Journal of Integrated
SCIENCE & TECHNOLOGY

Assessing the efficacy of Machine learning classifier for Android malware detection

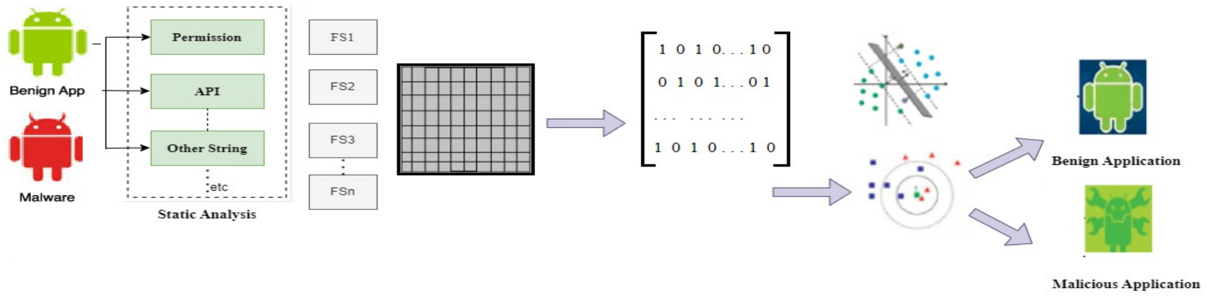
Harshal Devidas Misalkar, Pon Harshavardhanan

School of Computing Science and Engineering, VIT Bhopal University, Madhya Pradesh, India.

Received on: 14-Oct-2023, Accepted and Published on: 04-Jan-2024

Article

ABSTRACT



The primary challenges faced by software security experts is the identification and detection of malware within Android applications, as dangerous software is increasingly being embedded in sophisticated manners in application software. The existing applications, as well, are expanding in size and becoming increasingly intricate in terms of their functionalities. The ongoing endeavor of extracting valuable and indicative functionality from applications is a perpetual undertaking. There has been a lack of comprehensive studies that examine the specific attributes designed for identifying malicious applications on the Android platform. This is despite the existence of several feature extraction methods employed in prior research endeavors. Here, a comprehensive and concise analysis is presented to comprehend the behavior of applications using various criteria to identify harmful applications. This study evaluates the efficacy of ten different machine learning classifiers by analyzing a dataset including 15,036 applications categorized as either harmful or benign. The evaluation of classifiers involved the utilization of many metrics like Accuracy, Area Under the Curve (AUC), False Positive Rate (FPR), and False Negative Rate (FNR) towards development of illustrative framework for the detection of Android malware applications.

Keywords: Android applications, IoT, Ensemble learning, feature extraction, malware detection, reverse engineering, machine learning

INTRODUCTION

Numerous mobile applications have been developed with the aim of facilitating consumers in adopting a more intelligent living environment. These programs encompass a wide range of functionalities, including social networking platforms as well as applications pertaining to financial management. Over the past few years, Android smartphones have consistently held an average global market share of 80%, establishing their dominance in the mobile device industry. Malicious programs are mushrooming in number in the meantime. Numerous security issues with mobile applications pose a risk to the privacy and property of users. As an illustration, certain malicious software may take users' personal account information without the users' consent.

These applications' bad habits mostly involve consuming traffic, stealing personal data, making erroneous calculations, etc. As the number of smartphone apps rises, so does the security issue caused by unauthorized access to various personal resources. The applications thus become less secure, stealing personal information and engaging in SMS fraud, ransomware, etc.

The official store for Android apps is called Google Play (apps, henceforth). Bouncer is a Google-implemented automated monitoring service that checks submitted software for possibly dangerous behaviour in order to safeguard the market against malicious programmes.[1] To help keep devices secure, Google recently launched a new security feature for Android Play Protect in addition to the Bouncer utility. The device is routinely scanned by this feature, which also issues danger alerts. There are a few unofficial repositories for Android apps in addition to the official

*Corresponding Author: Harshal Devidas Misalkar
Tel: 9503251870
Email:harshal.misalkar2019@vitbhopal.ac.in

Cite as: *J. Integr. Sci. Technol., 2024, 12(4), 788. URN:NBN:sciencein.jist.2024.v12.788*

market. However, the majority of these do not have any means for scanning for malware when user-uploaded apps are present; as a result, these repositories are among the main sources of malware.

It is evident that the perpetration of destructive activities has significantly jeopardised the mental well-being and material possessions of individuals. Several strategies for analysing and detecting malicious applications have been created through the examination of app behaviour. The primary objective of these techniques is to prevent the distribution of malicious application and low-quality applications in the marketplaces.

In recent years, research has focused heavily on how to identify programs that exhibit dangerous behavior and safeguard users' privacy. The Android ecosystem faces a significant and difficult problem with malware detection. To differentiate between harmful and good applications, numerous strategies have been developed.

Machine learning is a frequently used technology that is used for Android malware identification. It is frequently used in classification processes, and creating features is the key stage in identifying malicious Android apps. The efficacy of the detection is contingent upon the optimal functioning of the selected attributes. The current characteristics can be categorised into three distinct groups: static features obtained through static analysis, dynamic features obtained through dynamic analysis, and meta-data-based features. Despite the rapid progress in detecting malicious applications that exploit extracted app features, certain challenges persist.

Malware applications can be predicted by training a model with extracted static features from reverse-engineered Android applications. This can be accomplished using machine learning techniques such as the Support Vector Machine (SVM) algorithm, logistic regression, ensemble learning, and other applicable algorithms. The use of intrinsic attributes in reverse-engineered Android applications is prevalent in machine learning approaches, thereby easing the difficulty of this endeavor.

String features, or structural features, are the basis for Android malware detection techniques based on static analysis. String features, also known as meta-data, are exhaustive descriptions of software or application source code. Permissions, intents, API queries, etc. are frequently discussed. String features, or structural features, are the basis for Android malware detection techniques based on static analysis. String features, also known as meta-data, are exhaustive descriptions of software or application source code. Permissions, intents, API queries, etc. are frequently discussed.

A. Issues

1. Common Issues of Extracting The Features

- The extraction of features from an Android Package (APK) might be time-consuming due to its increasing size and intricate behaviours, hence diminishing the effectiveness of identification. For example, when employing static analysis techniques, it is commonly observed that the process of extracting function call graphs for a Google Play apk with a size of 15 MB typically requires approximately 15 minutes. The real-time discovery of this is clearly undesirable for end users.
- Up to a million features can be taken from a single programme. Many features, however, are zero.[2] It is crucial to figure out how to handle the sparse vectors effectively.[3]

2. Issues of Extracting Static Features

For app verification, static features analysis is frequently employed. However, there are a number of significant difficulties that static analysis faces

- Extracting well-discriminated static data from Android apps poses a significant challenge due to the ever intricate and polymorphic nature of their behaviour.
- The proliferation of applications is accompanied by a corresponding growth in the number of characteristics. A significant concern is to the optimal processing of the continuously increasing array of features in a manner that is both useful and efficient. Based on the categorization of feature sets in previous study,[2] we classified all static features into two distinct types: platform-defined features and app-specific features. As the number of applications increases, the amount of app-specific features tends to increase, while the number of platform-defined features

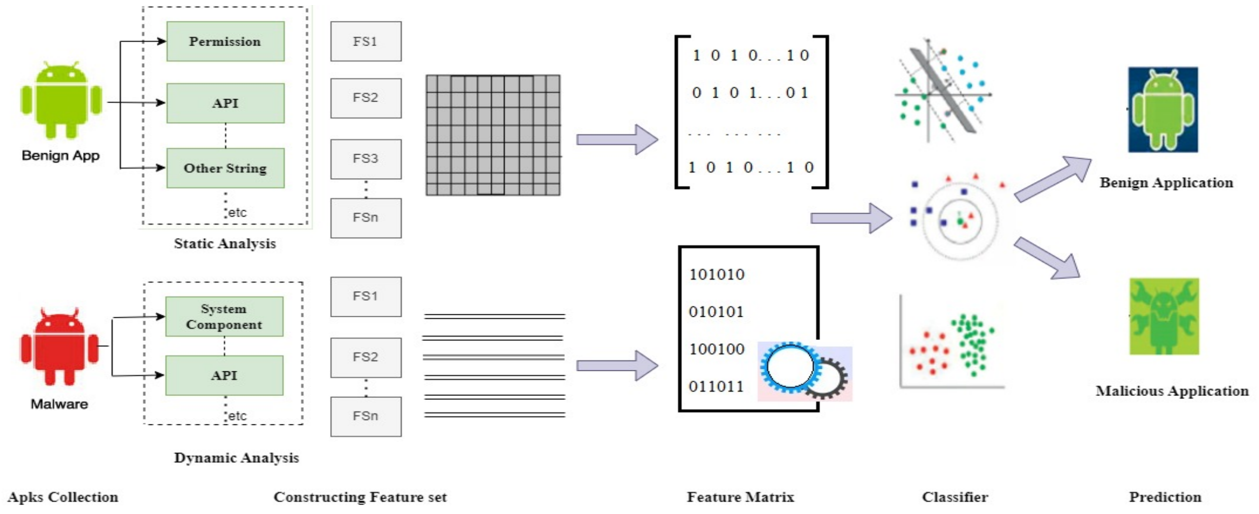


Figure 1: The process of Android malware application detection.

remains constant. Platform-defined features are commonly utilised for the automated detection of malicious applications due to their higher level of persistence compared to features that are specific to individual apps. In essence, the presence of numerous application-specific features for processing may potentially result in inefficient identification. applications increases, the amount of app-specific features tends to increase, while the number of platform-defined features remains constant.

