advances in Intelligent Systems and Computing*, in *Proceedings of the International Conference on Advanced Machine Learning Technologies and Applications*, pp. 277–287, Springer, Jaipur, India, February 2020.

[57] A. R. Javed, M. O. Beg, M. Asim, T. Baker, and A. H. Al-Bayatti, "Alphalogger: detecting motion-based side- channel attack using smartphone keystrokes," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–14, 2020.

[58] D. J. Tan, T.-W. Chua, and V. L. Thing, "Securing android: a survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–45, 2015.

[59] H. Meng, V. L. L. Thing, Y. Cheng, Z. Dai, and L. Zhang, "A survey of android exploits in the wild," *Computers & Security*, vol. 76, pp. 71–91, 2018.

[60] P. Yan and Z. Yan, "A survey on dynamic mobile malware detection," *Software Quality Journal*, vol. 26, no. 3, pp. 891–919, 2018.

[61] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, and privacy leaks: a longitudinal study of pii leaks across android app versions," *Network and Distributed System Security Symposium, Internet Society*, 2018, https://pdfs.semanticscholar.org/5809/5ad29b6f8fb51a1a422fa9d58e63d1f02f77.pdf.

[62] M. Kaur, D. Singh, K. Sun, and U. Rawat, "Color image encryption using non-dominated sorting genetic algorithm with local chaotic search based 5D chaotic map," *Future Generation Computer Systems*, vol. 107, pp. 333–350, 2020.

[63] H. Cai and J. Jenkins, "Towards sustainable android malware detection," in *Proceedings of the 40th International*

Conference on Software Engineering: Companion Proceeedings, pp. 350-351, Gothenburg, Sweden, May 2018.

[64] "Kaspersky mobile threat report," 2019, https://securelist.com/threat-category/mobile-threats/.

[65] "Lookout Mobile Security Report," 2012, https://www.lookout.com/company/media-center/press-releases/state-of-mobile-security-2012 Last Accessed.

[66] V. Avdiienko, K. Kuznetsov, A. Gorla et al., "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 426–436, IEEE, Florence, Italy, May 2015.

[67] R. Fedler, J. Schu¨tte, and M. Kulicke, "On the effectiveness of malware protection on android," *Fraunhofer AISEC*, vol. 45, 2013.

[68] Z. Xu, "Android Installer Hijacking Vulnerability Could Expose Android Users to Malware," 2015, https://researchcenter.paloaltonetworks.com/2015/03/android-installer-hijacking-vulnerability-could-expose-android-users-to-malware/.

[69] W.-S. Chun and D.-W. Park, "Malicious code hiding android app's distribution and hacking attacks and inci- dent analysis," in *Proceedings of the 2012 8th International Conference on Information Science and Digital Content Technology (ICIDT2012)*, vol. 3, pp. 686–689, IEEE, Jeju, June 2012.

[70] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, "Android malware detection & protection: a survey," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, pp. 463–475, 2016.

[71] F. Shahzad, M. A. Akbar, and M. Farooq, "A Survey on Recent Advances in Malicious Applications Analysis and Detection Techniques for Smartphones," National University of Computer & Emerging Sciences, Islamabad, Pakistan, Tech. Rep, 2012.

[72] Y. Zhou, Z. Wang, W. Zhou, X. Jiang, and you Hey, "Get off of my market: detecting malicious apps in official and alternative android markets," *NDSS*, vol. 25, pp. 50–52, 2012.

[73] A. F. Abdul Kadir, N. Stakhanova, and A. A. Ghorbani, "Android botnets: what urls are telling us," *Network and System Security*, Springer, in *Proceedings of the International Conference on Network and System Security*, pp. 78–91, November 2015.

[74] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for android ran- somware," in *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1338–1343, IEEE, New York, NY, USA, August 2015.

