JIMDT, Vol. 6, 2024
RESEARCH ARTICLE
https://doi.org/10.5564/jimdt.v6i1.3383

Android malware detection using machine learning

Usukhbayar Baldangombo[1*], Zolzaya Kherlenchimeg[1*], Ugtakhbayar Naidansuren[1*],

[1] *School of Information Technology and Electronics, National University of Mongolia, Ulaanbaatar 14201, Mongolia*

**Corresponding author: usukhbayar@num.edu.mn; ORCID:0000-0002-0789-6618*

Article Info: Received: 2024.09.15; Accepted: 2024.10.01; Published: 2024.12.26

Abstract: In this study, we present a static Android malware detection system using data mining and machine learning techniques that includes five feature selection methods: Information Gain, Binormal Separation, Chi-squared, Relief, Principal Component Analysis, and four machine learning algorithms: Naive Bayes, SVM, J48, and Random Forest. To overcome the lack of usual signature-based antivirus products, we use static analysis to extract valuable features of Android applications. We extract permission and API call features of Android APK files. Afterward, the feature selection methods are used to select valuable feature subsets. This feature subset is selected by conducting extensive experimental analysis in which experimental thresholds select various feature subsets, and the subset trains the machine learning algorithms to find the best model. By adopting the concepts of machine learning and data mining, we construct a malware detection system that has an Overall Accuracy of 96%

Key words: Android malware, mobile malware, machine learning, malware analysis, static analysis

1. Introduction

Phone companies shipped 326.1 million smartphones worldwide in the fourth quarter of 2023, as reported in "Smartphone OS Market Share, 2023 Q4" by the IDC Quarterly Mobile Phone Tracker [1]. As stated in this report, the Android operating system is the most popular smartphone platform, with 70.16% of the market share of smartphones, 29.14% for iOS, 0.02% for Windows, and 0.68% for others in the fourth quarter of 2023. As the Android operating system possesses the largest mobile phone market share, malware, especially targeting Android devices, has grown and become more sophisticated [2]. In the third quarter of 2023 [3], Kaspersky Lab experts detected 1,333,605 mobile malicious programs. Notably, there was a dramatic increase in attacks involving mobile ransomware from the Trojan-Ransom.AndroidOS.Egat family, with the number of users targeted by this malware rising more than 13-fold compared to the previous quarter. The effect of malware on mobile devices has become a real danger in that it can steal credential information from the device, sniff user activity and location, overbill users by sending unwanted SMS and MMS to contacts, launch denial of services attacks from user devices, and overloading device resources such as memory, battery, and storage [4]. The malware analysis can roughly be divided into static and dynamic analysis [5]. In the static analysis, the code and structure of a program are examined without running the program. In contrast, the program is executed in a real or virtual environment in dynamic analysis. Therefore, the dynamic analysis method suffers from

significant system overhead when executing and monitoring the application. However, the most common malware detection is a signature-based method that is the core building block of most commercial antivirus programs. Antivirus programs based on this method are ineffective against zero-day attacks or previously unknown malware. The inability of traditional signature-based detection approaches to catch this new breed of malware has shifted the focus of malware research to find more generalized and scalable features that can identify malicious behavior as a process instead of a single static signature [6]. Machine learning-based approaches have become one of the most attractive directions for detecting zero-day malicious applications. Many works have recently proposed Android malware detection methods, especially those based on data mining and machine learning algorithms [7]. For example, Drebin [8] extracts permissions, APIs, and IP addresses as features and uses the Support Vector Machine (SVM) algorithm to learn a classifier from the existing ground truth datasets, which can detect unknown malware. Droidmat [9] uses permissions and intent mining with KNN (KNearest Neighbor) to detect malware. In addition, DoridAPIMiner [10] focuses on providing several lightweight classifiers based on the API-level features. These approaches use a large number of extracted features, which has led to some significant limitations in their works because some features are irrelevant or redundant. The machine learning-based classification can lead to classifier drawbacks, such as misleading the learning algorithm, over-fitting, reducing generality, and increasing model run-time [11]. Furthermore, the raw features must be preprocessed to select a subset of features to use in the learning scheme, as well as the performance of learning algorithms can frequently be improved by preprocessing, even though the algorithms themselves try to select attributes appropriately and ignore irrelevant or redundant ones [12]. This work introduces a mobile malware detection system based on feature selection and machine learning algorithms. This work conducts experiments to address the issues related to selecting appropriate feature sets and reducing irrelevant or redundant ones using the five feature selection algorithms and the five most frequently used machine learning algorithms in this field. We evaluate several models using feature subsets ranked and selected by various feature selection algorithms. Through this process, we identify the model that achieves the highest detection performance based on the evaluation criteria. The contributions of this work are twofold: First, we present a machine learning-based approach for Android malware detection that relies on permission and API call features. This method is lightweight, computationally efficient, and suitable for deployment in real-world malware detection scenarios. Second, we identify the most effective feature subsets for detecting newly emerging malicious applications. These subsets not only enhance detection accuracy but also provide deeper insights into application behavior The rest of this work is organized as follows. Section 2 presents the necessary background about Android malware. Related works are presented in Section 3. Section 4 describes the malware detection system architecture, research methodology, and experimental design. In Section 5, we also evaluate the outcome of the experimental analysis and its results. Finally, Section 6 concludes this work.

2. Research materials and data

2.1. Background

Before we discuss the details of our proposed framework, it is essential to understand how Android malware is structured and works. This section introduces Android malware, including its type, propagation techniques, and threats.

2.1.1. Android malware

In the following, we first describe some of the terms commonly used to describe the characteristics and properties of malware. Then, we consider malware propagation techniques

and threats. Android malware can be divided into the following terms [5]:

- Trojan:It masquerades as a benign app but performs harmful activities without the users' consent or knowledge. It offers the user useful functionalities but performs malicious activities in the background. For example: FakeNetFlx, Zsone, Zitmo, Spitmo, and Fakeplayer.
- Backdoor: It allows entry to the system, bypassing all security procedures and facilitating the installation of other malicious applications. Backdoor uses root-level exploits to gain superuser privilege to hide itself from other security apps, or worse, it may also disable them. For example, Basebridge, KMin, Obad are well-known backdoor applications.
- Worm: The worm app can create an exact or similar copy and spread it through the network or removable media. For example, Bluetooth worms can automatically send copies to other devices. Android.Obad is one such example that can spread malicious apps via Bluetooth.
- Bot: These applications enable an attacker to remotely control the device from a bot-master server through a series of commands. A network of such bots is called a botnet. The attacker commands the infected system to send confidential information to a remote server or something as complex as causing a denial-of-service attack. The bot can also include commands to download malicious payloads automatically. Geinimi, Anserverbot, and Beanbot are well-known examples of bots.
- Spyware: It looks like a typical application, but it has hidden functions to surreptitiously monitor contacts, messages, location, etc., to perform malicious actions later. They may also send all the collected information to the remote server. For example, Nickyspy, GPSSpy are spyware.
- Adware: It performs tasks such as monitoring the victim's personal data, showing unwanted advertisement content, annoying victim users with aggressive advertisement push, and showing and tempting the victim to download and install potentially harmful applications based on the victim's personal data. Plankton is an example of well-known adware.
- Ransomware: This type of app locks the victim's device by making it completely inaccessible or encrypting data until the user pays some ransom. For example, FakeDefender.B.