- Platform-defined features are commonly utilised for the automated detection of malicious applications due to their higher level of persistence compared to features that are specific to individual apps. In essence, the presence of numerous application-specific features for processing may potentially result in inefficient identification.
- Numerous malicious programmes employ obfuscation techniques such as dynamic code loading or code encryption in order to evade detection based on static characteristics. According to a comprehensive investigation by D. Wermke et.al.,[4] it has been determined that over 25% of the applications available on Google Play undergo obfuscation. Furthermore, this ratio significantly increases to 50% for the most popular applications with over 10 million downloads. Nevertheless, the efficacy of numerous static analysis approaches is compromised by the presence of obfuscation, hence diminishing their efficiency and becoming them more challenging to implement compared to dynamic analysis.

3. Issues of Extracting Dynamic Features

Malapps' covert behaviors can be extracted through dynamic analysis. The information gathered from observing and documenting an app's actions may accurately reflect the app's intended use. However, there are still some problems with dynamic feature extraction.

- Due to the inherent limitations of dynamic analysis in fully exploring all possible execution paths, the utilisation of dynamic features for malware application identification may result in the occurrence of false negatives.
- If an application is protected by runtime security mechanisms, it may prevent the extraction of dynamic features, such as DexGuard.

Figure 1 elucidates the process involved in the detection of Android malware applications.

Contributions

To successfully detect malicious software, it is essential to identify and extract unique traits. While many different types of characteristics have been explored in the past, it appears that no comprehensive study has been undertaken on the features used for the detection of harmful programmes. In order to provide a comprehensive and coherent understanding of the recent developments in identifying hazardous applications through the characterization of app behaviors with varied properties, this study zeroes in on the challenges involved with assessing efficient features and introduces a feature taxonomy.

Overview of Android System and Security

A. Android Platform

Android is an open-source mobile platform based on the Linux kernel and developed primarily for smartphones and other connected devices. There are four distinct tiers that make up Android's architecture: the application layer, the framework layer, the library layer, and the Linux kernel layer. Memory management, task management, and network protocols are only a few of the crucial features provided by Linux's kernel layer. The fundamental drivers for all hardware components are located at this layer. The library layer, consisting of both the native library and external libraries from third-party sources, supplies the application's primary library in order to improve the framework layer's capabilities.[5]

The application framework layer is analogous to a middle layer that provides strategic management of the system's components and improves its scalability. The Activity Manager, Window Manager, Resource Manager, Location Manager, Content Provider, and other components of the application framework work together to achieve this goal. The application layer encompasses all running applications on Android smartphones and serves as the sole layer responsible for user communication.[6,7]

B. Android Application

It is usual practice to use the Java programming language and the Android Software Development Kit's (SDK) application programming interfaces (APIs) to create Android applications. The Android platform and third-party developers both offer native libraries that may be incorporated into applications alongside Java code. To install and run an app on an Android device, developers package its source code, data, and resources into a file known as an Android Application Package (APK). The Android runtime environment is used by an APK once it has been installed on an Android device.

Activity, Broadcast Receivers, Service, and Content Provider are the four main parts of any Android app. The manner in which users engage with the smartphone screen and the resulting User Interface are influenced by activity controls. The exchange of information between the operating system and applications is facilitated through the utilization of broadcast receivers. The background processing of an application is overseen by a service in order to execute operations that need a significant amount of time.

C. Incorporated Security Mechanisms

Traditional Access Control Mechanism

The Linux kernel security mechanism used by Android is analogous to conventional access control mechanisms. The ability to access an item can be restricted using an access control system. Maintaining data secrecy and integrity through this method is crucial. Both obligatory and discretionary controls (abbreviated MAC and DAC, respectively) are possible to impose. MAC is supported by Linux's security module. DAC is made possible by the usage of secure file sharing.

Mechanism Based on Inspection of Permission

The permission-based security strategy implemented in Android applications restricts the access to resources that these applications are allowed to utilize. In order to access restricted resources, applications are required to employ XML files to seek permissions. Applications are incapable of accessing restricted resources unless they obtain authorization from the Android operating system. The Android permissions are classified into four tiers, specifically Normal, Dangerous, Signature, and Signature/System. Once an application seeks permissions, both low-level permissions, such as usual and dangerous levels, are granted. Advanced rights encompass two categories: signature level permissions and signature/system level permissions. The platform-level authentication required by these permissions must first be completed by the app. However, this approach has a lot of drawbacks. Users must decide whether the permissions that an app requests should be approved, but they lack the knowledge necessary to do so. In addition, the user will be prompted to provide all potentially risky permissions during installation if the target SDK version of the application is less than 23 or if the device is running Android 5.1.1 (API level 22) or earlier. Unless users make modifications, the accepted permissions remain valid during the duration of the application.

Encryption Mechanism

The Android operating system incorporates an encryption mechanism designed to prevent unauthorized users or applications from accessing certain sensitive data. The Android operating system, starting with version 3.0 and onwards, incorporates encryption techniques. There is a growing user concern regarding the safeguarding of private information, including phone events, SMS messages, and some payment details. Consequently, the encryption mechanism has gained increased significance inside the Android operating system.

Digital Signature Mechanism

The digital signature technique is crucial to the application layer's security. Since programmers without digital signatures cannot be installed, Android app creators must provide them for their creations. Attacker must resign the app if he purposefully alters the internal le of the APK. The replication of the original signature by the attacker is contingent upon the acquisition of the private key belonging to the original publisher. When applications require updating, the signatures associated with those applications will also undergo scrutiny. Application trustworthiness and integrity are guaranteed by digital signature.

Sandbox Mechanism

In the Android operating system, running apps are separated by sandboxes. Apps can operate in a strictly restricted environment called a sandbox. Each Android application has its own Dalvik virtual machine, process space, and resources when it is in use. As a result, distinct apps cannot communicate with one another or use one another's memory or resources.

Method of Detecting Android Application

Static, dynamic, hybrid, and meta-data analysis are the primary types of analysis techniques now used to identify Android apps. We briefly describe these analysis techniques and group the analyzed publications into categories based on the taxonomies of the attributes they used.

Static Analysis

The Android platform is increasingly being attacked and faces dangerous risks from malicious software. As a result, a lot of research focuses on using static analysis to find malicious programmers. Apps are initially unpacked and decompiled into files that contain the apps' most important data during static analysis. Then, these files are examined to see if they contain any harmful code. Static analysis is well-known in traditional malapp identification and is becoming more and more popular as an effective market protection technique.

Android smartphones with limited resources can benefit from it because the analysis is done without actually running the app. Static analysis uses a lot less time and resources. As a result, it is a fairly quick process. Malicious apps that employ reverse engineering strategies like obfuscation and repackaging, however, can foil this strategy.

The current study focuses on static feature extraction.

Dynamic Analysis

Dynamic analysis, in contrast, looks for dangerous behavior after the apps have been installed and run on emulators or actual devices. The system creates snapshots of the execution of the processor, network activities, system calls, SMS transmissions, phone conversations, and other pertinent data to determine whether or not a programme is malicious. The implementation of this method requires the involvement of either human or automated entities, as malevolent activities may only be triggered under certain circumstances.

The data obtained via dynamic analysis provides a realistic representation of the program's actual usage. Android's OS has to spend a lot of time and energy on the implementation of dynamic analysis. Furthermore, it is important to note that dynamic analysis techniques may fail to detect malicious applications that have been specifically designed to prevent their execution in emulated environments.

Static Features

Finding and utilizing the appropriate features is crucial to improve the model's accuracy. Using the SimpleImputer class of the ScikitLearn Library, a dataset is first preprocessed in this research paper. After preprocessing, features are retrieved, and using the feature importance technique, the features that are significant in determining whether or not an application is malicious are selected. Total of 10 features were chosen based on their score values. significant attributes.

The feature importance for each feature can be determined by employing a model's feature importance technique. The relative importance of each feature with respect to the output variable is assessed by providing a numerical score to each feature, where higher scores correspond to greater significance. The Scikit-Learn toolkit for machine learning includes a Tree Based Classifier, which provides feature significance as a default feature. This relevant feature selection method reduces 215 features to 10 features that are crucial for differentiating between malicious and benign programs. Following 10 static features used in this paper for malicious application detection.

1. TELEPHONYMANAGER.GETLINE1NUM0ER
2. TELEPHONYMANAGER.GETDEVICEID
3. ON1ERVICECONNECTED
4. ANDROID.O1.0INDER
5. 1ERVICECONNECTION
6. ATTACHINTERFACE
7. ANDROID.TELEPHONY.1M1MANAGER
8. TRAN1ACT
9. READ_PHONE_1TATE
10. 1END_1M1

1.TELEPHONYMANADER.GETLINE1NUMBER

Within the domain of Android application development, the TelephonyManager class assumes a crucial role as a fundamental asset for retrieving telephony-related data on a given device. One of its several functionalities includes the provision of the ability to retrieve the telephone number linked to the SIM card of the device by utilizing the getLine1Number method.
Here's how you can integrate this method into your Android code:

```
TelephonyManager telephonyManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
String phoneNumber = telephonyManager.getLine1Number();
if (phoneNumber != null && !phoneNumber.isEmpty()) {
  // Feel free to utilize the retrieved phoneNumber as needed.
} else {
```

It is important to acknowledge that the accessibility of a phone number might vary depending on factors such as the carrier, SIM card, and device settings. In certain cases, there may be circumstances where specific carriers or devices do not make the phone number accessible using this particular function, or it may result in an empty string being returned. Furthermore, it is imperative to verify that the AndroidManifest.xml file incorporates the essential permissions required for accessing telephony-related data.

TelephonyManager.getLine1Num0er method, which is part of the Android framework, is designed to retrieve the phone number that is linked to the SIM card of the device. It is imperative to recognize that placing exclusive reliance on this particular methodology for the detection of Android malware has some constraints and may not generate outcomes of optimal efficacy. The following are important factors to keep in mind:

Privacy Implications: The act of obtaining a user's phone number without their explicit consent might give rise to substantial privacy considerations. Google has implemented rigorous restrictions regarding the utilization of sensitive data, such as phone numbers. As a result, legitimate applications are now required to provide justification for their necessity of accessing such information.