[75] V. G. Shankar, M. Jangid, B. Devi, and S. Kabra, "Mobile big data: malware and its analysis," in *Proceedings of the 1st International Conference on Smart System, Innovations and Computing*, pp. 831–842, Springer, Singapore, January, 2018.

[76] T. Strazzere and T. Wyatt, "Geinimi trojan technical teardown," *Lookout Mobile Security*, https://www.androidcommunity.com/wp-content/uploads/2011/01/Geinimi_Trojan_Teardown.pdf, 2011.

[77] Y. Zhou and X. Jiang, "An analysis of the anserverbot trojan," North Carolina State University, Tech. Rep, 2011.

[78] A. Rehman Javed, Z. Jalil, S. Atif Moqurrab, S. Abbas, and X. Liu, "Ensemble Adaboost Classifier for Accu- Rate and Fast Detection of Botnet Attacks in Connected Vehicles," *Transactions on Emerging Telecommunications Technologies*, Article ID e4088, 2020.

[79] C. Orthacker, P. Teufl, S. Kraxberger et al., "Android security permissions–can we trust them?" in *Proceedings of the International Conference on Security and Privacy in Mobile Information and Communication Systems*, pp. 40–51, Springer, Aalborg, Denmark, May 2011.

[80] Z. Man, Y. Lin, and S. Zou, "Method for preventing a mobile communication device from leaking secret and system thereof," *US Patent*, vol. 8, 2014.

[81] J. Cho, G. Cho, and H. Kim, "Keyboard or keylogger?: a security analysis of third-party keyboards on android," in *Proceedings of the 2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pp. 173–176, IEEE, September 2015.

[82] "Google Official Application Store," 2019, https://play.google.com/store?hl=en.

[83] A. Mart´ın, J. Hernandez-Castro, and D. Camacho, "An in-depth study of the jisut family of android ransomware," *IEEE Access*, vol. 6, pp. 57205–57218, 2018.

[84] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 2012 IEEE symposium on security and privacy*, pp. 95–109, IEEE, San Francisco, CA, USA, May 2012.

[85] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "A new android malware detection approach using bayesian classification," in *Proceedings of the 2013 IEEE 27th international conference on advanced information networking and applications (AINA)*, pp. 121–128, IEEE, Barcelona, Spain, March 2013.

[86] S. Liang and X. Du, "Permission-combination-based scheme for android mobile malware detection," in *Proceedings of the 2014 IEEE international conference on communications (ICC)*, pp. 2301–2306, IEEE, Sydney, NSW, Australia, June 2014.

[87] A. Reina, A. Fattori, and L. Cavallaro, "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors," *EuroSec*, https://www.artificialstudios.org/alessandro.reina/pubs/eurosec13.pdf, 2013.

[88] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: a text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[89] Y. Fang, Y. Gao, F. Jing, and L. Zhang, "Android malware familial classification based on dex file section features," *IEEE Access*, vol. 8, Article ID 10614, 2020.

[90] F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, and G. Vaglini, "Model checking to detect the hummingbad malware," in *Intelligent Distributed Computing XIII*pp. 485–494, Springer, Berlin, Germany, 2019.

[91] "Kaspersky mobile threat report," 2019, https://go.kaspersky.com/rs/802-IJN-240/images/KSB_statistics_2018_eng_final.pdf.

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

[93] J. Samhi, K. Allix, T. F. Bissyand´e, and J. Klein, "A First Look at Android Applications in Google Play Related to Covid-19," arXiv preprint arXiv:2006.11002, 2020.

[94] A. R. Javed, M. Usman, S. U. Rehman, M. U. Khan, and M. S. Haghighi, "Anomaly detection in automated vehicles using multistage attention-based convolutional neural network," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, 2020.

[95] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proceedings of the 2010 5th International Conference on Malicious and Unwanted Software*, pp. 55–62, IEEE, Nancy, France, October 2010.