2.1.2. Malware propagation techniques

Here, we summarize the propagation techniques commonly employed by Android malware. One of the most prevalent methods is repackaging, where attackers download popular benign applications, inject them with malicious payloads, and re-upload the repackaged versions to various Android marketplaces. Another technique is the update attack, which does not initially include a malicious payload. Instead, the malware is delivered later by disguising itself as a legitimate app update. A third method, drive-by download, mirrors traditional web-based attacks. It involves redirecting users to download malware through deceptive means such as aggressive app advertisements or malicious QR codes [13].

- Repackaging: One of the most common techniques malware authors use to piggyback malicious payloads into popular applications. Malware authors may locate and download popular applications, disassemble them, enclose malicious payloads, reassemble, and then submit the new apps to official or alternative Android markets. Users could be vulnerable by being enticed to download and install these infected apps.

- Drive-by Download: This technique employs traditional drive-by downloads for Android devices, enticing users to download interesting or attractive apps. An attacker can use social engineering, aggressive advertisements, and clickjacking attacks to make users mistakenly download malicious apps. When a user visits a malicious URL, it downloads a malicious application automatically. Alternatively, it may disguise itself as a legitimate one to get permission from the user to install itself. For example, GTracker malware has an application advertisement library. After clicking on that advertisement link, the user is redirected to a website that displays a message to download an app that can save the device's battery. However, that downloaded application is malware that subscribes to premium-rate services without the user's knowledge.

- Update attacks: An application can also embed a malicious payload into the resources as an APK/JAR file, either encrypted or plain format. After the installation, when the application executes, it optionally decrypts the payload. If the payload is a jar file, using the DexClassLoader API, it dynamically loads the payload into the Dalvik VM to execute it. Otherwise, instead of including a malicious payload within the application, malware developers include only an update component that downloads the malicious payload at run time after the application is installed. Hence, scanning the source code will not be able to detect the malware because there is no malicious code within the application.

2.1.3. Malware threats

Here, Android malware is categorized by its carried payloads into the following:

- Privilege Escalation: The Android platform is a complicated system consisting of the Linux kernel and the entire Android framework with over 90 open-source libraries. This attack is done by leveraging publicly available software vulnerabilities to gain root access to the device. It can also happen by exploiting one or more vulnerable components of an app that uses dangerous permissions granted to it.

- Remote Control: Smartphones can even behave as bots under a remote server's control. After infection, the device can receive commands from the server and perform the corresponding action. The action can be sending spam mail, sending SMS, rooting the phone, etc.

- System Damage: Some malware successfully brings the device to a hanging state, where the mobile does not operate normally. They may even block calling functionality in mobile devices or root the Android OS to drain the battery, which is also one of the consequences

- Financial Loss: Sending SMS/MMS messages to premium rate numbers or dialing phone calls to premium rate numbers without the user's knowledge in the background incurs a monetary loss to the mobile device user.

- Information Leakage: Some malware can enable an attacker with the capability to browse through confidential information like SMS/MMS, contact details, call logs, emails, etc.

3. Methodology

3.1. Related works

In this section, we consider related works. Researchers have recently focused on data mining and machine learning to detect unknown malware.

Aswini et al. [14] proposed a static analysis approach to detect Android malware by mining prominent permissions from the Android Manifest.xml file. After extracting permissions from 436 Android package files, feature pruning was applied to examine the accuracy with respect to the feature length. However, the proposed approach cannot investigate the application code deeply for possible malicious behavior. Enck et al. [15] proposed Kirin, which reads application permission requirements during installation and checks them against predefined security rules. Sanz et al. [16] proposed a machine-learning method for automatic Android application categorization and malware detection. This method analyzes different sets of the application's features extracted from the Android Manifest.xml file, the source code files (the frequency of occurrence of the printable strings), and the Android Market (permissions, rating, and number of ratings). Machine learning algorithms were applied, such as Decision Trees, K-Nearest Neighbor, Bayesian Networks, Random Forest, and Support Vector Machines. For the results, Bayesian Networks were reported as the best classifier, Random Forest was the second, and Decision Trees were the worst. Sahs Khan presented a system that uses the extracted permissions and the control flow graphs from benign applications to train a one-class Support Vector classifier. The classifier was always trained to give a positive for the trained data and a negative for the tested data, which is adequately different from the trained ones. The team used kernels over binary vectors, strings, sets of features, non-standard permissions, and applications. The Support Vector classifies applications based on the requested permissions, which allowed it to report all the malware, but also incorrectly reported half of the benign applications as malicious. Arp et al. [8] proposed DERBIN, a lightweight static analysis framework that extracts features from the application's Android Manifest.xml and disassembles code to generate a joint vector space. Support Vector was applied to the dataset to learn a separation between the two classes of applications. The system successfully reported 94% of the malware with a false positive rate of 1%. Felt et al. [17] developed Stowaway, a web-based tool that calculates the permissions used by API calls and compares them with the permissions requested by the application from the manifest file. The work mentioned applications' extensive use of Java reflections as a major reason for difficulty tracing API calls and a limitation of static code analysis. Crowdroid [18] analyzes application system call patterns to detect malware. In this work, the authors proposed that DroidAPIMiner extract API calls using a modified Androguard tool, and different classifiers are evaluated using the feature set. They achieved an accuracy of 99% and a false positive rate of 2.2% using a k-NN classifier. Gibler et al. [19] applied static analysis to the Android API to find permission checks. Their permission map includes unreachable internal methods within the system process across the RPC boundary, which we excluded because applications cannot access them. Seo et al. [20] proposed DroidAnalyzer, which uses permissions, dangerous APIs, and keywords associated with malicious behaviors to detect potential malicious scripts in Android applications. Lists of common malicious APIs and keywords were collected by static analysis from a large dataset of Android malware. These lists are used in the algorithm of DroidAnalyzer, which depends on the keyword-searching technique. The lists define suspicious keywords used in malicious actions such as root exploiting, leakage of user personal data, and costing money through sending SMS or calling premium numbers. Malware from different families was analyzed to build up those lists of malicious and suspicious keywords, APIs, and commands. Peiravian Zhu [21] extracted permissions from 130 permissions and 1,326 API calls and then used three classification methods to build a detection model. Droidminer [22] uses a similar idea to mine API sequences, which can represent software behavior patterns automatically. However, they introduce a different algorithm to generate frequent sequences and do more detailed experiments about the machine learning process. Zhang et al. [23] proposed DroidSIFT, which has unique design features regarding the distance among API dependency graphs. It builds the API dependency graphs G for each application and then constructs the feature vector of the application. The features represent the similarity of the

graph G with a reference database of graphs of known benign and malware applications. Finally, the feature vectors are used in anomaly or signature detection.

