Feature Engineering and Evaluation for Android Malware Detection Scheme

Jaemin Jung[1], Jihyeon Park[2], Seong-je Cho[2], Sangchul Han[3], Minkyu Park[3], Hsin-Hung Cho[4]

[1] Departmetnt of Computer Science and Engineering, Dankook University, Korea
[2] Department of Software Science, Dankook University, Korea
[3] Department of Software Technology, Konkuk University, Korea
[4] Department of Computer Science and Information Engineering, National Ilan University, Taiwan
{snorlax, jihyeon, sjcho}@dankook.ac.kr, {schan, minkyup}@kku.ac.kr, hhcho@niu.edu.tw

Abstract

Android is one of the most popular platforms for the mobile and Internet of Things (IoT) devices. This popularity has made Android-based devices a valuable target of malicious apps. Thus, it is essential to devise automatic and portable malware detection approaches for the Android platform. There are many studies on detecting mobile malware using machine learning techniques. In these studies, however, the dataset is imbalanced or is not large enough to generalize the machine learning model, or the dimensionality of features is too high to apply nonlinear classifiers. In this article, we propose a machine learning-based Android malware detection scheme that uses API calls and permissions as features. To restrict the dimensionality of features, we propose minimal domain knowledge-based and Gini importance-based feature selection. We construct large and balanced real-world datasets to build a generalized and non-skewed model and verify our model through experiments. We achieve 96.51% classification accuracy using Random Forest classifier with low overhead. In addition, we also provide an analysis on falsely classified samples in detail. The analysis results show that API hiding can degrade the performance of API call information-based malware detection systems.

Keywords: Android app, Malware detection, Feature engineering, False alarm

1 Introduction

Android is a platform for smart devices and lightweight Internet of Things (IoT) devices. Developers can build apps on top of popular platforms without previous knowledge of embedded systems [1]. Android has advantages over conventional platforms that have been employed in developing WSNs [2]. Since it forms a layer supported by well-designed components interacting with each other, one can build IoT systems easily on it. It can also be easily scaled by making a new functionality as a module. We can provide various services utilizing the connectivity of mobile and IoT devices on Android [3-6].

As Android-based platforms become more popular, Android devices including smartphones and IoT are becoming attractive targets for cyber criminals [6-10]. According to a recent report from cybersecurity company McAfee [11], more than 30 million malicious mobile apps were found in the fourth quarter of 2018, and more than 6 million new mobile malware instances have been introduced each year since 2016. Hence, many researchers contribute to mobile malware detection and prevention including Android malware [12-24].

There have been several different approaches to detecting mobile malware. Traditional malware detection approaches [16-17] compare suspicious apps with signatures. Signatures are known malware patterns based on the executable code. The demerit of these approaches is that they can detect only the malicious apps that have signatures currently known. They cannot detect newborn malware [18-20, 24]. Besides, these approaches require continuous updating of the predefined signature database. Christodorescu and Jha [16] concluded that "Signature-based approaches never keep up with the speed at which malware is created and evolved".

Instead of using malware signatures, other effective approaches [18-25] utilize machine learning or data mining techniques to detect not only *known* but also *unknown* malware instances. Machine learning classifiers can address some of the problems of signature-based malware detection by automatically reasoning about benign and malicious apps to fit detection model parameters [22-23]. Machine learning techniques take a labeled dataset and generate a model that can deal with data not included in the dataset. It is shown that employing machine learning classifiers can improve detection performance [20, 25]. When a machine learning technique is used for malware

*Corresponding Author: Seong-je Cho; E-mail: sjcho@dankook.ac.kr
DOI: 10.3966/160792642021032202017

detection and classification, there are several challenges: feature extraction and selection [19, 23-24, 26-30], collection of a comprehensive real-world dataset [13], choosing and optimizing a suitable learning algorithm [21-23], performance evaluation [20, 31], and identifying false alarm [25].