[96] Q. Jerome, K. Allix, R. State, and T. Engel, "Using opcode-sequences to detect malicious android applications," in *Proceedings of the 2014 IEEE International Conference on Communications (ICC)*, pp. 914–919, IEEE, Sydney, NSW, Australia, June 2014.

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

[98] H. Cai and B. G. Ryder, "Artifacts for dynamic analysis of android apps," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, September 2017.

[99] "Dexguard," 2019, http://www.saikoa.com/dexguard.

[100] Y. Piao, J.-H. Jung, and J. H. Yi, "Server-based code obfuscation scheme for apk tamper detection," *Security and Communication Networks*, vol. 9, no. 6, pp. 457–467, 2016.

[101] H. Cai and B. Ryder, "A longitudinal study of application structure and behaviors in android," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2934–2955, 2020.

[102] M. Zheng, P. P. Lee, and J. C. Lui, "Adam: an automatic and extensible platform to stress test android anti- virus systems," in *Proceedings of the International conference on detection of intrusions and malware, and vulnerability assessment*, pp. 82–101, Springer, Heraklion, Greece, July 2012.

[103] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pp. 329–334, ACM, Hangzhou, China, May 2013.

[104] "Androguard," 2019, https://code.google.com/androguard.

[105] G. Suarez-Tangil and G. Stringhini, "Eight Years of Rider Measurement in the Android Malware Ecosystem: Evolution and Lessons Learned," arXiv preprint arXiv:1801.08115, 2018.

[106] F. Fasano, F. Martinelli, F. Mercaldo, and A. Santone, "Android run-time permission exploitation user awareness by means of formal methods," in *ICISSP*, pp. 804–814, 2020, https://www.scitepress.org/Papers/2020/93723/93723.pdf.

[107] F. Wei, S. Roy, X. Ou, and A. Robby, "Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329–1341, ACM, Scottsdale, AZ, USA, November 2014.

[108] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pp. 15–26, ACM, Chicago, Illinois, October 2011.

[109] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, pp. 152–159, ACM, Aksaray, Turkey, November 2013.

[110] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspec- tive combining risks and benefits," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pp. 13–22, ACM, New York, NY, USA, June 2012.

[111] J. Jenkins and H. Cai, "Dissecting android inter-component communications via interactive visual explorations," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 519–523, IEEE, Shanghai, China, September 2017.

[112] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "Puma: permission usage to detect malware in android," in *Proceedings of the International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, pp. 289–298, Springer, Ostrava, Czech Republic, September 2013.

[113] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 235–245, ACM, Chicago, Illinois, USA, November 2009.

[114] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *ICT Systems Security and Privacy Protection*, pp. 142–155, Springer, Berlin, Germany, 2014.

[115] M. A. Atici, S. Sagiroglu, and I. A. Dogru, "Android malware analysis approach based on control flow graphs and machine learning algorithms," in *Proceedings of the 2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, pp. 26–31, IEEE, Little Rock, AR, USA, April 2016.

[116] H. P. Enterprise, "Fortify static code analyzer," *Retrieved on*, vol. 9, 2016.

[117] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot," arXiv preprint arXiv:1205.3576, 2012.

[118] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," in *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, pp. 66–72, IEEE, Fajardo, PR, USA, October 2011.

[119] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," in *Proceedings of the 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, pp. 1–6, IEEE, London, UK, June 2017.

[120] C. Da, Z. Hongmei, and Z. Xiangli, "Detection of android malware security on system calls," in *Proceedings of the 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pp. 974–978, IEEE, Xi'an, China, October 2016.

[121] M. Dimjasevic, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of android malware detection based on system calls," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pp. 1–8, New Orleans, LA, US, March 2016.

[122] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," *Presented as part of the 21st USENIX Security Symposium*, vol. 12, pp. 569–584, 2012.

[123] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""Andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[124] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in

Proceedings of the Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, February 2018.

[125] M. Krč´al, O. Švec, M. B´alek, and O. Jašek, "Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only," in *Proceedings of the ICLR 2018 Workshop Acceptance Decision*, Vancouver, Canada, May 2018.