4. The results

4.1. Malware detection system architecture

In this section, we describe the architecture of our Android malware detection system. This system is to accurately detect new Malware (unknown malware) binaries using a machine learning algorithm. Figure 1 shows the architecture of our malware detection system. The system consists of three main modules: (1) F-Miner, (2) feature selection or reduction, and (3) classifier.

- Module 1: It parses AndroidManifest.xml and classes.dex files of Android applications in APK format to find out and extract all Permission and API calls data, and store them in the database.
- Module 2: All raw features are ranked and selected by the feature selection algorithm, and a feature space is created in a CSV file.
- Module 3: According to the feature preprocessing, every program in the dataset was transformed into a corresponding feature vector and then used a learning algorithm to derive a classification result from these labeled feature vectors.

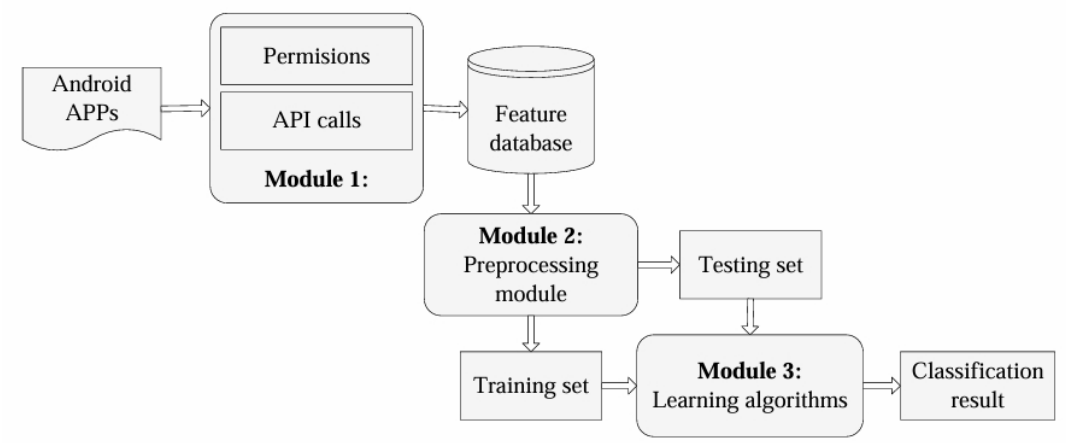

Figure 1: Architecture of the Malware Detection System.

4.1.1. Dataset description

To conduct the experimental analysis, we used a dataset comprising 17,752 Android APK files, including 5,801 malicious and 11,951 clean programs. In the following, we describe the dataset we used for our experiments. In this work, we considered only unpacked and unobfuscated malware samples.

All the 5,801 Android malware samples used in this project were collected from two sources. The first is the most recent publicly available Drebin dataset [8], which includes 5,560 malware samples from 179 families. Since the Drebin malware samples had been collected from August 2010 to October 2012, we added recently discovered malicious applications retrieved from the VirusTotal online scanning service [24]. This second dataset comprises 241 new releases, including Ransomware, etc., and all the samples have been collected if a sample is flagged as malware by at least three engines of VirusTotal. The benign 11,951 Android applications downloaded from Google Play and alternative Android markets. These benign

applications were used to obtain samples belonging to all the distinct categories available on the market, as we wanted to have heterogeneous applications in the dataset.

To prepare reliable data, we implemented an application named APK2Database.py based on the VirusTotal Public API in Python. The VirusTotal is a free online service that uses over 60 different antivirus engines to scan the sample [24]. This analysis shows the total number of engines that have been detected as malicious and malware for that engine. Since the experiment desired to have all the possible clean malware samples, we decided that if a sample were detected as malicious by at least one of the antivirus engines, it would be immediately separated from the rest of the dataset. Furthermore, the ones detected as malware by at least three engines are combined into a malware dataset immediately. Finally, the program stores all the information about the Android applications in the apk_list table of the database. The database has three tables, apk_list, apk_perms, and apk_apis, as shown in Figure 2.

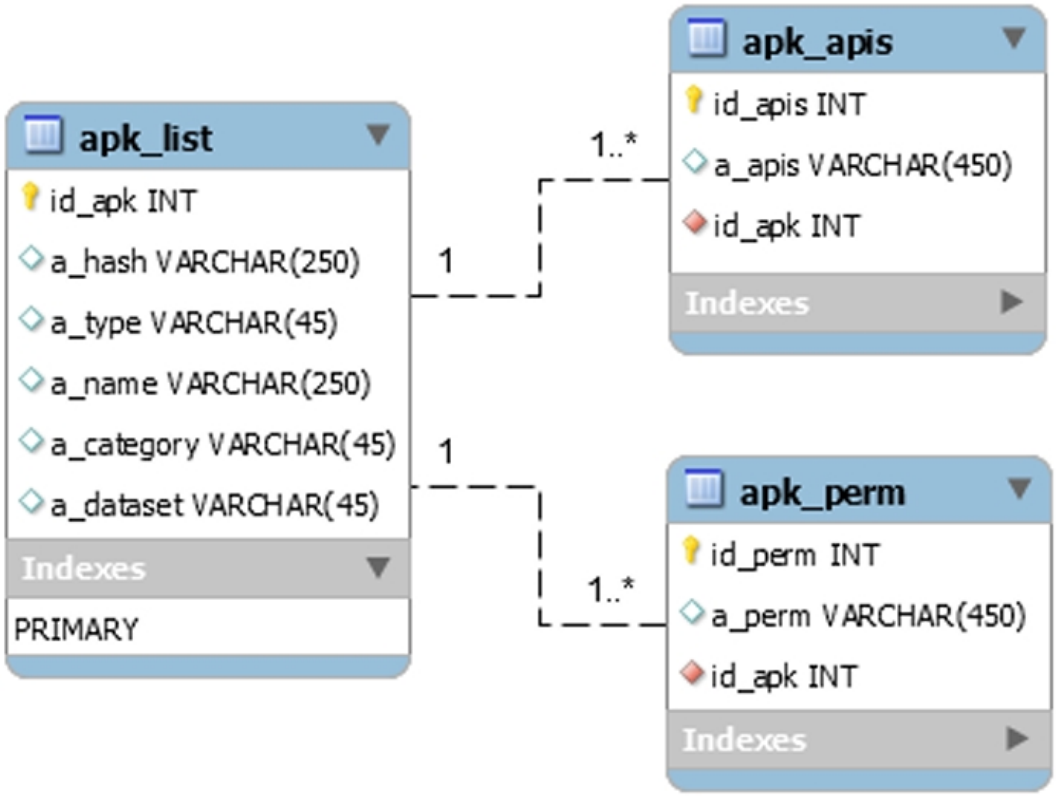

Figure 2: Database relationship diagram.