Limited Applicability: The functioning of certain authorized programs does not always require access to the user's phone number. As a result, malevolent applications can effectively elude detection by refraining from employing this technique.

Evasion Strategies: Individuals who develop malicious applications possess a high level of skill in evading detection mechanisms. The adversaries have the ability to utilise many tactics, such code obfuscation, postponing suspicious actions, or dynamically seeking permissions. These techniques make it difficult to detect their activities purely based on the usage of getLine1Number.

Potential for false positive: The complete reliance on the getLine1Number method for malware detection has the potential to yield false positive results. Certain authentic applications utilise this technique for legitimate intentions, such as authenticating a user's identity throughout the process of setting up an account.

Inadequate Understanding of Runtime Behavior: The effectiveness of malware detection often relies on the observation of an application's behavior during its execution, thorough examination of network connections, and identification of any indications of suspicious or malicious activities. An exclusive reliance on methods such as getLine1Number in a static analysis does not offer a thorough understanding of an application's true behaviour.

2. TELEPHONYMANAGER.GETDEVICEID

The TelephonyManager.getDeviceId function in Android development is a useful tool for acquiring a distinct identification linked to the device's radio equipment. The generally used term for this identity is the IMEI (International Mobile Equipment Identity) for GSM devices or the MEID (Mobile Equipment identity) for CDMA devices. Identifiers are of utmost importance in the process of differentiating and verifying mobile devices within wireless networks and in relation to service providers.

```
TelephonyManager telephonyManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
String deviceId = telephonyManager.getDeviceId();
if (deviceId != null && !deviceId.isEmpty()) {
  // Proceed to utilize the acquired deviceId according to your
requirements.
} else {
  // Be prepared for scenarios where the device ID retrieval is
unsuccessful or unavailable.
}
```

To implement the getDeviceId method in your Android code, follow these steps:

It is noteworthy to acknowledge that the device identification (ID) may vary depending on the kind of device (GSM or CDMA) and the specific implementation on the device. In addition, it is important to note that there may be limitations on accessing the device ID in some situations, mostly owing to concerns surrounding privacy and security. In order to access this information, it may be

essential to set the required permissions in the AndroidManifest.xml file.
The static feature known as "TelephonyManager.getDeviceId" holds considerable importance in the realm of Android malware detection owing to its diverse range of applications.

Device Identification: Within the context of malware detection, the establishment of a distinct device identification is of utmost importance. The method "getDeviceId" is utilized to obtain an identifier that is linked to the radio equipment of the device, such as the International Mobile Equipment Identity (IMEI) for Global System for Mobile Communications (GSM) devices or the Mobile Equipment Identifier (MEID) for Code Division Multiple Access (CDMA) devices. The aforementioned identification plays a crucial role in differentiating one device from another.

Anomaly detection is a common approach employed in malware detection systems, wherein a reference point is established to define the normal behavior of a device. This reference point often includes several attributes such as the device ID. Any departures from the specified baseline have the potential to activate warnings. In the event that the device identification (ID) undergoes an abrupt alteration or manifests as invalid, it could potentially serve as an indication of suspicious activity, hence suggesting the possibility of intervention by malicious software (malware).

The identification of cloned or emulated devices is a common concern in the context of detecting malicious software, as such software often functions on these types of devices. In certain scenarios, it is possible for numerous devices to possess the same device ID. The identification of this abnormality can play a crucial role in the detection of suspicious behavior.

The monitoring of malicious activities involves the observation of malware's communication with command and control servers. Through the analysis of device IDs in conjunction with other parameters, security systems have the capability to identify patterns of malicious activity spanning over numerous devices, hence facilitating the timely identification of such behavior.

Policy enforcement in organizations and mobile device management (MDM) systems heavily depend on device attributes, such as the device ID, to effectively implement security policies. The identification of modifications to this identity is of utmost importance in order to guarantee adherence to security rules.

User Authentication: Certain applications incorporate the device ID as a component within their user authentication and authorization mechanisms. In the event that malware obtains possession of this identification, there is a possibility for the malware to assume the identity of the user, resulting in unauthorized access and potential breaches of security.

Forensic Analysis: Following a security incident, possessing knowledge of the device ID is of utmost importance in conducting forensic investigations. This technology facilitates the identification of the origin of hostile actions and contributes to the comprehension of the extent of an assault.

Nevertheless, it is crucial to recognize that although the "getDeviceId" method holds significance in detecting malware, its utilization must adhere to privacy standards and obtain user authorization. With the continuous development of Android, there may be limitations or discontinuation of some device IDs in newer versions, which may require modifications in the methods used for identification.

3. ON1ERVICECONNECTED

The onServiceConnected method holds significant importance in the context of Android app development, as it serves as a pivotal callback function that is intricately linked to the Service Connection framework. This framework facilitates the exchange of information and engagement across many elements of an Android application, including Activities or Fragments, as well as background services.

The initial step in setting up a ServiceConnection involves the instantiation of a ServiceConnection object within an Android component, often an Activity or Fragment. The aforementioned entity assumes the role of overseeing the establishment and maintenance of a connection to a certain service. The process of establishing a connection with a service. The binding process is initiated by invoking the bindService method, wherein an Intent is provided to identify the desired service for establishing a connection.

The onServiceConnected callback is triggered upon successful establishment of the service connection. The callback function offers the user the chance to get a reference to the connected service, so providing them access to its capabilities.

Service Interaction: Once the service reference is obtained in the onServiceConnected callback, users are able to engage with the service by executing its methods or transmitting data as required.

The onServiceDisconnected callback is triggered when the service connection is unexpectedly terminated, such as in the event of a crash or explicit unbinding. The callback function facilitates the execution of essential cleanup operations or resource management tasks.

This is an illustrative example showcasing the potential implementation of the onServiceConnected method within an Android Activity.

```
private MyService myService; // Declare a reference to the service
private ServiceConnection serviceConnection = new ServiceConnection() {
  @Override
  public void onServiceConnected(ComponentName name, IBinder service) {
      // This block executes when the service connection is successful.
      // You can access the service via the IBinder.
      MyService.LocalBinder binder = (MyService.LocalBinder) service;
      myService = binder.getService();
      // Now, you can utilize myService to call methods within the service.
  }
  @Override
  public void onServiceDisconnected(ComponentName name) {
      // In case of an unexpected service disconnection, this callback is invoked.
```

```
        // It's an opportunity to perform cleanup or handle disconnection gracefully.
        myService = null;
    }
};
```

The onServiceConnected method assumes a significant role in the creation of Android applications as it enables the establishment of communication and interaction between an Android component and a background service. This technique facilitates the smooth integration of services into the functioning of your application.

The onServiceConnected method in the Android framework serves as a callback function that facilitates the establishment of connections to bound services. The importance of this feature within the context of Android malware detection comes in its function as one of the monitored elements that can be examined to identify possibly malicious activities. Nevertheless, it is crucial to comprehend that its significance in the detection of malware is merely one element within a more comprehensive framework.

Behavioral Analysis: Malicious software frequently exhibits atypical behavioral patterns, such as the initiation of connections with distant servers, the transmission or reception of data, or the interaction with system components. Analyzing the onServiceConnected method facilitates the identification of atypical behavior, particularly in cases when it pertains to questionable service connections.

Android applications are obligated to solicit explicit permissions in order to gain access to particular system services. When an application endeavors to establish a connection with a service without possessing the requisite permissions, it has the potential to arouse suspicions regarding its potentially malevolent intentions. The examination of the onServiceConnected method facilitates the identification of probable violations of permissions.

Intent-based assaults involve the utilization of the onServiceConnected method by malware to carry out malicious activities. This strategy deceives users into initiating harmful services or components. The act of monitoring this particular method has the potential to facilitate the detection of service connections that are deemed suspicious or unauthorized.

The assessment of network traffic reveals that malware frequently engages in communication with remote servers to carry out various activities, including but not limited to receiving instructions and extracting data. The utilization of onServiceConnected for establishing connections pertaining to network activity can offer valuable insights into possibly malicious network behavior when observed. The significance of the onServiceConnected method is influenced by the contextual factors in which it is invoked. For example, if a service connection is deemed superfluous within the application's functioning, it may give rise to problems.

It is imperative to recognize that the detection of Android malware is contingent upon the utilization of a blend of static and dynamic analytic methodologies. Static analysis is a process that involves the examination of an application's code and manifest files without executing it. On the other hand, dynamic analysis refers to the practice of running the application in a controlled environment in order to monitor and analyze its behavior. The onServiceConnected method is categorized as dynamic analysis as it is conducted at runtime.

4. 1ERVICECONNECTION

The ServiceConnection interface in the Android platform is not a static entity, but rather a pivotal component that facilitates the interaction and binding of services within applications developed for the Android operating system. The aforementioned aspect bears considerable significance within the realm of Android malware detection, as it functions as a mechanism for observing and evaluating the behavior of an application when initiating connections with various services. While not fundamentally designed as a security feature, the utilization and behavior of a system can provide valuable insights for the detection of potential infection. The following are arguments that underscore the significance of ServiceConnection in the identification of Android malware:

Behavioral Analysis: Malicious software frequently displays atypical behavior within Android applications, such as establishing connections with dubious services or executing unauthorized actions. Through a thorough examination of the utilization of the ServiceConnection interface, it becomes feasible to scrutinize whether an application's interactions with services diverge from anticipated and lawful patterns.

The establishment of connections to services is a common practice among Android applications, as it enables them to carry out a range of functions. The diligent observation of the ServiceConnection interface can aid in the detection of situations when an application establishes connections with services that it should not possess authorization for, hence potentially indicating malevolent motives.

The identification of privilege escalation involves the detection of malware that seeks to bind to system-level services or elevate its privileges in order to enhance its authority and control over the Android device. Examining the utilization of ServiceConnection can aid in identifying instances of privilege escalation endeavors.

Protecting Against Intent-Based Attacks: Malicious applications have the potential to abuse the ServiceConnection interface in order to carry out intent-based attacks, with the aim of tricking users into establishing harmful services. Conducting a comprehensive analysis of the ServiceConnection can facilitate the detection of these fraudulent strategies.