We propose a new machine learning technique to detect Android malware utilizing permissions and API calls. Among the above-mentioned challenges, we focus on feature extraction and selection, dataset collection and identifying false alarms. Feature extraction maps a large collection of input data onto a small set of features while preserving the relevant information [29-30]. Feature extraction may transform original features into an organized and more significant subset of information. Feature selection reduces the dimensionality of datasets, which is a general preprocessing method in high dimensional data analysis [24, 27, 30]. Through feature selection, we select the relevant feature that we expect to be useful for malware detection. The classification results can be improved by selecting the most relevant features from the extracted features. Feature extraction and selection methods can be applied separately or combined in one step. They significantly affect the performance in terms of efficiency, robustness, and accuracy.

In our scheme, we first extract the information on all API invocations and permission requests from sample apps. Next, we reduce the size of the feature set by using two feature selection methods: (1) a minimal domain knowledge-based method and (2) a Gini importance-based selection method. The minimal domain knowledge-based method simply chooses the API calls and permissions used in the existing well-known studies [19, 32-34] and the Gini importance-based method decreases the size of the feature set under consideration. We adopt the feature importance [35-36] of each feature derived from the Gini impurity of the resulting Random Forest (RF) trees.

Many existing studies used imbalanced and/or small datasets. However, imbalanced dataset may result in a skewed model and too small dataset may lead to poor generalization. In our study, we construct a large and balanced dataset to build a generalized and non-skewed model. We collect 27,041 benign apps and 26,276 malwares from a real-world dataset, `AndroZoo`.

We have carried out several experiments and evaluated the proposed Android malware detection scheme. It achieved up to 96.51% accuracy with Random Forest algorithm. We have also investigated the undetected or misclassified apps in detail and discovered that we might incorrectly classify apps that are transformed by code obfuscation tools or written with cross-platform development tools.

The main contribution of this work is summarized as follows:

- We reduce the dimensionality of datasets and decrease the curse of dimensionality using the combined feature selection technique without degrading the detection performance: the minimal domain knowledge-based plus the Gini importance-based. Using minimal domain knowledge is recent trends in the research on malware detection [38-39].
- We construct the balanced datasets using real-world datasets, `AndroZoo` [37] and `Drebin` [33], in our experiments. The well-known but older datasets such as `Drebin`, `AMD` [40] and `GooglePlay` (during 2014 – 2016) show some different characteristics compared with the latest AndroZoo dataset, especially in terms of the number of APIs invoked by apps (see Section 4).
- We disclose the causes of incorrect classification where a malicious app is undetected or a benign app is misclassified as malicious. To the best of our knowledge, a few studies have been conducted on identifying incorrect classification issued by a machine learning technique in malware detection.

This article is organized as follows. Section 2 explains background knowledge about API calls and permissions on the Android platform. Section 3 presents our machine learning-based malware detection technique. Section 4 explains our experimental results and analyzes the misclassified samples. In Section 5, we compare our work with the related works. Finally, we give the concluding remarks and present possible future work in Section 6.

2 Background

2.1 API (Application Programming Interface)

The Android platform provides Application Programming Interfaces (APIs) that applications can use to interact with the underlying Android system to do various things [19]. The framework API refers to the collection of various software that makes up the Android SDK such as a core set of packages and classes, a set of XML elements and attributes for declaring a manifest file, etc. Android apps contain many API calls and permissions. Each API call is composed of four types of information: *class name*, *method name*, *argument information*, and *return data type*.

API calls reflect the functionality and behavior of an app and have been widely used in studies for malware detection, especially using machine learning algorithms. Android apps use the official Android APIs and third-party APIs [41]. Third-party APIs are often only used in a few apps and utilizing those APIs as a feature for machine learning can lead to sparse data problems. Also, third-party APIs may have different names but the same functionality, and vice versa. Hence, we use only the official Android APIs in malware detection.

Salehi et al. [42-43] mentioned that API name alone might not represent its operations and both API calls

and their arguments could be an effective representative of the executable behavior. They adopted each API call name, its arguments, and return value to detect Microsoft Windows malware. In our work, we consider the following API call information: class name, method name, method's argument types, and method's return data type. The API calls with the same class and method name are counted as different API calls if they have different arguments or return data type. The total number of API calls belonging to Android 7.1 (API level 25) is 133,271 [44]. Figure 1 shows a bytecode-level API call that consists of a class name (including a package name), a method name, and a method descriptor. The method descriptor consists of the types of arguments and return value [45].