All the information of an application is stored in apk_list table that includes name, hash, type, category, and dataset. For example, a_type column of apk_list table contains information about whether the application is benign or malware. a_category column of the table includes an application's related family information, such as, in the case of malware, whether it is Backdoor or Trojan, or, in the case of benign, whether it is Travel or Finance application, etc. As shown in Figure 4, the apk_perms and apk_apis tables of the database contain all the corresponding raw features of Android applications. Figure 3 shows a sample of apk_list table.

id_apk	a_hash	a_type	a_name	a_category	a_dataset
1	090b5be26bcc4df6186124c2b47831eb96761fcf61282d63e13fa235a20c7539	malware	Plankton	NULL	Drebin
2	bedf51a5732d94c173bcd8ed918333954f5a78307c2a2f064b97b43278330f54	malware	DroidKungFu	NULL	Drebin
3	149bde78b32be3c4c25379dd6c3310ce08eaf58804067a9870cfe7b4f51e62fe	malware	Plankton	NULL	Drebin
4	dd11c105ec8bb3c851f5955fa53eebb91b7dc46bef4d919ee4b099e825c56325	malware	GinMaster	NULL	Drebin
5	6832234c4eae7a57be4f68271b7eecb056c4cd8352c67d2273d676208118871d	malware	FakeDoc	NULL	Drebin
6	f39f20ec060481bd89cfbd44654077fcd6404d87a1286685570334bd430e2f18	malware	GinMaster	NULL	Drebin
7	eb1bcca87ab55bd0fe0cf1ec27753fddcd35b6030633da559eee42977279b8db	malware	FakeInstaller	NULL	Drebin
8	5010f34461e309ea1bc5539bb24fccc320576ce6d677a29604f5568c0a5e6315	malware	Opfake	NULL	Drebin
9	f1c8b34879b04dc94f0a13d33c1e1272bdf9141e56e19da62c1a1b27af128604	malware	FakeInstaller	NULL	Drebin
10	4e355df8f0843afc4a7bfc294ee4b1db9e9b896269c754c2d57dcb647dcd3efb	malware	Opfake	NULL	Drebin
11	ecf9c8520e13054bcc1b1a18cc335810f7eb97bdbe75fc204ad050228f805216	malware	BaseBridge	NULL	Drebin
12	255eae7859b0855b15de30e5405a2714837ac556c238bc009ac74c5bfa69714a	malware	Nisev	NULL	Drebin
13	54f2a636e000c55bb725d7e552a22117837c1676fb4b96decd135ae10e6f7049	malware	BaseBridge	NULL	Drebin
14	73b2fd2dfb5860f0701b16002832987d5edcfe059df454d30548aa4fa45bebe6	malware	Opfake	NULL	Drebin
15	1035296bb576144475b684a9f703fc0220e8c0107a9e7e87c4bf76f72496b6c4	malware	FakeInstaller	NULL	Drebin
16	a022c24d8295dee70c0f090e4328063971170c33f7bf313b4debbc51d1a2b127	malware	Adrd	NULL	Drebin

Figure 3: apk$_{l}isttable$.

4.1.2. Feature extraction and selection

In this subsection, we detail all our methods of extracting features. We use the static analysis method to extract valuable raw features of Android applications. Android applications are packaged in APK file format, which is a ZIP file; therefore, we need to parse their content to get the necessary features.

This work uses two kinds of features: permissions and API calls. Because permission represents the macro behavior of an application, API calls contain rich semantic information and can expose the application's behaviors or tell you a lot about its functionality. To extract these two kinds of features from an APK file, we implemented an application based on the Python API of Androguard [25]. The program has two separate modules, one for extracting the permission feature and the other for extracting the API call feature. The permission feature module of the program parses the Android Manifest. An XML file containing permissions extracts all the permission data and stores it in the database's apk_perms table. The API permission module decompiles classes.dex file containing all API calls extracts all the API call data and stores them in the apk_apis table of the database.

Figure 4 shows a sample of the apk_perms table with three columns: id_perm, id_apk, and a_perms. The column id_perm provides the number of the permissions stored in the table. The column a_perms stores the permissions extracted from the applications. The features of an application can be differentiated by the id_apk column, which is the unique ID of an application. The structure of the apk_apis table is the same as the apk_perms table; the only difference is the a_apis column, which stores API call features of the applications.

id_perms	id_apk	a_perms
1	5695	android.permission.GET_TASKS
2	5695	android.permission.WRITE_EXTERNAL_STORAGE
3	5696	android.permission.ACCESS_NETWORK_STATE
4	5696	android.permission.ACCESS_WIFI_STATE
5	5696	android.permission.ADD_SYSTEM_SERVICE
6	5696	android.permission.BROADCAST_STICKY
7	5696	android.permission.CALL_PHONE
8	5696	android.permission.CHANGE_WIFI_STATE
9	5696	android.permission.INTERNET
10	5696	android.permission.MOUNT_UNMOUNT_FILESYSTEMS
11	5696	android.permission.READ_CALL_LOG
12	5696	android.permission.READ_CONTACTS
13	5696	android.permission.READ_PHONE_STATE

Figure 4: An example of the apk$_{p}ermstable$.

There are 151 permissions available in the current Android platform that are used in the permission feature space. According to all these permission features, we could create a binary feature vector for a program to represent the permission feature set of the program, which will finally create our permission feature space S_1. Each feature vector element related to a permission feature is denoted as f_i, and its binary value indicates whether a program has the corresponding feature. The permission feature vector Pi for a program is defined as follows: $P_i = f_1, f_2, f_3, \ldots, f_n$. Figure 8 shows a sample permission feature space S_1 in permission_set.csv file with corresponding values of the Android programs.

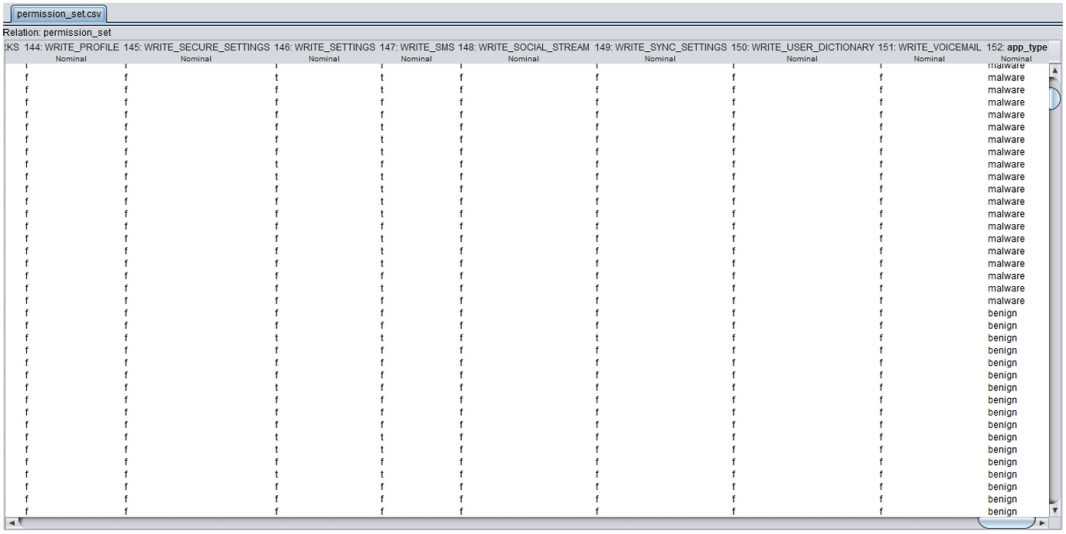

Figure 5: A Sample csv file of the Permission features.