The examination of network activity reveals that the utilization of a service connection for tasks pertaining to network operations may give rise to concerns regarding potentially harmful network behavior. The examination of this behavior is crucial in identifying malware that establishes communication with external servers with nefarious intentions.

The significance of the ServiceConnection interface in dynamic analysis should be underscored, since it plays a crucial role in actively observing an application's behavior during runtime. The utilization of dynamic analysis is crucial in the identification of malware that exhibits dangerous behavior exclusively during the active execution of the application.

It is imperative to acknowledge that the identification of Android malware involves the integration of static and dynamic analytic methodologies. Static analysis is the process of examining an application's code and resources without executing it, whereas dynamic analysis involves running the application in a controlled environment to observe its behavior. This includes monitoring the utilization of the ServiceConnection interface.

5. ATTACHINTERFACE

The attachInterface function is not commonly considered a static feature in the context of Android. Instead, it is an internal method employed by the Android framework to facilitate inter-process communication (IPC) across different components. Although it plays a crucial role in the operation of the Android system, Android malware detection does not often prioritize this aspect.

The inclusion of internal method analysis, such as attachInterface, in the detection of Android malware is not widely adopted in current practices. Typically, the main emphasis lies on higher-level behaviors and patterns that can be discerned during the execution of an application or by a comprehensive analysis of its code and manifest file. There are multiple reasons why the attachInterface function is not commonly considered a vital aspect in the identification of Android malware.

The attachInterface function is a restricted mechanism that is solely utilized within the Android framework. The lack of accessibility or exposure to manipulation by Android app developers renders it an impracticable target for misuse.

The primary focus of malware detection efforts revolves around the examination of an application's behavior at an elevated level, encompassing its interactions with system services, network activities, and permission utilization. The practice of detecting malware at the level of internal framework methods, such as attachInterface, is not commonly employed.

Dynamic analysis is a prevalent approach in the identification of Android malware, wherein the behavior of an application is continuously monitored within a controlled environment. The utilization of dynamic analysis allows for the detection of potentially suspicious or malicious actions that may not be readily discernible through static code inspection.

Privacy and security concerns may arise when analyzing or monitoring internal framework methods such as attachInterface. The utilization of these methods is not intended for public consumption, and their unauthorized access may potentially jeopardise the stability and security of the Android system.

6.READ_PHONE_STATE

The privilege known as "READ_PHONE_STATE" holds significant importance within the realm of Android app permissions. This permission grants an application the ability to retrieve data pertaining to the current state and identity of the phone, encompassing information such as the phone number, device identification, and call status. The "READ_PHONE_STATE" permission is of considerable importance in the domain of Android malware detection due to numerous important factors.

The inclusion of the "READ_PHONE_STATE" permission in apps raises concerns over privacy, as it grants potential access to sensitive user data pertaining to device information and identity. The granting of this permission gives rise to privacy concerns, as it might potentially be exploited by malicious applications to gather user data without obtaining authorization, so leading to substantial breaches of privacy.

The permission known as "Call Interception" on Android devices might potentially be exploited by malware to intercept or monitor both incoming and outgoing calls. The engagement in such malevolent conduct has the potential to result in unfavorable consequences, such as the unauthorized interception of telephone conversations or the unauthorized redirection of calls to external parties.

Device Identification: The permission to collect unique device identifiers, such as the IMEI (International Mobile Equipment Identity) number, can be used by malware for malicious purposes. The utilization of these identifiers has the potential to be leveraged for activities such as device fingerprinting or user tracking, hence giving rise to apprehensions regarding user privacy and security.

Telephony fraud is a prevalent issue in the realm of Android malware, wherein certain forms of malware use the "READ_PHONE_STATE" permission for illicit activities. This include actions such as the unauthorized transmission of premium-rate SMS messages, which may result in adverse financial consequences for the recipient.

Indicators for Malware Detection: The identification of the "READ_PHONE_STATE" permission, particularly in applications that lack a legal necessity for its access, might be regarded as a discerning factor suggestive of dubious or conceivably harmful conduct. In the realm of Android security, both security mechanisms and antivirus software frequently identify applications that possess superfluous or uncommon permissions, hence prompting a more thorough examination.

It is crucial to underscore that although the inclusion of the "READ_PHONE_STATE" permission may arouse concerns in the context of Android malware detection, it should not be assumed that all applications asking this permission possess harmful intent. Legitimate applications, especially those that offer call management or caller ID services, may legitimately necessitate this authorization in order to fulfil their intended functions.

The achievement of efficient malware detection on the Android platform necessitates the implementation of a thorough strategy that incorporates static and dynamic analysis techniques, alongside ongoing behavior monitoring. The evaluation of permissions, such as "READ_PHONE_STATE," is an integral aspect of static analysis. However, their importance is assessed within the framework of an application's behavior and the requirement of the permission for valid functionality of the app.

7. 1END_1M1

The "SEND_SMS" permission is an essential Android permission that confers upon an application the capability to dispatch SMS (Short Message Service) messages from the user's mobile device. The significance of this permission within the realm of Android malware detection is noteworthy due to various compelling factors.

Unwanted SMS Messages: Malicious applications may deceptively seek authorization to "SEND_SMS" in order to covertly send SMS messages without the user's knowledge or agreement. The content of these communications may involve services that require payment at premium rates or through subscription models, which could lead to unforeseen financial expenses for the consumer.

Certain types of Android malware exploit the "SEND_SMS" permission to disseminate unsolicited messages or fraudulent communications, commonly known as spam or phishing, to individuals listed in the user's contact directory. These acts have the potential to result in the occurrence of unsolicited solicitations or even deliberate efforts to steal confidential personal information.

The utilization of SMS as a communication channel between a command-and-control server and some types of Android malware, such as botnets, has been seen. The potential exists for the transmission and reception of SMS messages in order to carry out orders or obtain updates from the server controlled by the attacker.

The exploitation of the "SEND_SMS" permission by malicious programmes might result in the flooding of recipients with a significant number of SMS messages. This has the potential to create a denial-of-service (DoS) situation, which can interrupt the functioning of the recipient's device or network.

Unauthorized operations: This particular ability can be utilized to carry out operations on the device without proper authorization, like modifying device settings or sending messages to premium-rate services, all without obtaining the explicit consent of the user.

The inclusion of the "SEND_SMS" permission in an application's manifest is sometimes regarded as an initial indication of possible suspicious behavior in the context of malware detection. The security features integrated into the Android operating system, along with antivirus software, have the capability to identify and subject programmes that possess this particular permission to additional examination.

It is imperative to acknowledge that authentic applications may legitimately necessitate the "SEND_SMS" permission for valid reasons, such as messaging applications or those supporting two-factor authentication by SMS. Nevertheless, the importance of this authorization in the detection of Android malware relies on its susceptibility to exploitation by malicious programmes.

The detection of malware on Android devices involves a comprehensive methodology that incorporates several techniques such as static and dynamic analysis, continuous monitoring of behavior, and heuristic evaluation. The examination of permissions, such as "SEND_SMS," is a component that is subject to scrutiny during static analysis. The assessment of the significance of this permission relies on the wider context of the application's behavior and its justified requirement for sending SMS messages. As a result, applications that request this permission without a transparent and valid rationale can give rise to concerns within the context of malware detection.

Related Work

In recent years, there has been a significant amount of research focused on the detection of Android malware using machine learning techniques. Different detection strategies are employed depending on how the features used in machine learning algorithms are collected.

The aforementioned analytical techniques commonly fall into three categories: static, dynamic, or hybrid. Machine learning algorithm characteristics are collected through dynamic analysis by running applications on either a physical or virtual device. Characteristics pertaining to machine learning approaches in static analysis are acquired without the need to launch any applications. Constructing the necessary infrastructure poses challenges due to the utilisation of dynamic analysis in programme execution. They do, however, successfully fend off zero-day assaults. Considering that no applications are launched during static analysis, the procedure is fairly quick. There is a hybrid analysis method in addition to static and dynamic analysis methodologies. This method combines features gained through static and dynamic methods.[8]

The methodologies proposed in the referenced paper contribute to the enhancement of key factors, such as selected features for classification and the overall accuracy in predicting malware detection. Numerous research has integrated all of these elements in order to enhance the efficiency of the detection rate. Certain studies have focused on enhancing precision, whereas others have prioritized the provision of an expanded dataset. Various feature sets have been employed for implementation.[9]

The approach to doing static assurance analysis for Android applications is contingent upon the objectives of the user. The process of performing a static assurance analysis on an application entails the computation of percentages that indicate the occurrence of risky function calls. These function calls are dependent on the successful execution of user interactions. The inclusion of crucial function calls is taken into consideration while calculating the assurance score. Furthermore, programmes running on other operating systems can benefit from static assurance analysis. Using static analysis, a method was put forth by G. Jacob et.al.[10] to compare an application with known malware. The application was compared using a similarity metric to known malware.[11]

Z. Aung et.al.[12] offer a system aimed at improving the organization of the Android Market. This framework focuses on detecting and identifying harmful applications that specifically target the Android platform. The proposed framework seeks to develop an Android malware detection system that utilizes machine learning methods to differentiate harmful applications, hence enhancing the security and privacy of individuals using smartphones. The current system has been created with the purpose of monitoring and documenting various permission-based attributes and events that are obtained from Android applications. The aforementioned attributes are next analyzed by machine learning classifiers to ascertain the characteristics of the programme, specifically whether it is benign or malicious. The paper combines a total of 700 malware samples and 160 attributes from two datasets. The Random Forest (RF) method achieved an accuracy rate of approximately 91% for both datasets.

A total of 2000 malicious programmes, classified into 18 distinct families based on their characteristics, were organised and documented in a previous study.[13] The Cuckoo Sandbox was used to evaluate applications and extract the most distinguishing behavioural traits that set different harmful families apart from one another. A technology known as online machine learning was used to classify malware into different families using the features that were acquired. In the experiments, all seven groups of applications were appropriately classified. The android.trojan.smskey family was found to have the lowest performance rate.