Figure 1. An exapmle of bytecode-level API call representation

2.2 Permissions

Android apps require some permissions to perform specific functions [19, 46-47]. Android permissions enable the system or user to protect sensitive data or system features from apps. Permission requests reflect the app's behavior. An app must declare its permissions in its manifest file to access protected resources and interact with other apps. For example, if an app wants to read an address book on the device, it should declare the `READ_CONTACTS` permission in the `AndroidManifest.xml`. We collected lists of permissions from an Android application analysis tool AndroGuard [48]. The total number of Android permissions collected is 474.

The permissions declared in a manifest file are useful in catching the potential risks of apps [19, 32, 47]. The system's behavior depends on how sensitive the permission is. There are three protection levels in the Android permission system: *normal*, *signature*, and *dangerous*. Permissions for resources and data involving the user's private information or affecting the action of other apps fall on dangerous permissions [19, 32]. For example, `ACCESS_FINE_LOCATION` (to read the location of the user) and `READ_CONTACTS` (to read the user's contacts) are classified as dangerous. For dangerous permissions, apps should obtain the permission grant from the user at runtime.

3 The Proposed Method

Our malware detection technique consists of three steps: feature extraction, feature selection, and machine learning. Figure 2 shows a schematic diagram of the proposed technique. We explain each step in detail in the following subsections.