We have analyzed how many permissions, on average, an application requests based on collected information in the database. According to our analysis, a malicious application usually requests more permission than a benign application, where malware requires more than 12 permissions on average. In comparison, a benign application requires six permissions on average. Besides the number of requested permissions, the frequently requested permissions are also considered in this study.

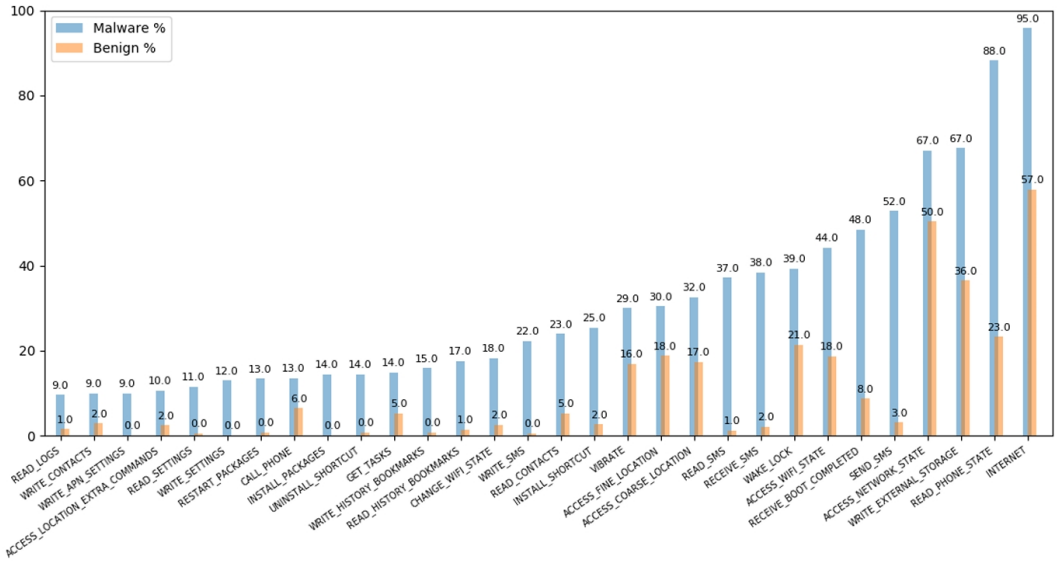

Figure 6: Top 30 Requested Permissions by Malware and comparison with Benign

As shown in Figure 6, the INTERNET is in a category of normal permission, and it is the top requested permission by both Malware (95%) and benign (57%) applications that allow the applications to connect to the network. Android applications requesting permissions in the category of dangerous want data or resources that involve the user's private information or could potentially affect the user's stored data or the operation of other applications. For example, READ_PHONE_STATE allows the application to get your phone number, current cellular network information, the status of ongoing calls, and so on.

CALL_PHONE permits the application to call anywhere, including paid numbers, at your charge. SMS related permissions, such as SEND_SMS and READ_SMS, let the application receive and read your incoming SMS messages and send them, which is charged to you. Alternatively, WRITE_SMS permits the application to write SMS messages stored on your phone or SIM card. Also, the application may delete your messages. ACCESS_COARSE_LOCATION permission allows the application to access your location. It is based on data from cellular base stations and Wi-Fi hotspots. At the same time, ACCESS_FINE_LOCATION lets the application access your exact location based on GPS data. If an application can read the list of contacts and edit and alter them, it gains the privilege through READ_CONTACTS and WRITE_CONTACTS permissions.

The latest Android API (API 26, Android O) has 287 packages, including thousands of calls and methods. According to our analysis, the most unique frequently called 675 API call features are selected in this project. Also, we use a binary feature vector for a program to represent the API call feature set, which will create our API call feature space S2. We analyzed the average number of API calls used by malware and benign applications. The results show that benign applications use more API calls, which is 1940, than malware applications, which are 678. The most frequent API calls in Malware applications are presented in Figure 7.

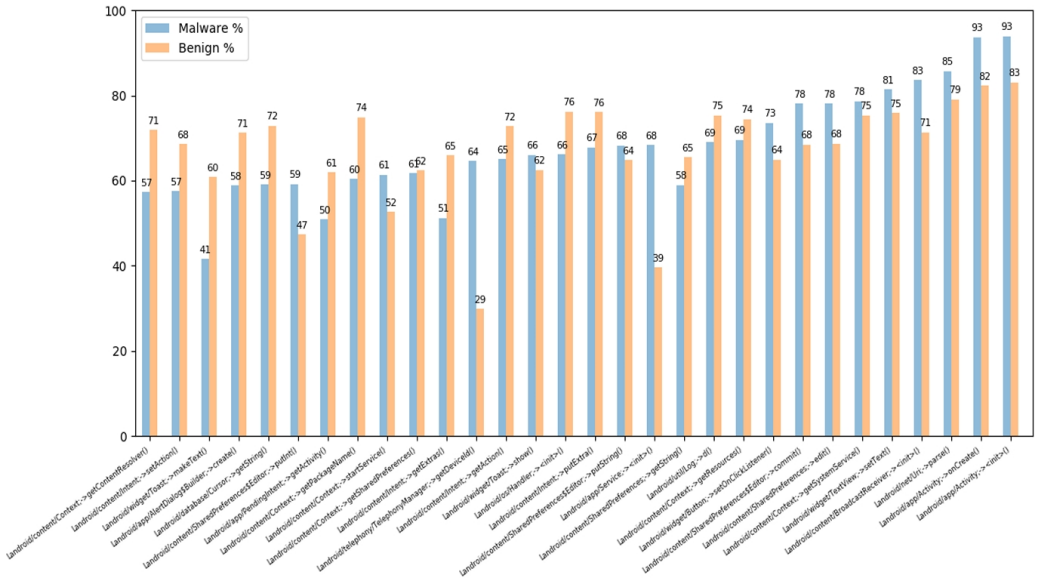

Figure 7: Top 30 Requested API calls by Malware and comparison with Benign.

As shown in Figure 7, Activity is the top requested call by both malware and benign applications. An activity can be implemented using the Activity class, and subclasses have such a critical interaction between the system and an application that they dictate the user interface and handle the user interaction with the smartphone screen. Uri is one of the most frequently called APIs, which refers to an immutable uniform resource identifier. A broadcast receiver allows the user to register for system or application events that the BroadcastReceiver implements. TextView is a user interface that displays text to the user and optionally allows them to edit it.