Bayesian classification is used to create a unique malware detection approach.[14] The Bayesian classifier approaches analysis of static data by disassembling Android apps. The APK (Android Package Kit) tools are utilized to extract various attributes, which are afterwards employed in a Bayesian classifier for the purpose of identifying malicious code. An additional technique employed in the realm of malware detection involves the employment of an inter-component communication taint analysis tool in combination with inter-component communication, with the aim of detecting and identifying instances of information breaches. It is advisable to employ the technique of "chasing stains" in order to detect vulnerabilities associated with hijacking in Android applications, namely those that arise in the interface between sensitive sources and externally accessible interfaces. Both Leak Miner and Android Leaks offer users the opportunity to effectively control the Android life cycle and callback mechanisms. Nevertheless, the limited ability of these instruments to adapt to specific contexts makes it impractical to conduct precise analyses of a wide range of potential scenarios.

In the work by Rovelli et.al.,[15] the authors have presented a new approach, known as Permission-based Malware Detection Systems (PMDS), for the identification of Android malware. The methodology employed in this study is the examination of a dataset including 2950 Android applications, encompassing both benign and malicious samples. Within the framework of Permission-based Mobile Device Security (PMDS), the aggregation of permissions requested by an application is seen as a behavioral indicator. Following this, a machine learning model is developed using these signs in order to detect potentially risky behavior exhibited by unverified applications. The PMDS system has demonstrated a high level of efficacy in detecting previously undetected malware, with a success rate of 92.94%. Furthermore, it has managed to maintain a low false-positive rate of 1.523.93%.

N. Milosevic et.al.[16] employed machine learning techniques to propose two separate methodologies that rely on static analysis. The initial approach involved the utilization of static analysis to obtain application permissions. The second approach utilized the bag-of-words model to analyze source code. Based on the available facts, it can be deduced that the computational expense associated with the initial strategy is comparatively lower in comparison to the second approach. The M0Droid dataset, comprising 200 Android applications categorized as harmful and 200 applications categorized as good, was subjected to machine learning algorithms. With the SMO algorithm, the permission-based method showed the best results. The f-measure performance score for this was 0.879. By experimenting with various bagging methods, this success was raised to 0.894. According to the f-measure metric, this performance was 0.951. By experimenting with various bagging methods, this success was raised to 0.9560.

Congyi[17] proposes a method for distinguishing and to employ an ensemble learning approach for the purpose of categorizing Android malware. The first phase entails doing a static analysis of the Android Manifest file contained within the Android Application Package (APK) in order to extract system features such as permission calls, component calls, and intents. Subsequently, the researchers employ the XGBoost methodology, a form of ensemble learning, to identify instances of fraudulent programmes. The primary dataset utilized in this experiment was sourced from the Kaggle platform, namely from their examination of over 6,000 Android applications. The researchers assessed a testing set including 2,000 examples to evaluate both good and bad applications. They employed three distinct feature sets for this evaluation. Subsequently, the remaining data was utilized to create a training set consisting of 6,315 samples.

Flowdriod, a static taint analysis tool,[18] conducts an examination of the byte code and configuration files of applications on the Android platform in order to identify instances of privacy leaks. While Flowdroid has high efficacy and exceptional accuracy, it does not provide inter-component communication. There have been various virus detection methodologies proposed to tackle this matter, yet complete static analysis still exhibits notable limitations. These methods lack the capability to do comprehensive static analysis and are incapable of resolving reflected method calls. In order to conduct an analysis of the security of third-party programmes found on different play stores, the researchers have introduced SAM (Static Analysis Module)[19] as a component of the mobile application verification cluster. The testing and implementation of SAM is conducted on the Android platform.

1233 Android malware samples were categorized into several kinds in a study by F. Alswaina et.al.[20] There are a total of 28 different forms of Android malware that were classified. Machine learning algorithms are given application permissions as input. Some permissions fell into the category of "extremely risky," while others did so under the category of "slightly less dangerous." The authors suggested a method they call an "extremely randomized tree" to digitize these variations and enhance the effectiveness of classification algorithms. The task of feature selection was also met by the suggested approach. The study employed six different categorization algorithms. These include nearest neighbour, nearest tree, ID3 decision trees, RF, neural networks, and bagging techniques. The RF algorithm produces the best classification results. The RF categorization result is 95.97% accurate.

The study by Li et.al.[21] proposes an SVM-based approach for detecting malware on the Android platform, which considers both combinations of dangerous API requests and permission requests. The dataset has a total of 400 Android applications, with 200 being classified as excellent applications sourced from the official Android market, and the remaining 200 being categorized as bad applications sourced from the Drebin dataset. The user did not provide any text to rewrite. The analysis aims to determine the program's degree of risk and, if applicable, categorize it inside the malware classification. State-of-the-art algorithms are employed to detect malware, achieving a remarkable accuracy rate of 99.82% with no instances of false positives. This is accomplished by utilizing only a small portion of the available computational resources and combining a restricted feature set.

G. Suarez-Tangil et.al.[22] proposed a machine learning-based malware detection system for Android that is permission-based. Instead of employing all permissions, the significant permission identification (SIGPID) method allows you to pick the ones that will make it easier to distinguish between harmful software and other malicious software. 135 permissions were downsized to 22 permissions using the suggested approach. 22 permissions were used for classifying data, which led to more successful and quicker outcomes. Additionally, it was underlined that the SVM in the study enabled over 90% of classification success.

Proposed Methodology

The proposed methodology consists of three stages: gathering data, selecting and extracting features, and using machine learning classifiers. Referring to Fig. 3, The proposed work's initial phase is dedicated to data gathering.

A. Data Collection

The dataset being used is crucial for malware detection. Both benign and malicious application samples are gathered in a data collection. The performance of several machine learning techniques is assessed using the debrian-215 dataset[23] and utilized as the dataset for the proposed work. 9,476 benign samples and 5,560 malware samples are included in this collection. The dataset comprises a total of 215 properties, wherein manifest permissions account for 53% of the properties, API call signatures account for 33%, and the remaining 14% are attributed to other factors. The dataset encompasses information pertaining to the features of all applications, wherein these properties are denoted by binary values of either 0 or 1. The values of 0 and 1 are indicative of whether a specific characteristic necessitates authorization.
Figure 2 provides a detailed explanation of the designed methodology incorporated in this study.

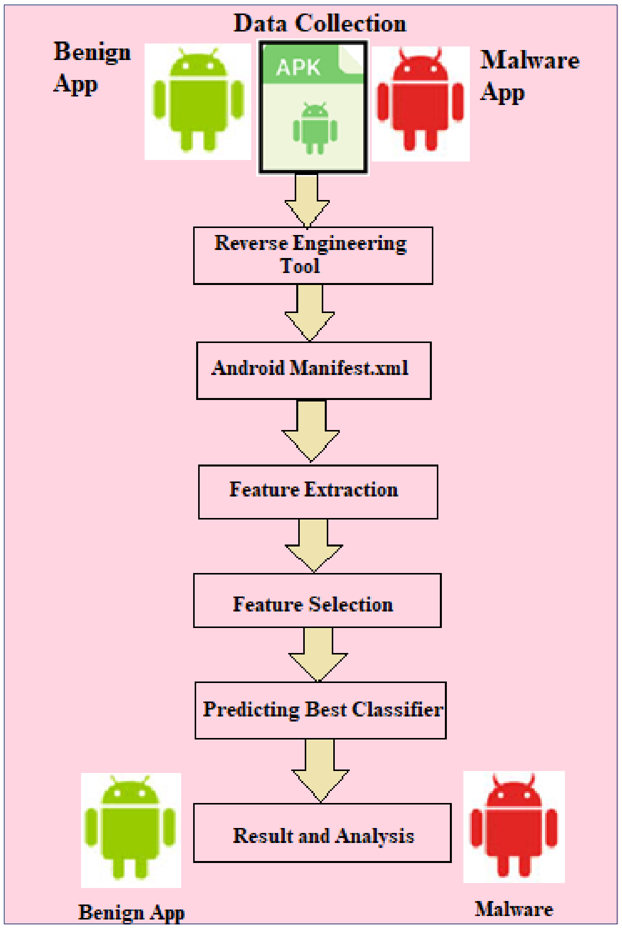


Figure 2. Designed Methodology

B. Preprocessing, Feature Extraction & Feature Selection

Preprocessing

The dataset may include values other than 0 and 1. The dataset is also characterized by a significant number of irrelevant variables as well as a considerable amount of missing information. When doing training for a machine learning model, the presence of missing values has the potential to introduce errors. Preprocessing the dataset is therefore crucial. Numerous libraries for machine learning exist, including ScikitLearn, pandas, Numpy, and others.[24] These libraries contain a wide variety of data preparation tools. For this project, the ScikitLearn Library's SimpleImputer class and average value approach are employed.

Feature Extraction

The Java programming language is commonly employed in the creation of Android applications. The Java code that has been generated is subsequently subjected to the process of compilation, leading to the generation of byte code. The byte code is later converted into DEX byte code. The designated file extension for byte code is ".class". Through the utilization of a dx tool, the discrete .class files are amalgamated into a cohesive dex file, hence facilitating the bundling of the Android programmed as APK content. To facilitate the examination of the APKs, it is important to categorize these Android applications. Apktool, dex2jar, JADX, and other reverse engineering tools are utilized for the purpose of disassembling and analyzing these software applications. The extraction of features from these APKs represents the starting stage in our endeavor. Figure 3 provides insights into the Feature Extraction Technique utilized in the study

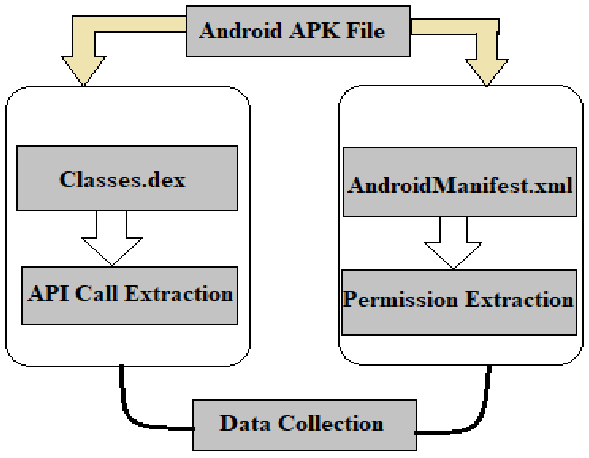


Figure 3. Feature Extraction Technique

Analogous to the compression of files into a zip format, an Android application undergoes the process of archiving. The AndroidManifest.xml file contained within this APK offers a multitude of capabilities that can be utilized for the purpose of static analysis. In order to extract the features from an APK file, it is necessary to employ a reverse-engineering programed.