[126] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers & Security*, vol. 77, pp. 578–594, 2018.

[127] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, *Evading Machine Learning Malware Detection*, Black Hat, Isanti, Minnesota, 2017.

[128] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM - Computer Communication Review,* vol. 44, pp. 371-372, ACM, 2014.

[129] L. Nataraj, D. Kirat, B. Manjunath, and G. Vigna, "Sarvam: search and retrieval of malware," in *Proceedings of the Annual Computer Security Conference (ACSAC) Worshop on Next Generation Malware Attacks and Defense (NGMAD)*, New Orleans, Louisiana, December 2013.

[130] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Proceedings of the 2012 European Intelligence and Security Informatics Conference*, pp. 141–147, IEEE, Odense, Denmark, August 2012.

[131] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale," in *Proceedings of the 2013 9th international wireless communications and mobile computing conference (IWCMC)*, pp. 1666–1671, IEEE, Sardinia, Italy, July 2013.

[132] L. Nataraj, *A Signal Processing Approach to Malware Analysis*, University of California, Santa Barbara, 2015.

[133] W. Li, J. Ge, and G. Dai, "Detecting malware for android platform: an svm-based approach," in *Proceedings of the 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pp. 464–469, IEEE, New York, NY, USA, November 2015.

[134] T. R. Patil and S. Sherekar, "Performance analysis of naive bayes and j48 classification algorithm for data classification," *International Journal of Computer Science and Applications*, vol. 6, no. 2, pp. 256–261, 2013.

[135] N. Bhargava, G. Sharma, R. Bhargava, and M. Mathuria, "Decision tree analysis on j48 algorithm for data mining," *Proceedings of International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 6, 2013.

[136] K. Xin, G. Li, Z. Qin, and Q. Zhang, "Malware detection in smartphone using hidden Markov model," in *Proceedings of the 2012 fourth international conference on multimedia information networking and security*, pp. 857–860, IEEE, Nanjing, China, November 2012.

[137] Y. Chen, M. Ghorbanzadeh, K. Ma, C. Clancy, and R. McGwier, "A hidden Markov model detection of malicious android applications at runtime," in *Proceedings of the 2014 23rd Wireless and Optical Communication Conference (WOCC)*, pp. 1–6, IEEE, Newark, NJ, USA, May 2014.

[138] Z. Aung and W. Zaw, "Permission-based android malware detection," *International Journal of Scientific & Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.

[139] M. S. Alam and S. T. Vuong, "Random forest classification for detecting android malware," in *Proceedings of the 2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*, pp. 663–669, IEEE, Beijing, China, August 2013.

[140] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu, "Performance evaluation on permission-based detection for android malware," in *Advances in Intelligent Systems and Applications - Volume 2,* vol. 2, pp. 111–120, Springer, 2013.

[141] M. Ficco, "Comparing api call sequence algorithms for malware detection," *Advances in Intelligent Systems and Computing*, in *Proceedings of the Workshops of the International Conference on Advanced Information Networking and Applications*, pp. 847–856, Springer, Toronto, Canada, May 2020.

[142] G. D'Angelo, M. Ficco, and F. Palmieri, "Malware detection in mobile environments based on autoencoders and api-images," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 26–33, 2020.

[143] L. Wei, W. Luo, J. Weng, Y. Zhong, X. Zhang, and Z. Yan, "Machine learning-based malicious application detection of android," *IEEE Access*, vol. 5, Article ID 25591, 2017.

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

[145] A. Mahindru and A. L. Sangal, "MLDroid-framework for Android malware detection using machine learning techniques," *Neural Computing & Applications*, vol. 33, no. 10, pp. 5183–5240, 2021.

[146] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[147] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "Context-aware, adaptive, and scalable android malware detection through online learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 157–175, 2017.