We have had two independent feature spaces related to two different feature sets. We combined each feature set into an ensemble feature space (S1 and S2). After that, we analyze our proposal performance using these three sets and several evaluation metrics. According to [11], selecting a subset of features to use in the learning scheme must be preprocessed because some of them, perhaps the overwhelming majority, are irrelevant or redundant. The performance of learning algorithms can frequently be improved by preprocessing, even though the algorithms try to select attributes appropriately and ignore irrelevant or redundant ones. Therefore, we have used several feature ranking methods: Information Gain, Bi-Normal Separation, Chi-squared, and Relief. We also use Principal Component Analysis to select feature subsets. Using these ranking and selection methods, we conduct experiments to find the best feature set that will improve the classification result of the learning algorithm.

5. Discussion

5.1. Experimental results

We used a dataset of 17,752 Android applications or APKs to conduct the experimental analysis, including 5,801 malicious and 11,951 clean programs. The experiments are performed on a computer with Windows 10, Intel Core i7 CPU @ 3.50 GHz, and 16GB RAM. We have selected Information Gain, Bi-Normal Separation, Chi-squared, and Relief feature ranking algorithms, and Random Forest, J48, Support Vector Machine, and Naïve Bayes classifiers to evaluate the performance of our proposal. All these feature selection algorithms and classifiers are available as a part of the Java-based open-source machine learning toolkit Weka [11]. Our experiments used the standard 10-fold cross-validation process [12]. The dataset is randomly divided into 10 smaller subsets, where 9 subsets are used for training and 1 subset for testing. The process is repeated 10 times for every combination. To evaluate our system, we were interested in several quantities listed below:

- True Positive Rate (TPR) the percentage of malware that is correctly identified as malware. It is also called Detection Rate (DR):

$$TP = \frac{TP}{TP + FN}$$

- False Positive Rate (FPR): the number of benign programs incorrectly classified as malicious executable:

$$FPR = \frac{FP}{TN + FP}$$

- Precision: the number of correctly classified programs within all classified programs:

$$TP = \frac{TP}{TP + FP}$$

- Overall Accuracy (OA): the number of correctly identified applications. In other words, it shows the ratio of correctly classified programs:

$$TP = \frac{TP + TN}{TP + TN + FP + FN}$$

We are interested in the DR of the classifiers, which in our case was the percentage of the total malicious programs labeled malicious. We are also interested in the FPR, the percentage of benign programs labeled as malicious, also called false alarms. OA evaluates the detection efficiency of the system, which considers not only the higher TPR but also the lower FPR. Therefore, the OA could be the main reference for us to compromise TPR and FPR.

5.1.1. Results

To obtain baseline results for detecting Android malware, experiments were conducted to select the best feature subset and model with each of the individual and ensemble features, including both permission and API feature sets. The first set of experiments was performed on the top 30, 70, and 100 of permission subset features selected by four feature ranking algorithms, namely Information Gain (IG), Bi-Normal Separation (BNS), Chi-Squared (Chi) with Relief (Rf) and four frequently used classification algorithms in this field, such as Naïve Bayes (NB), Support Vector Machine (SVM), J48, and Random Forest (RF), then the results are shown together in Table 1 and 2. It also includes the result of the classification algorithms with "ALL"(150) features and 100 subset features selected by the Principal Component Analysis (PCA) feature reduction and selection algorithm.

Table 2 shows that the highest overall accuracy is for the random forest (95.7%) classifier with relief (70 and 100), which outperforms all the other classifiers. This algorithm also gives the best results in all the other criteria, which are Detection Rate (97.6%), False Positive Rate (7.4%), and Precision (95.8%). Support Vector Machine (75.7%) gives the worst Overall Accuracy with Bi-Normal Separation (30). Random Forest (97.7%) shows the highest True Positive Rate with Principal Component Analysis (100). Here, the lowest False Positive Rate is given by Random Forest (7.3%) with Relief (100). The highest precision is J48 (96.4%) with information gain (30).

Table 1: Classification results of NB and SVM with the permission features.

Feature Selection Algorithm	**Number of features**	**Naïve Bayes**				**Support Vector Machine**			
		OA	TPR	FPR	Precision	OA	TPR	FPR	Precision
IG	30	86.0	92.8	25.5	86.2	90.3	94.9	17.5	90.3
BNS	30	76.3	94.2	54.3	74.8	75.7	94.9	39.5	75.7
Chi	30	85.7	92.5	25.9	86.0	90.0	95.5	19.2	90.1
Rf	30	84.4	91.9	25.7	96.0	91.2	94.7	14.7	91.7
IG	70	87.2	93.7	23.7	87.2	91.6	95.5	13.6	91.5
BNS	70	80.9	91.1	44.5	77.8	79.7	86.2	32.4	82.1
Chi	70	87.2	93.7	23.7	87.1	91.6	95.5	16.7	91.6
Rf	70	86.9	93.3	23.9	87.2	92.1	95.5	13.5	92.1
IG	100	87.3	93.8	23.7	87.2	92.1	95.5	13.2	92.0
BNS	100	81.1	92.5	38.3	80.5	83.7	86.2	27.6	87.6
Chi	100	87.2	93.7	23.7	87.1	93.5	95.7	13.1	92.7
Rf	100	87.2	93.7	23.7	87.1	92.1	95.5	13.5	92.1
PCA	100	80.9	81.3	19.7	87.6	91.3	96.1	16.8	90.7
ALL	150	82.2	88.7	23.7	82.1	87.2	90.6	13.4	87.4

However, the highest precision (96.4%) was given by the J48 classifier with IG (30). This is not a good result to consider in this work because of the other criteria, such as Overall Accuracy (84.4%), True Positive Rate (91.9%), and False Positive Rate (25.7%). To mention here, it is a considerable result in all criteria that Overall Accuracy (95.1%), True Positive Rate (97.2%), False Positive Rate (8.4%), and Precision (95.2), shown by Relief (30) with Random Forest classifier, because of the reason that the smaller number of features could result in less processing overhead. Finally, we can conclude from the first set of experimental results that the Relief (70) feature subset training Random Forest classifier is given the best model in all the evaluation metrics.

The next set of experiments was performed on the top 250, 450, and 600 API call subset features selected by the four ranking algorithms and one feature reduction algorithm with the four classifiers. Then, the results are shown together in Tables 3 and 4. The tables also include the result of the classification algorithms with "ALL"(675) features and 239 subset features selected by Principal Component Analysis (PCA). We can see in Table 3 that the worst result is shown in all evaluation criteria by Naïve Bayes (52.4%) classifier with Principal

Table 2: Classification results of J48 and RF with the permission features.