In this context, the extraction of permission requests from the Android manifest.xml file and API calls from the Classes.dex file is performed, following the example provided in Figure 4. Other traits can likewise be retrieved in a similar manner. Using the feature importance technique, a few key characteristics can be chosen from the extracted features and used for malware detection. The next part goes into detail about feature selection.

Feature Selection

Finding and utilizing the appropriate features is crucial to improve the model's accuracy. Using the SimpleImputer class of the ScikitLearn Library, a dataset is first preprocessed in this research phase. After preprocessing, features are retrieved, and using the feature importance technique, the features that are significant in determining whether or not an application is malicious are selected. As can be seen in below Fig. 4, a total of 10 features were chosen based on their score values. significant attributes. Figure 4 displays crucial features along with their respective scores.

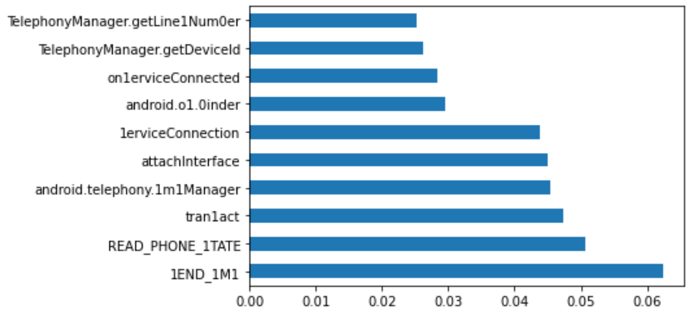


Figure 5. Important Features

The feature importance for each feature can be determined by employing a model's feature importance technique. The relative importance of each feature with respect to the output variable is assessed by providing a numerical score to each feature, where higher scores correspond to greater significance. The Scikit-Learn toolkit for machine learning includes a Tree Based Classifier, which provides feature significance as a default feature.
This relevant feature selection method reduces 215 features to 10 features that are crucial for differentiating between malicious and benign programs.

Machine Learning Algorithm

The identification of malicious software, commonly referred to as malware, is predominantly dependent on the utilization of machine learning methodologies. The machine learning model undergoes training and testing using the selected Important features.

The aforementioned characteristics are presented as input to a machine learning model, which employs many classifiers to ascertain the level of risk associated with an application.

To evaluate the effectiveness of the classifiers using various parallel combination approaches, a 10-fold cross-validation methodology is employed. The dataset is divided into ten distinct and non-overlapping segments, as indicated by its nomenclature.

The components can be classified into parts 1 through 3, and further expanding to section 10. The evaluation methodology at each stage employs three segments as the testing dataset, while the remaining seven segments are allotted for training the model. The utilization of the cross-validation training dataset is integral to the training process of the machine learning model. Subsequently, the predicted outcomes are compared with the validation dataset to evaluate the model's correctness. The justification for employing this methodology is to ensure that our strategic approach effectively enables the detection of unidentified hazardous applications.

To achieve best results, it is customary to divide a dataset into two separate sets: a training set and a testing set. The training set typically accounts for 70% of the dataset, while the testing set accounts for the remaining 30%. The algorithms are subjected to testing using the remaining 30% of the dataset after being trained on 70% of the data. The technique of K-fold cross-validation is extensively utilized in the domain of machine learning to assess the efficacy of models. In this methodology, the dataset is partitioned into K subsets, where K is commonly designated as 10. The model is subsequently trained and assessed K times, with each iteration utilizing a distinct subset as the validation set and the remaining subsets as the training set. This procedure facilitates a thorough evaluation of the model's proficiency and aids in addressing concerns pertaining to overfitting and bias.

The methodologies utilized in this research encompass Linear Regression, K-Nearest Neighbours (KNN), Naive Bayes, Decision Tree, Random Forest, Support Vector Machine (SVM), Linear Support Vector Machine (Linear SVM), XGBoost, Adaboost, and Gradient Boosting. The subsequent section of this research study presents a comprehensive examination and evaluation of the outcomes and analyses pertaining to each of the aforementioned algorithms.

Evaluation Criteria

Various criteria are employed to assess the efficacy of distinct machine learning algorithms: The confusion matrix is a commonly employed method for assessing the effectiveness of a classifier. The numerical numbers within this matrix provide a succinct depiction of the quantities of precise and imprecise predictions. The determination of false positive and false negative rates necessitates the application of a particular methodology. Table 1 illustrates the Confusion Matrix.

Table 1 Confusion Matrix

Confusion Matrix **Predicted** **Observed**			
	Positive	**Negative**	**Total**
Positive	TP(p)	FN(q)	p+q
Negative	FP(r)	TN(s)	r+s
Total	p+r	q+s	P+q+r+s

True Positive Ratio (TPR)

The accuracy rate of identifying malicious APKs, expressed as the ratio of successfully categorized malicious APKs to the total number of harmful APKs in the dataset.
TPR = p / p + q

True Negative Ratio (TNR)

The determination of the proportion of accurately categorized non-harmful APKs can be achieved by dividing the entire count of correctly classified non-harmful APKs by the overall count of non-harmful APKs present in the dataset.
TNR = s / r + s

False Positive Ratio (FPR)

The fraction of incorrectly labelled benign APKs relative to the total number of benign APKs in the dataset.
FPR = 1 − TNR = 1 – Specificity

FPR = r / r + s

False Negative Ratio (FNR)

The proportion of misclassified hazardous applications in the dataset relative to the total number of harmful applications.
FNR = q / p + q

Accuracy (Acc)

The accuracy of predictions is defined as the ratio of correct predictions to the total number of predictions generated by the dataset.

Acc = p + s/ (p + q + r + s)

Area Under Curve

The measure in question is utilized as a means of assessing the classifier's efficacy. Value of 0.5 represents random guesses, while a value of 1 signifies perfect predictions. Figure 5 illustrates the Area Under the ROC Curve.

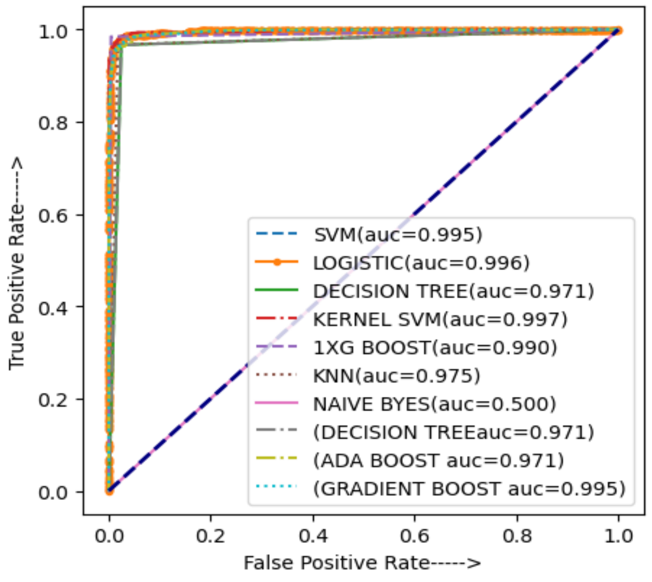


Figure 5: Area Under Curve

Findings

Table II presents the performance metrics for each of the assessed algorithms.

Table 2 Performance of Classifiers

Name of Algorithm	False Positive Rate (%)	False Negative Rate(%)	AUC	Accuracy
SVM	0.033	0.026	0.9774	97.73
LOGISTIC REGRESSION	0.030	0.026	0.9773	97.66
RANDOM FOREST	0.012	0.036	0.9781	98.33
KERNEL SVM	0.013	0.037	0.9768	97.96
1XG BOOST	0.008	0.015	0.9900	98.72
KNN	0.031	0.032	0.9746	97.75
NAÏVE BYES	0	1	0.5	69.40
DECISION TREE	0.043	0.033	0.9710	97.53
ADA BOOST	0.033	0.041	0.9700	97.12
GRADIENT BOOST	0.024	0.044	0.9713	97.23

Result and Discussion

Support Vector Machine

The SVM algorithm showcased robust performance in our study, achieving a False Positive Rate (FPR) of 0.033 and a False Negative Rate (FNR) of 0.026. With an impressive Area Under the ROC Curve (AUC) of 0.9774, the SVM model demonstrated its efficacy in effectively distinguishing between positive and negative instances. The overall accuracy of 97.66% underscores its reliability and competence in addressing the study's objectives. The balanced trade-off between false positives and false negatives, coupled with the high AUC score, highlights the excellent discriminative power of the SVM model.

Logistic Regression

Parallel to the robust performance of SVM, Logistic Regression demonstrated commendable results, featuring a slightly lower False Positive Rate (FPR) of 0.030 and an identical False Negative Rate (FNR) of 0.026. Possessing an AUC value of 0.9773 and an accuracy rate of 97.66%, Logistic Regression exhibited reliability and effectively addressed the challenges presented by the task. The comparable performance to SVM, as indicated by the similar AUC values, underscores the logistic regression model's high ability to distinguish between positive and negative instances.