[148] X. Wang, D. Zhang, X. Su, and W. Li, "Mlifdect: android malware detection based on parallel machine learning and information fusion," *Security and Communication Networks*, vol. 2017, Article ID 6451260, 14 pages, 2017.

[149] X. Liu, Q. Lei, and K. Liu, "A graph-based feature generation approach in android malware detection with machine learning techniques," *Mathematical Problems in Engineering*, vol. 2020, Article ID 3842094, 15 pages, 2020.

[150] J. Li, Z. Wang, T. Wang, J. Tang, Y. Yang, and Y. Zhou, "An android malware detection system based on feature fusion," *Chinese Journal of Electronics*, vol. 27, no. 6, pp. 1206–1213, 2018.

[151] K. Bakour and H. M. Ünver, "VisDroid: android malware classification based on local and global image features, bag of visual words and machine learning techniques," *Neural Computing & Applications*, vol. 33, no. 8, pp. 3133–3153, 2021.

[152] H. S. Basavegowda and G. Dagnew, "Deep learning approach for microarray cancer data classification," *CAAI Transactions on Intelligence Technology*, vol. 5, no. 1, pp. 22–33, 2020.

[153] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, Article ID 53040, 2019.

[154] J. Kim, Y. Ban, E. Ko, H. Cho, and J. H. Yi, "Mapas: a practical deep learning-based android malware detection system," *International Journal of Information Security*, pp. 1–14, 2022.

[155] I. Almomani, A. Alkhayer, and W. El-Shafai, "An automated vision-based deep learning model for efficient detection of android malware attacks," *IEEE Access*, vol. 10, 2022.

[156] K. Bakour and H. M. Ünver, "Deepvisdroid: android malware detection by hybridizing image-based features with deep learning techniques," *Neural Computing & Applications*, vol. 33, no. 18, Article ID 11499, 2021.

[157] I. U. Haq, T. A. Khan, and A. Akhunzada, "A dynamic robust dl-based model for android malware detection," *IEEE Access*, vol. 9, Article ID 74510, 2021.

[158] S. Millar, N. McLaughlin, J. Martinez del Rincon, and P. Miller, "Multi-view deep learning for zero-day android malware detection," *Journal of Information Security and Applications*, vol. 58, Article ID 102718, 2021.

[159] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[160] D. Li, Z. Wang, L. Li, Z. Wang, Y. Wang, and Y. Xue, "Fgdetector: fine-grained android malware detection," in *Proceedings of the 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pp. 311–318, IEEE, Shenzhen, China, June 2017.

[161] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, "Deepflow: deep learning-based malware detection by mining android application for abnormal usage of sensitive data," in *Proceedings of the 2017 IEEE symposium on computers and commu- nications (ISCC)*, pp. 438–443, IEEE, Heraklion, July 2017.

[162] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[163] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[164] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted con- textual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pp. 1105–1116, ACM, Scottsdale, AZ, USA, November 2014.

[165] Y. Zhang, Y. Yang, and X. Wang, "A novel android malware detection approach based on convolutional neural network," in *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, pp. 144–149, Guiyang, China, March 2018.

[166] T. Lu, Y. Du, L. Ouyang, Q. Chen, and X. Wang, "Android malware detection based on a hybrid deep learning model," *Security and Communication Networks*, vol. 2020, Article ID 8863617, 11 pages, 2020.

[167] Y. Zhang and B. Li, "Malicious code detection based on code semantic features," *IEEE Access*, vol. 8, 2020.

[168] H. Zhou, X. Yang, H. Pan, and W. Guo, "An android malware detection approach based on simgru," *IEEE Access*, vol. 8, Article ID 148404, 2020.

[169] F. Jaafar, G. Singh, and P. Zavarsky, "An analysis of android malware behavior," in *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 505–512, IEEE, Lisbon, Portugal, July 2018.

[170] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé, and J. Klein, "Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode," *Deployable Machine Learning for Security Defense*, in *Proceedings of the International Workshop on Deployable Machine Learning for Security Defense*, pp. 81–106, Springer, Virtual Event, August 2021.