Feature Selection Algorithm	Number of features	J48				RandomForest			
		OA	TPR	FPR	Precision	OA	TPR	FPR	Precision
IG	30	92.7	96.4	13.3	96.4	93.5	96.9	12.1	93.2
BNS	30	76.9	95.5	54.8	74.9	77	95.2	83.9	75.1
Chi	30	92.7	96.4	13.4	92.5	93.6	96.9	12.1	93.2
Rf	30	85.4	91.9	25.7	86.0	95.1	97.2	8.4	95.2
IG	70	94.0	96.3	9.3	94.7	95.0	97.1	8.6	95.1
BNS	70	83.7	87.3	22.3	87.0	84.3	87.1	21.4	87.5
Chi	70	94.0	96.3	9.3	94.7	94.9	97.1	8.7	95.1
Rf	70	86.9	93.3	23.9	87.0	**95.7**	**97.6**	**7.4**	**95.8**
IG	100	94.1	96.1	9.0	94.8	95.5	97.4	7.6	95.6
BNS	100	86.1	90.2	20.7	88.2	86.7	90.0	18.9	89.1
Chi	100	94.2	96.1	9.0	94.8	95.6	97.5	7.6	95.6
Rf	100	94.3	96.2	8.9	94.8	95.7	97.6	7.3	95.8
PCA	100	94.2	96.0	8.9	94.9	95.5	97.7	8.2	95.3
ALL	150	89.4	91.4	8.4	89.9	90.8	92.6	7.8	90.8

Component Analysis (239). The lowest False positive rate, 7.5%, is shown by Naïve Bayes with Bi-Normal Separation (600).

As shown in Table 4, we can see that the result of the Random Forest classifier outperforms the other classifiers in all feature selection methods. This algorithm gives the best results in all Overall Accuracy (95.9%), Detection Rate (98.9%), False Positive Rate (8.9%), and Precision (95%) with Chi-squared (600). The highest Precision is 95%, given by several classifiers with different numbers of subset features. However, here the lowest False Positive Rate is 7.5% by Naïve Bayes, it does not give good results in the other criteria that are Overall Accuracy 80%, True Positive Rate 72.8%, and Precision 94.4%.

Table 3: Classification results of NB and SVM with the API features.

Feature Selection Algorithm	Number of features	Naïve Bayes				Support Vector Machine			
		OA	TPR	FPR	Precision	OA	TPR	FPR	Precision
IG	250	79.3	72.2	8.4	93.6	93.4	96.9	12.5	93.0
BNS	250	77.6	70.6	10.3	92.2	83.9	97.4	39.2	81.0
Chi	250	79.3	72.3	8.5	93.6	93.6	96.9	12.2	93.2
Rf	250	79.4	72.2	8.2	93.8	93.5	96.9	12.2	93.2
IG	450	79.3	72.2	8.5	93.6	93.5	96.9	12.2	93.1
BNS	450	79.0	72.2	8.9	93.3	91.5	96.7	17.6	90.4
Chi	450	79.3	72.2	8.5	93.6	93.4	96.9	12.4	93.1
Rf	450	79.4	72.2	8.2	93.8	93.5	96.9	12.2	93.2
IG	600	79.4	72.2	8.2	93.8	93.5	96.9	12.2	93.2
BNS	600	80.0	72.8	7.5	94.4	93.7	97.2	12.4	93.1
Chi	600	79.4	72.2	8.2	93.6	93.6	96.9	12.2	93.2
Rf	600	79.4	72.2	8.2	93.8	93.5	96.9	12.2	93.2
PCA	239	**52.4**	30.0	9.0	85.1	91.4	97.7	11.7	90.3
ALL	675	79.4	72.2	8.2	93.8	93.5	96.9	12.2	93.2

Finally, we can conclude that the best API call feature subset is Principal Component Analysis (239) with the RandomForest classifier. The last set of experiments was performed with top 450, 550, 650, "ALL" (825), and PCA (226), with different numbers of ensemble features selected by the algorithms as shown in Table 5.

We can see in Table 5 that the highest overall accuracy is 96%, as shown by RandomForest with information gain (450), Chi-squared (550), and Releaf (650). The highest True Positive Rate is 99.1%, given by RandomForest with Information Gain (600), while the lowest False Positive Rate is 5.8%, given by Naïve Bayes with Information Gain (550). RandomForest gives the highest Precision (95.1%) with Information Gain (450), Releaf (450), and Chi-squared (550). We can conclude that the best ensemble feature subset is Information Gain

Table 4: Classification results of J48 and RF with the API features.

Feature Selection Algorithm	Number of features	J48				RandomForest			
		OA	TPR	FPR	Precision	OA	TPR	FPR	Precision
IG	250	94.1	96.8	10.3	94.2	95.8	98.6	8.0	95.0
BNS	250	90.1	94.9	18.0	90.0	91.5	95.6	16.9	90.7
Chi	250	94.1	96.8	10.2	94.2	95.8	98.7	8.8	95.0
Rf	250	94.2	96.8	10.2	94.2	95.8	98.6	8.0	95.0
IG	450	94.2	96.9	9.2	94.2	95.9	98.9	8.2	95.2
BNS	450	93.4	96.5	11.9	93.1	95.1	98.1	10.1	94.4
Chi	450	94.2	96.8	10.2	94.2	95.8	98.6	8.2	95.0
Rf	450	94.2	96.8	10.2	94.2	95.8	98.8	8.0	94.9
IG	600	94.2	96.9	9.0	94.2	95.8	98.9	9.0	95.0
BNS	600	94.0	96.6	10.1	94.1	95.8	98.6	9.1	94.9
Chi	600	94.2	96.8	10.2	94.2	**95.9**	**98.9**	**8.9**	**95.0**
Rf	600	94.2	96.8	10.2	94.2	95.8	98.8	9	95
PCA	239	94.2	96.8	10.3	94.2	95.5	97.8	8.9	94.9
ALL	675	93.4	96	11	93.8	90.1	90.3	10.3	90.3

Table 5: Classification results with the different numbers of ensemble features

		NB				SVM				J48				RF			
Feature Selection		OA	TPR	FPR	Precision	OA	TPR	FPR	Precision	OA	TPR	FPR	Precision	OA	TPR	FPR	Precision
IG	450	71.5	59.0	6.8	93.7	90.4	96.7	20.3	89.1	94.4	97.2	10.3	94.2	96.6	98.9	3.8	95.1
BNS		70.2	56.3	5.9	94.2	92.1	98.0	18.0	90.4	93.9	97.1	11.5	93.6	95.4	98.7	1.0	94.4
Chi		71.5	59.0	6.8	93.7	93.2	98.0	14.7	91.9	94.4	97.2	10.3	94.2	95.9	98.8	8.9	95.0
Rf		71.6	59.0	6.8	93.7	93.3	98.0	13.2	92.8	94.4	97.1	10.1	94.3	95.8	98.3	8.8	95.1
IG	550	71.5	59.0	5.8	93.7	90.3	96.7	20.6	89.0	94.4	97.2	10.3	94.2	96.8	98.5	8.8	95.1
BNS		71.1	58.1	6.4	94.0	94.2	97.9	16.8	90.0	94.2	97.0	10.4	94.1	95.8	98.3	8.8	94.8
Chi		71.5	59.0	6.8	93.7	93.2	98.0	14.9	91.9	94.4	97.2	10.3	94.2	95.9	98.8	9.0	95.0
Rf		71.6	59.0	6.8	93.7	93.7	98.1	13.1	91.8	94.4	97.1	10.1	94.3	95.8	98.3	8.8	95.1
IG	650	71.5	58.7	6.5	93.7	90.2	96.7	20.6	88.9	94.3	97.2	9.9	94.4	96.4	98.5	8.8	95.0
BNS		71.5	58.7	6.5	93.7	94.2	97.9	15.7	91.0	94.4	97.2	9.9	94.4	96.4	98.5	8.8	95.0
Chi		71.5	59.0	6.8	93.7	92.9	97.9	15.6	91.5	94.4	97.1	10.3	94.3	95.8	98.3	8.8	95.1
Rf		71.6	59.0	6.8	93.7	92.9	98.0	15.7	91.5	94.4	97.1	10.3	94.3	95.8	98.3	8.8	95.1
PCA	226	50.7	28.6	11.2	81.4	85.5	97.2	34.4	82.9	94.4	97.1	10.3	94.3	96.4	98.5	8.8	95.1
ALL	825	71.5	59.0	6.8	93.7	90.3	96.7	20.6	88.9	92.5	92.2	10.0	93.7	94.9	97.3	9.3	95.0