Random Forest

Random Forest demonstrated impressive performance in our study, boasting a low False Positive Rate (FPR) of 0.012 and a moderate False Negative Rate (FNR) of 0.036. With a high Area Under the ROC Curve (AUC) value of 0.9781 and an exceptional accuracy of 98.33%, Random Forest showcased its effectiveness in accurately classifying instances. The combination of a low false positive rate, a reasonable false negative rate, and a high AUC score emphasizes the robust discriminative power and overall competence of the Random Forest algorithm in addressing the objectives of the study.

Decision Tree

The Decision Tree algorithm exhibited an accuracy of 97.53%, accompanied by a marginally higher False Positive Rate (FPR) of 0.043 and a False Negative Rate (FNR) of 0.033. While boasting good discriminatory power with an AUC of 0.9710, it slightly lags behind SVM and Logistic Regression in overall performance. Despite its accurate predictions, the Decision Tree model shows a trade-off with slightly elevated false positive and false negative rates when compared to the aforementioned algorithms. The AUC score of 0.9710 indicates acceptable discriminative power, reinforcing its capability to distinguish between positive and negative instances.

Kernel SVM

Distinguishing itself with a notably low False Positive Rate (FPR) of 0.013, the Kernel SVM algorithm demonstrated an impressive accuracy of 97.96%. Alongside a False Negative Rate (FNR) of 0.037 and an AUC of 0.9768, it emphasized its prowess in effectively discriminating between positive and negative instances. Kernel SVM stands out not only for its remarkable accuracy but also for its distinctly low false positive rate, underscoring its precision in minimizing misclassifications. The high AUC score of 0.9768 further reinforces its overall strong discriminative power, making it a standout performer in the study.

XG Boost

XG Boost emerged as an unequivocal standout performer in our study, showcasing the lowest False Positive Rate (FPR) at 0.008 and an impressively low False Negative Rate (FNR) of 0.015. With an outstanding AUC value of 0.9900 and a remarkable accuracy of 98.72%, XG Boost demonstrated exceptional effectiveness in meeting the study's objectives. Its unparalleled combination of the lowest false positive rate, highest accuracy, and an impressive AUC score underscores its robustness and efficacy in accurately distinguishing between positive and negative instances, making it a highly recommended choice for deployment in similar tasks.

K-Nearest Neighbors

KNN showcased reliable performance in our study, exhibiting a False Positive Rate (FPR) of 0.031, a balanced False Negative Rate (FNR) of 0.032, and an AUC of 0.9746. The algorithm's accuracy reached 97.75%, firmly establishing its competence in the context of the study. KNN's consistent and reliable performance is evident in its ability to maintain a balance between false positive and false negative rates, supported by a commendable AUC score. These results highlight the algorithm's robustness in accurately classifying instances and reinforce its suitability for application in similar contexts.

Naive Bayes

In a distinctive profile, Naive Bayes exhibited an unusual pattern by displaying no false positives (False Positive Rate, FPR: 0). However, it encountered challenges marked by a high False Negative Rate (FNR) of 1, leading to an overall accuracy of 69.40%. The AUC value of 0.5 further emphasizes chance-level performance in discrimination. Naive Bayes' unique characteristic of avoiding false positives is noteworthy, but its struggle with a high false negative rate contributes to a lower overall accuracy. The AUC score of 0.5 suggests that the model's discriminatory power is no better than random chance, indicating limitations in its effectiveness for the specific task at hand.

Ada Boost

Ada Boost demonstrated competitive performance in our study, showcasing a False Positive Rate (FPR) of 0.033 and a slightly higher False Negative Rate (FNR) of 0.041. The model achieved an AUC of 0.9700 and an accuracy of 97.12%. These results underscore Ada Boost's efficacy in the given task, as it strikes a commendable balance between false positive and false negative rates. The AUC score of 0.9700 further confirms its effectiveness in discrimination, positioning Ada Boost as a robust algorithm with competitive performance in the context of the study.

Gradient Boost

Gradient Boost demonstrated solid performance in our study, featuring a low False Positive Rate (FPR) of 0.024 and a marginally higher False Negative Rate (FNR) of 0.044. With an Area Under the ROC Curve (AUC) value of 0.9713 and a commendable accuracy of 97.23%, Gradient Boost showcased its effectiveness in classification tasks. The low false positive rate indicates a precise identification of negative instances, while the AUC score of 0.9713 highlights its overall good discriminative power. Despite a slightly elevated false negative rate, Gradient Boost's strong accuracy and competitive performance make it a valuable candidate for tasks requiring accurate and reliable predictions.

The algorithm that demonstrates superior performance among the ten evaluated in this study is XG Boost. XG Boost demonstrates remarkable performance by presenting the most favorable metrics, including the lowest False Positive Rate (FPR) at 0.008, the lowest False Negative Rate (FNR) at 0.015, and an exceptional Area Under the ROC Curve (AUC) value of 0.9900. Moreover, XG Boost achieves an impressive accuracy rate of 98.72%. This amalgamation of the minimal false positive and false negative rates, along with a high accuracy and an outstanding AUC score, accentuates its robustness and efficacy in accurately discerning between positive and negative instances. Consequently, based on the information provided, XG Boost emerges as the top-performing algorithm among the evaluated models in the specific task under consideration.

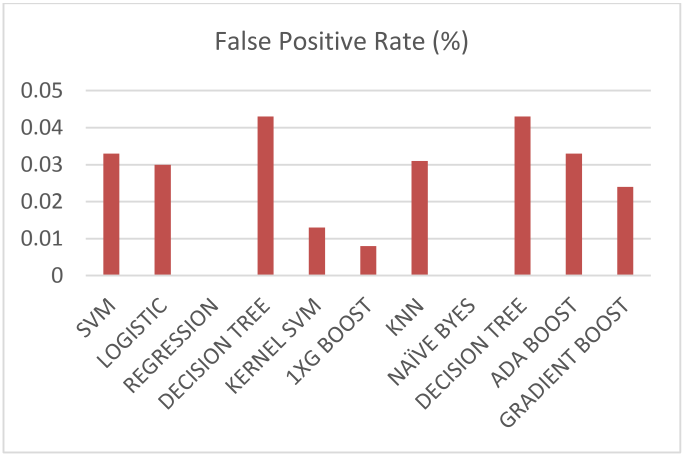


Figure 6: False Positive Rate

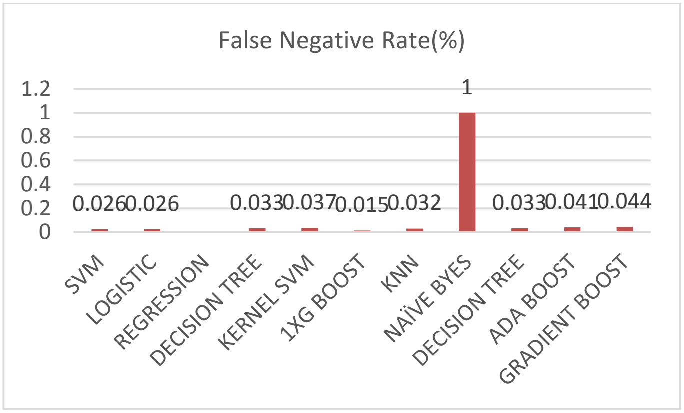


Figure 7: False Negative Rate

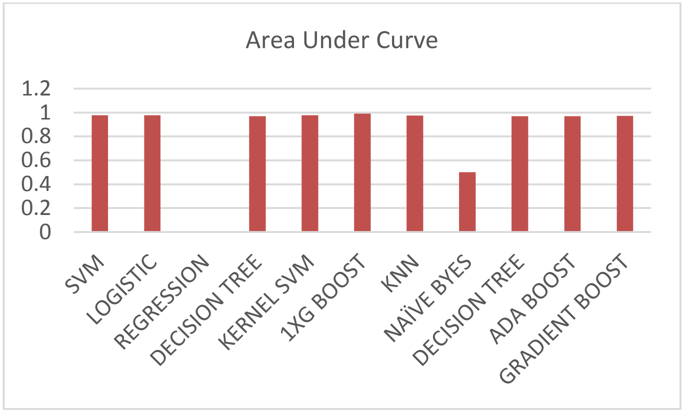


Figure 8: ROC Area Under Curve

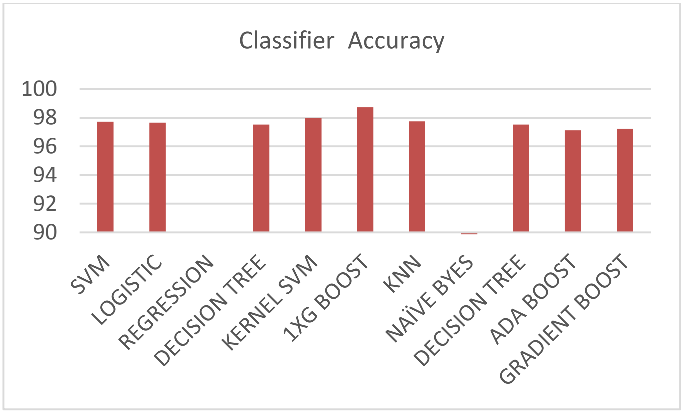


Figure 9: Classifier Accuracy

XGBoost, short for Extreme Gradient Boosting, represents an advanced and expanded iteration of the Boosting methodology, specifically derived from the Gradient Boosted Decision Tree (GBDT) technique. The core principle of this methodology is defined by the inclusion of additive training, wherein a new function is introduced to the model while maintaining the integrity of the present model.[17]

XGBoost, has a mathematical definition that is

$$\hat{b_l} = \sum_{p=1}^{P} f_p(a_i), f_p \in \mathcal{F} \ldots (1)$$

In this context, P represents the quantity of trees, while f represents a function within the function space F. The function space F encompasses the entirety of viable classification and regression trees. The optimization problem requires the provision of the target function to be optimized.

$$\text{obj}(\theta) = \sum_{i}^{q} l(b_i, \hat{b_l}) + \sum_{p=1}^{P} \Omega(f_p) \ldots (2)$$

Ω Can be defined as

$$\Omega(f) = bN + \frac{1}{2}\lambda \sum_{c=1}^{N} M_c 2 \ldots (3)$$

In this context, the variable N denotes the quantity of leaves, whereas M represents the weight vector of a leaf node. The anticipated tree structure will exhibit greater simplicity as the values of "and" increase in magnitude.