[171] E. B. Karbab and M. Debbabi, "Petadroid: adaptive android malware detection using deep learning," in *Proceedings of the Interna- tional Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 319–340, Springer, 2021.

[172] R. Surendran, T. Thomas, and S. Emmanuel, "Gsdroid: graph signal based compact feature representation for android malware detection," *Expert Systems with Applications*, vol. 159, Article ID 113581, 2020.

[173] H. Bai, N. Xie, X. Di, and Q. Ye, "Famd: a fast multifeature android malware detection framework, design, and implementation," *IEEE Access*, vol. 8, Article ID 194729, 2020.

[174] A. Mehtab, W. B. Shahid, T. Yaqoob et al., "Addroid: rule-based machine learning framework for android malware analysis," *Mobile Networks and Applications*, vol. 25, no. 1, pp. 180–192, 2020.

[175] L. Cai, Y. Li, and Z. Xiong, "Jowmdroid: android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters," *Computers & Security*, vol. 100, Article ID 102086, 2021.

[176] S. I. Imtiaz, S. u. Rehman, A. R. Javed, Z. Jalil, X. Liu, and W. S. Alnumay, "DeepAMD: detection and identification of android malware using high-efficient deep artificial neural network," *Future Generation Computer Systems*, vol. 115, pp. 844–856, 2021.

[177] W. Zhang, H. Wang, H. He, and P. Liu, "Damba: detecting android malware by orgb analysis," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 55–69, 2020.

[178] L. N. Vu and S. Jung, "Admat: a cnn-on-matrix approach to android malware detection and classification," *IEEE Access*, vol. 9, Article ID 39680, 2021.

[179] H. Gao, S. Cheng, and W. Zhang, "Gdroid: android malware detection and classification with graph convolutional network," *Computers & Security*, vol. 106, Article ID 102264, 2021.

[180] A. Guerra-Manzanares, H. Bahsi, and S. Nõmm, "Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization," *Computers & Security*, vol. 110, Article ID 102399, 2021.

[181] S. K. Sasidharan and C. Thomas, "ProDroid - an Android malware detection framework based on profile hidden Markov model," *Pervasive and Mobile Computing*, vol. 72, Article ID 101336, 2021.

[182] Y. Hei, R. Yang, H. Peng et al., "Hawk: rapid android malware detection through heterogeneous graph attention networks," 2021, https://arxiv.org/abs/2108.07548.

[183] D. O. ¨ . S¸ahın, S. Akleylek, and E. Kili¸, "Linregdroid: detection of android malware using multiple linear regression models-based classifiers," *IEEE Access*, vol. 10, Article ID 14246, 2022.

[184] Y. Ban, S. Lee, D. Song, H. Cho, and J. H. Yi, "Fam: featuring android malware for deep learning-based familial analysis," *IEEE Access*, vol. 10, Article ID 20008, 2022.

[185] S. S. Vanjire and M. Lakshmi, "Mdta: a new approach of supervised machine learning for android malware detection

and threat attribution using behavioral reports," in *Mobile Computing and Sustainable Informatics*, pp. 147–159, Springer, Berlin, Germany, 2022.

[186] H. Cai and B. G. Ryder, "Droidfax: a toolkit for systematic characterization of android applications," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 643–647, IEEE, Shanghai, China, September 2017.

[187] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: detecting android malware by building Markov chains of behavioral models," arXiv preprint arXiv:1612.04433, 2016.

[188] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: self-evolving android malware detection system," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 47–62, IEEE, Stockholm, Sweden, June 2019.

[189] W. Li, X. Fu, and H. Cai, "Androct: ten years of app call traces in android," in *Proceedings of the 2021 IEEE/ACM 18th Interna- tional Conference on Mining Software Repositories (MSR)*, pp. 570–574, IEEE, Madrid, Spain, May 2021.