(450) with the RandomForest classifier.

6. Conclusions

This work presents an Android malware detection system using data mining and machine learning techniques. This system is based on mining permission information, and API function calls inside Android applications for detecting malware and malicious codes. Therefore, we implemented a program that has two separate modules. One is for extracting the permission feature, and the other one is for extracting the API call feature, and this program stores all the extracted features in the database.

To obtain baseline results for detecting android malware, experiments were conducted to select the best feature subset and model with each individual and ensemble feature space, including both permission and API feature sets. According to the experimental results of our analysis, the Random Forest classifier outperforms the rest of the classifiers in most cases.

According to the results, the top 70 permission feature subset ranked by Relief is selected as the best permission feature subset because the model of Random Forest classifier with this feature subset results in Overall Accuracy (95.7%), Detection Rate (97.6%), False Positive Rate (7.4%), and Precision (95.8%). The model of Random Forest with API call features in total 239 selected by Principal Component Analysis gives the highest result in Overall Accuracy (95.9%), Detection Rate (98.8), False Positive Rate (9%), and Precision (94.9%). After that, the model of the Random Forest classifier with 450 ensemble features selected by Information Gain (450) shows the highest result in Overall Accuracy (96%), True Positive Rate (98.9%), False Positive Rate (8.8%), and Precision (95.1%).

Even though the Random Forest classifier model trained with 450 ensemble features gives the highest overall accuracy (96%), it is slightly better than the other models. This model provides a 0.3% better Overall Accuracy than the model with the 70 permission features and

a 0.1% better result than the model with the 239 API call feature set. On the other hand, if we look at the false positive rate of the models, the lowest was given by the model of Random Forest with 70 permission features. The highest Precision is also shown by the model with the permission feature subset.

These results show that our proposed android malware detection system using the permission feature subset can effectively detect malicious applications. Also, this malware detection system based on the permission features could be effective against any obfuscated malware since the permissions of an android application are strictly defined in android Manifest.xml file of the application.

In future work, we plan to add a preprocessing module that detects whether a malware sample is obfuscated or packed, and prepares it for input to Module 1. We will then conduct experiments to evaluate the system's effectiveness against obfuscated and packed malware. Additionally, we aim to reduce the false positive rate and improve the overall accuracy of the system based on permission features.

References

[1] K. A. Nelson, R. J. Davis, D. R. Lutz, and W. Smith, "Optical generation of tunable ultrasonic waves," *Journal of Applied Physics*, Vol. 53, no. 2, Feb., pp. 1144-1149, 2002, doi: 10.1063/1.329864. https://doi.org/10.1063/1.329864

[2] "Smartphone OS Market Share" 2023 Q4, Accessed Jan., 2024, http://www.idc.com/promo/smartphone-market-share/os..

[3] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution,"in IEEE Symposium on Security and Privacy, S.P. 2012, 21-23 May 2012, San Francisco, California, USA, pp. 95-109.

[4] I.T. threat evolution Q3 2023, accessed Jan. 2024, https://securelist.com/it-threat-evolution-q3-2023-mobile-statistics/111224/.

[5] D. Maiorca, F. Mercaldo, G. Giacinto, A. Visaggio, and F. Martinelli, "R-PackDroid: API Package-Based Characterization and Detection of Mobile Ransomware, in *ACM Symposium on Applied Computing (SAC)*, pp. 1718-1723, 2017.

[6] M. Sikorski and A. Honig, *Practical Malware Analysis*, No Starch Press, 2012. https://doi.org/10.1016/S1353-4858(12)70109-5

[7] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware In The Wild," in *Proc. of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[8] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided Android malware classification," *Computers and Electrical Engineering*, vol. 61, pp. 266-274, 2017. https://doi.org/10.1016/j.compeleceng.2017.02.013

[9] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck, "Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket," in *21st Annual Network and Distributed System Security Symposium (NDSS)*, February 2014.

[10] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Seventh Asia Joint Conference on Information Security (Asia JCIS)*, pp. 62-69, IEEE, 2012.

[11] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," Springer, pp. 86-103, 2013.

[12] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed., Morgan Kaufmann, 2005.

[13] J. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, USA, 1993.

[14] W. J. Li, S. J. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis, "A Study of Malcode-Bearing Documents," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 231-250, Springer, Switzerland, 2007.

[15] A. Aswini and P. Vinod, "Droid permission miner: Mining prominent permissions for Android malware analysis," in *Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*, IEEE, 2014.

[16] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification,"in *Proc. of the 16th ACM Conference on Computer and Communications Security*, pp. 235-245, ACM, 2009.

[17] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. G. Bringas, "On the automatic categorisation of android applications," in *Consumer Communications and Networking Conference (CCNC)*, pp. 149-153, IEEE, 2012.

[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified,"in *Proc. of the 18th ACM Conf. on Computer and Communications Security*, pp. 627-638, ACM, 2011.

[19] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior based malware detection system for android,"in *Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 15-26, ACM, 2011.

[20] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Detecting Privacy Leaks in Android Applications,"Tech. rep., UC Davis, 2011.

[21] S. H. Seo, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, vol. 38, pp. 43-53, 2014. https://doi.org/10.1016/j.jnca.2013.05.008

[22] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls,"in *Proc. of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI'13)*, pp. 300-305, IEEE, 2013.

[23] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications,"in *Computer Security-ESORICS*, pp. 163-182, Springer, 2014.

[24] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs,"in *Proc. of ACM CCS*, 2014.

[25] Free Online Virus, Malware and URL Scanner: https://www.virustotal.com.

[26] A. Desnos and G. Gueguen, "Android: From reversing to decompilation,"in *Proc. of Black Hat*, Abu Dhabi, 2011.