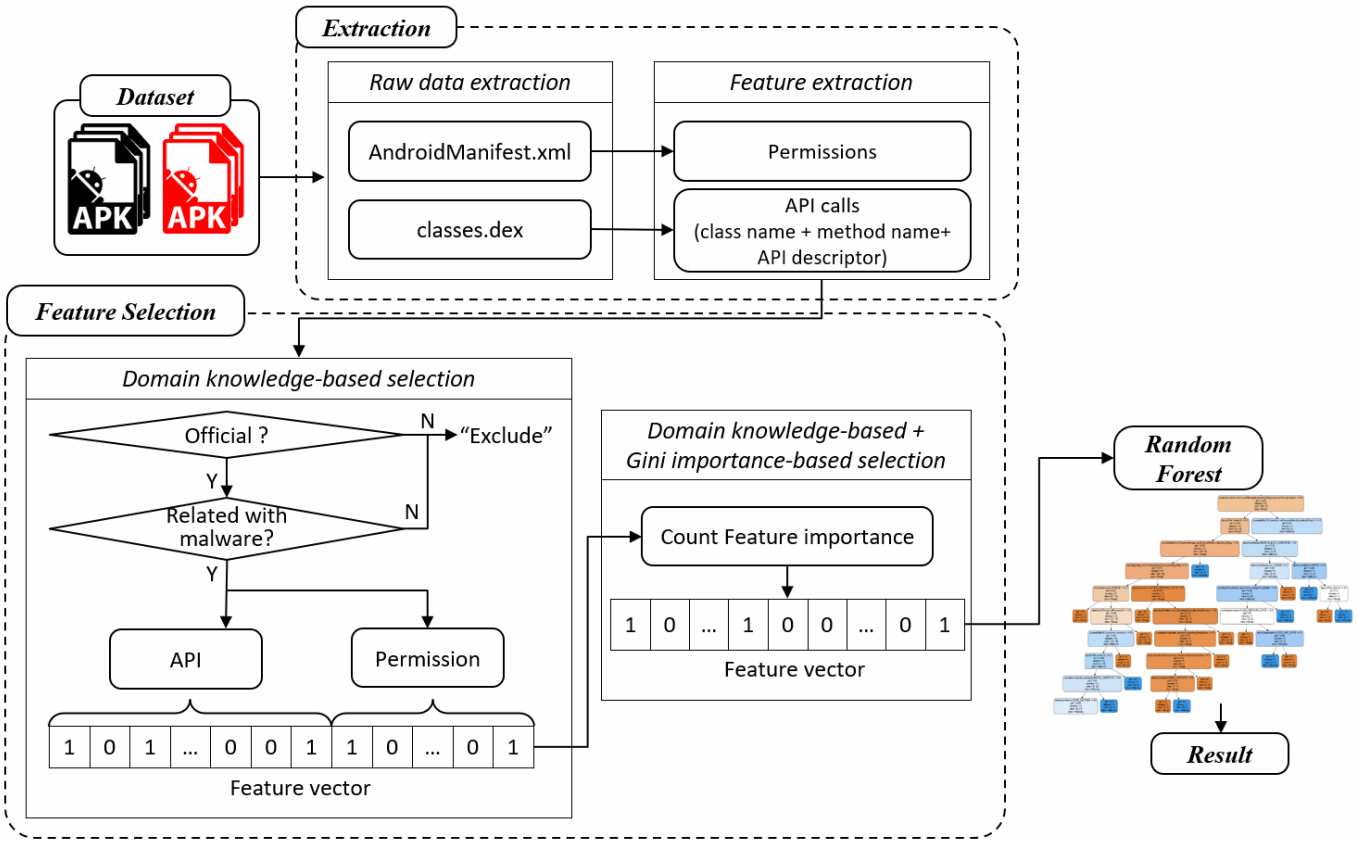

Figure 2. The schematic diagram of our approach

3.1 Android App Dataset

AndroZoo [37, 56] is a representative dataset that is currently widely used in many studies. The dataset collected a large number of apps from multiple sources, including the official Google Play App Market and continues to grow. Additionally, these apps are constantly being analyzed and classified by dozens of different anti-virus software. It was judged and used as the most appropriate dataset for this study in terms of quantity and quality.

The dataset for this work consists of the benign dataset and malicious dataset from the AndroZoo database [37]. The benign dataset has 27,041 Android apps published during 2017-2018. The malicious dataset has 26,276 malware (malicious apps) found during 2014-2018. To mitigate the imbalance of the number of benign and malicious apps, we collect malicious apps over a longer period.

3.2 Feature Extraction

We can statically extract API call information from each Android application packages (APK) file. First, we obtain `classes.dex` and `AndroidManifest.xml` files by decompressing an APK file. The manifest file declares permissions that the app needs. We decode the `AndroidManifest.xml` using `appt` [49] and extract the declared permissions. The `classes.dex` is an executable file format for the Android platform. We decompile `classes.dex` using the existing reverse assembler `DexDump` included in the Android studio [50], extract all API calls from the apps, and remove third-party APIs. Most of the previous studies [33-34] using API calls as a feature for machine learning considered only the class and method name of each API. In our work, the method descriptor (arguments and return data type) is also included in the feature set.

After extracting API calls and permissions, we create a feature vector for each app from the extracted features according to the procedure shown in Figure 2. Each app has two sub-vectors: one for API calls and the other for permissions. If an API call is invoked or permission is declared in the app, the corresponding element of the vector is set to 1, otherwise 0. We do not count how many times an API call is invoked. We build a list of all official Android APIs from the API level 25 SDK (`android.jar` file) of Android Studio [44]. The total number of different APIs is 133,271 and the total number of different permissions 474, as explained in Section 2.2. As a result, each app has a vast number of features, i.e., the dimensionality of features becomes very high.

3.3 Feature Selection

To decrease the dimensionality of features, we try to remove the irrelevant APIs and permissions that rarely contribute to malware detection. We use minimal domain knowledge-based and Gini importance-based techniques.

3.3.1 Minimal Domain Knowledge-based Method

To decrease the dimensionality of features, we first exclude all unofficial APIs and permissions from the extracted features since third-party APIs are often used only in a few apps as mentioned in Section 2.1. We then remove irrelevant APIs and permissions applying domain knowledge. Domain knowledge refers to the valid expertise used in a specific specialty rather than general knowledge [51]. In the field of malware detection, domain knowledge includes the knowledge about the functionality, the behavior, the patterns, or the intention of malware. We choose relevant API calls and permissions based on this minimal domain knowledge through malware analysis and its related literature.