The following benefits of using the XGBoost approach in real applications:

L1 and L2 regularisation can penalise a complex model, preventing over-fitting.

XGBoost is capable of effectively handling many types of sparse data patterns through the utilization of a segmentation search algorithm. The proposed methodology integrates sparse data sensing methods to tackle issues pertaining to missing data and the handling of sparse data, such as the utilization of one-hot coding for data encoding.

The XGBoost implementation has a block structure that facilitates parallel learning, enabling efficient computation by leveraging several CPU cores. The data is organized and stored within a memory component known as a block, which distinguishes it from alternative methodologies. The reusability of the data architecture allows for its use in subsequent cycles, obviating the need for recalculating all variables anew, hence reducing computational time.

Discussion

It is clear from the testing results of different classifiers that all of the algorithms worked effectively. With the exception of Naive Byes, practically all algorithms perform above 97% accurately. XG Boost demonstrates the highest level of accuracy among all models, achieving a notable accuracy rate of 98.72%. Furthermore, it has the most minimal occurrence of false positives and false negatives. The Random Forest approach exhibits a considerable degree of precision, specifically 98.34%, accompanied with a false positive rate of 0.012. This performance is surpassed only by the XG boost algorithm.

The XG Boost algorithm exhibits a higher level of performance in relation to the false negative rate, with the support vector machine (SVM) classifier closely trailing behind. The gradient

boost method is characterized by a relatively elevated percentage of false negatives, while the decision tree approach is associated with a comparatively higher incidence of false positives.
Based on the overall outcomes, the XG boost algorithm emerges as the most superior and precise technique, while the random forest algorithm ranks second.

The analysis of the experimental data shown in the table reveals that XG Boost demonstrates the best level of accuracy, reaching 98.72%. Additionally, it achieves an AUC value of 0.99.. This indicates that XG Boost outperforms all other classifiers utilized in this experiment, establishing it as the most effective classifier.

Conclusion

Our research delineates a sustained and significant interest in the examination of Android applications, spanning from 2014 to the present day. The evolution of this field is discernible through our rigorous methodology, inclusive of an extensive literature review and systematic analysis. Our findings assert the superiority of dynamic analysis over static analysis for effective feature extraction in identifying malicious applications. This preference stems from the escalating complexity of harmful app behaviors and the advanced functionalities exhibited by genuine applications. However, it is imperative to acknowledge that dynamic feature extraction, while potent, entails a time-intensive process, particularly when evaluating a large volume of applications. A noteworthy observation is the increasing reliance on proficient artificial intelligence and machine learning methodologies, leading to a substantial augmentation in the identification of characteristics and applications. This research significantly contributes to the ongoing discourse on enhancing Android application security, offering valuable insights for future investigations in this evolving landscape.

The primary objective of this study was to assess the efficacy of various machine learning techniques utilizing the debrian-215 dataset. Training on a merged dataset containing 15,036 samples of both malware and benign software, we evaluated ten machine learning methods employing performance metrics such as accuracy, area under the curve (AUC), false positive rate, and false negative rate. Remarkably, the XGBOOST technique exhibited exceptional performance when compared to alternative algorithms, achieving an impressive accuracy rate of 98.72% and an area under the receiver operating characteristic (ROC) curve of 0.9900. A comprehensive analysis of experimental data strongly suggests that XGBOOST outperforms alternative algorithms across key metrics, specifically within the domain of malware detection. These findings provide valuable insights to the field of machine learning for cybersecurity, emphasizing the efficacy of XGBOOST as a robust tool for enhancing malware detection capabilities.

Limitation

This study specifically emphasizes static analysis for Android malware detection, rather than centering on dynamic malware detection. The superiority of dynamic analysis in Android malware detection becomes evident when contrasted with static methodologies. Through the execution of applications within a controlled environment, dynamic analysis captures real-time behavioral nuances, revealing hidden malicious activities that may elude static examinations. This approach proves particularly adept at identifying emerging threats, intricate logic, and encrypted elements, showcasing its adaptability to the dynamic Android malware landscape. The context-aware attributes of dynamic analysis, encompassing interactions with device resources, network communication, and user inputs, significantly contribute to the precision in distinguishing between benign and malicious behaviors. While static analysis remains a crucial component in the broader malware detection strategy, the strength of dynamic analysis lies in its capacity to unveil an application's true nature during runtime, establishing a robust defense against sophisticated and ever-evolving Android malware threats.

Conflict of interest statement

Authors certify that it is their own work, takes complete responsibility for any text plagiarism and declare that they do not have any conflict of interest for publication of this work.

References

1. I. Alsmadi, R. Burdwell, A. Aleroud, et al. Mobile and Wireless Security: Lesson Plans. In *Practical Information Security*; Springer International Publishing, Cham, **2018**; pp 159–179.
2. X. Wang, W. Wang, Y. He, et al. Characterizing Android apps' behavior for effective detection of malapps at large scale. *Futur. Gener. Comput. Syst.* **2017**, 75, 30–45.
3. W. Wang, M. Zhao, Z. Gao, et al. Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions. *IEEE Access* **2019**, 7, 67602–67631.
4. D. Wermke, B. Reaves, N. Huaman, et al. A large scale investigation of obfuscation use in google play. In *ACM International Conference Proceeding Series*; Appl. Conf, **2018**; pp 222–235.
5. W. Wang, Z. Gao, M. Zhao, et al. DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features. *IEEE Access* **2018**, 6, 31798–31807.
6. X. Zeng, G. Xu, X. Zheng, Y. Xiang, W. Zhou. E-AUA: An efficient anonymous user authentication protocol for mobile IoT. *IEEE Internet Things J.* **2019**, 6 (2), 1506–1519.
7. G. Xu, Y. Zhang, A.K. Sangaiah, et al. CSP-E2: An abuse-free contract signing protocol with low-storage TTP for energy-efficient electronic transaction ecosystems. *Inf. Sci. (Ny).* **2019**, 476, 505–515.
8. B. Urooj, M.A. Shah, C. Maple, M.K. Abbasi, S. Riasat. Malware Detection: A Framework for Reverse Engineered Android Applications Through Machine Learning Algorithms. *IEEE Access* **2022**, 10, 89031–89050.
9. D.O. Sahin, S. Akleylek, E. Kilic. LinRegDroid: Detection of Android Malware Using Multiple Linear Regression Models-Based Classifiers. *IEEE Access* **2022**, 10, 14246–14259.
10. G. Jacob, P.M. Comparetti, M. Neugschwandtner, C. Kruegel, G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer, **2013**; Vol. 7591 LNCS, pp 102–122.
11. H. Bai, N. Xie, X. Di, Q. Ye. FAMD: A fast multifeature android malware detection framework, design, and implementation. *IEEE Access* **2020**, 8, 194729–194740.
12. Z. Aung, W. Zaw. Permission-Based Android Malware Detection. *Int. J. Sci. Technol. Res.* **2013**, 2 (3), 228–234.
13. A. Pektaş, M. Çavdar, T. Acarman. Android malware classification by applying online machine learning. In *Communications in Computer and Information Science*; Springer, Cham, Switzerland, **2016**; Vol. 659, pp 72–80.
14. Y. Suleiman, S. Sezer, G. McWilliams, I. Muttik. New Android malware detection approach using Bayesian classification. In *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*; IEEE, **2013**; pp 121–128.

15. P. Rovelli, Ý. Vigfússon. PMDS: Permission-based malware detection system. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Prakash, A., Shyamasundar, R., Eds.; Springer, Cham, Switzerland, **2014**; Vol. 8880, pp 338–357.
16. N. Milosevic, A. Dehghantanha, K.K.R. Choo. Machine learning aided Android malware classification. *Comput. Electr. Eng.* **2017**, 61, 266–274.
17. D. Congyi, S. Guangshun. Method for Detecting Android Malware Based on Ensemble Learning. In *ACM International Conference Proceeding Series*; Mach. Learn. Technol, Beijing, China, **2020**; pp 28–31.
18. S. Arzt, S. Rasthofer, C. Fritz, et al. FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Notices*; ACM, **2014**; Vol. 49, pp 259–269.
19. A. Armando, G. Bocci, G. Chiarelli, et al. Mobile app security analysis with the MAVeriC static analysis. *J. Wirel. Mob. Networks, Ubiquitous Comput. Dependable Appl.* **2014**, 5 (4), 103–119.
20. F. Alswaina, K. Elleithy. Android Malware Permission-Based Multi-Class Classification Using Extremely Randomized Trees. *IEEE Access* **2018**, 6, 76217–76227.
21. W. Li, J. Ge, G. Dai. Detecting Malware for Android Platform: An SVM-Based Approach. In *Proceedings - 2nd IEEE International Conference on Cyber Security and Cloud Computing, CSCloud 2015 - IEEE International Symposium of Smart Cloud, IEEE SSC 2015*; New York, NY, USA, **2016**; pp 464–469.
22. G. Suarez-Tangil, S.K. Dash, M. Ahmadi, et al. DroidSieve: Fast and accurate classification of obfuscated android malware. In *CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*; Scottsdale, AZ, USA, **2017**; pp 309–320.
23. U. V Nikam, V.M. Deshmuh. Performance Evaluation of Machine Learning Classifiers in Malware Detection. In *2022 IEEE International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE)*; IEEE, **2022**; pp 1–5.
24. J. Li, L. Sun, Q. Yan, et al. Significant Permission Identification for Machine-Learning-Based Android Malware Detection. *IEEE Trans. Ind. Informatics* **2018**, 14 (7), 3216–3225.