By adopting the results of [33-34], we select relevant 1,848 APIs among 133,271 official APIs. Some of them are listed in Table 1. It includes account-related APIs, location-related APIs, SMS-related APIs, etc. On the other hand, Google defines 9 dangerous permission groups and declares the permissions in these groups as dangerous [32, 46]. We collect such permissions in Android API level 4~28 and select 79 permissions as relevant. Table 2 shows some of them. It includes account-related permissions, Bluetooth-related permissions, location-related permissions, etc.

3.3.2 Combining the Minimal Domain Knowledge-based and GINI importance-based Selection Methods

In this step, we calculate the degree of importance of features selected in Section 3.3.1. We use the Gini importance [35-36] to measure the importance of each feature, which is included in the Random Forest library of `Scikit-learn` [52-53]. In `Scikit-learn` implementation, the node importance ni_j is defined as the decrease in the weighted Gini impurity as Equation (1)

$$\boldsymbol{ni_j = w_j C_j - w_L C_L - w_R C_R}, \quad \textbf{(1)}$$

where L and R are the child nodes, C_i is the Gini impurity of node i, and the weight is a ratio of samples reaching the node. And the importance of feature i in a tree is defined as Equation (2)

$$\boldsymbol{fi_i = \frac{\Sigma_{j \in NS(i)} ni_j}{\Sigma_{k \in N} ni_k}}, \quad \textbf{(2)}$$

where $NS(i)$ is the set of nodes that split on feature i and N is the set of all nodes. Then, the Random Forest-level feature importance is the average of fi_i over all trees. A higher fi_i value means that the feature is

Table 1. A partial list of APIs selected using domain knowledge

Some APIs (of the selected 1,848 APIs)	
User account API	Landroid/accounts/AccountManager;.getAccounts:()[Landroid/accounts/Account;
	Landroid/accounts/AccountManager;.clearPassword:(Landroid/accounts/Account;)V
	Landroid/accounts/AccountManager;.getPassword:(Landroid/accounts/Account;)Ljava/lang/String;
Bluetooth API	Landroid/bluetooth/BluetoothAdapter;.enable:()Z
	Landroid/bluetooth/BluetoothAdapter;.isEnabled:()Z
GPS/Location API	Landroid/location/LocationManager;.addGpsStatusListener:(Landroid/location/GpsStatus$Listener;)Z
	Landroid/location/LocationManager;.requestLocationUpdates:(JFLandroid/location/Criteria;Landroid/app/PendingIntent;)V
Audio API	Landroid/media/AudioRecord;.startRecording:()V
SMS API	Landroid/telephony/SmsManager;.sendDataMessage:(Ljava/lang/String;Ljava/lang/String;S[BLandroid/app/PendingIntent;Landroid/app/PendingIntent;)V
	Landroid/telephony/SmsManager;.sendTextMessage:(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;Landroid/app/PendingIntent;)V
Telephony API	Landroid/telephony/TelephonyManager;.getSimSerialNumber:()Ljava/lang/String;
	Landroid/telephony/TelephonyManager;.getDeviceId:()Ljava/lang/String;
Process API	Ljava/lang/Runtime;.exec:(Ljava/lang/String;)Ljava/lang/Process;
Notification API	Landroid/app/NotificationManager;.notify:(Ljava/lang/String;ILandroid/app/Notification;)V
String API	Landroid/lang/StringBuilder;.append:(Ljava/lang/CharSequence;)Ljava/lang/StringBuilder;

Table 2. A partial list of permissions selected using domain knowledge

Some Permissions (of the selected 79 permissions)	
User account Permission	GET_ACCOUNTS
	MANAGE_ACCOUNTS
	GET_ACCOUNTS_PRIVILEGED
Bluetooth Permission	BLUETOOTH
	BLUETOOTH_ADMIN
	BLUETOOTH_PRIVILEGED
Location permission	ACCESS_COARSE_LOCATION
	ACCESS_FINE_LOCATION
Camera permission	CAMERA
SMS permission	SEND_SMS
	READ_SMS
	WRITE_SMS
Phone info permission	READ_PHONE_STATE
	MODIFY_PHONE_STATE
	READ_PHONE_NUMBERS
Billing permission	Vending.BILLING
Launcher permission	com.android.launcher.permission.INSTALL_SHORTCUT
Overlay permission	android.permission.SYSTEM_ALERT_WINDOW

more suitable in classifying sample apps as malicious or benign

Figure 3 lists the top 20 APIs in the order of decreasing feature importance. The API call for displaying notifications in the `Notification Bar` is the most important. The other important APIs include the APIs related to the `ContentResolver` object that accesses data in the Content Providers or gets information about system settings, the APIs related to the Handler class for Android inter-thread communication, the APIs to perform operations like locating a device, the APIs for Wi-Fi or Bluetooth services, and the APIs for file write operation. On the other hand, SMS and audio-related APIs presented in Table 1 do not have important effects on Android malware detection. We found 861 APIs have the feature importance of 0 (zero).

Figure 4 lists the top 20 permissions selected in the order of decreasing feature importance. The `READ_PHONE_STATE` permission is ranked first. It allows access to device-specific information such as IMEI and phone number. The permissions associated with the file system, Wi-Fi service, and Android launcher have also high importance scores. On the other hand, SMS and Bluetooth-related permissions are ranked below 25th among the 79 permissions selected by the minimal domain knowledge. No permission has the importance of zero.

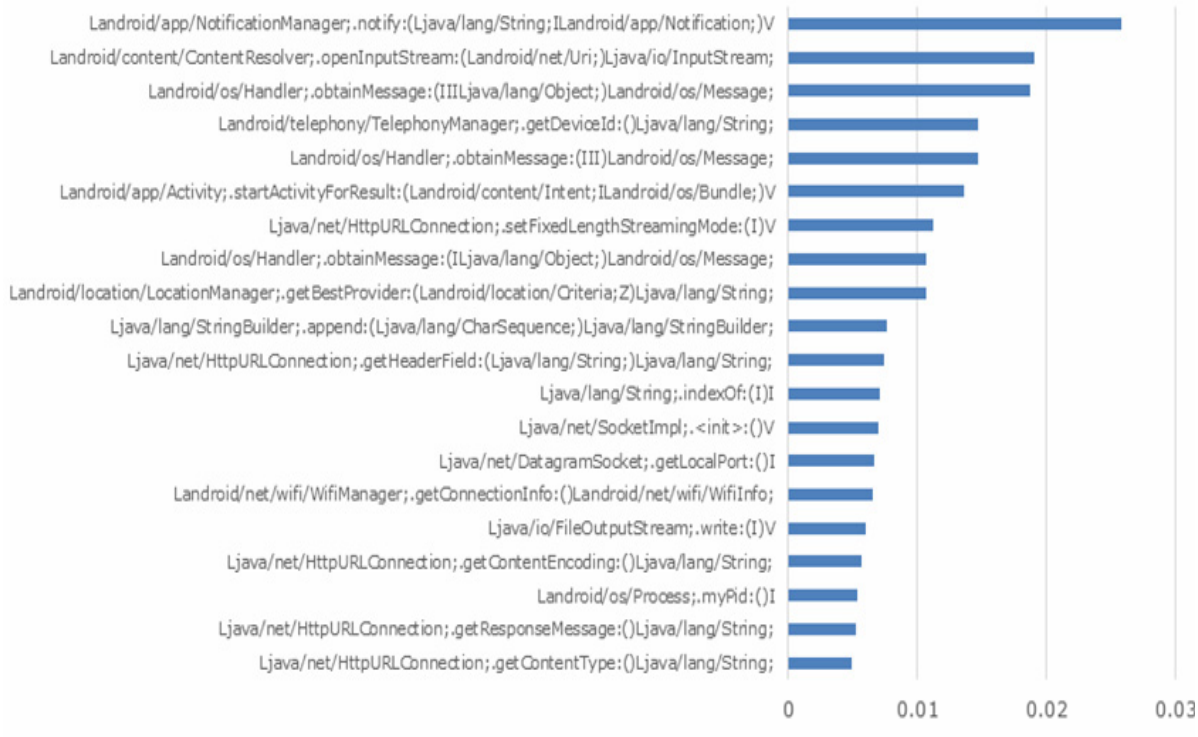

Figure 3. The top 20 APIs in order of feature importance

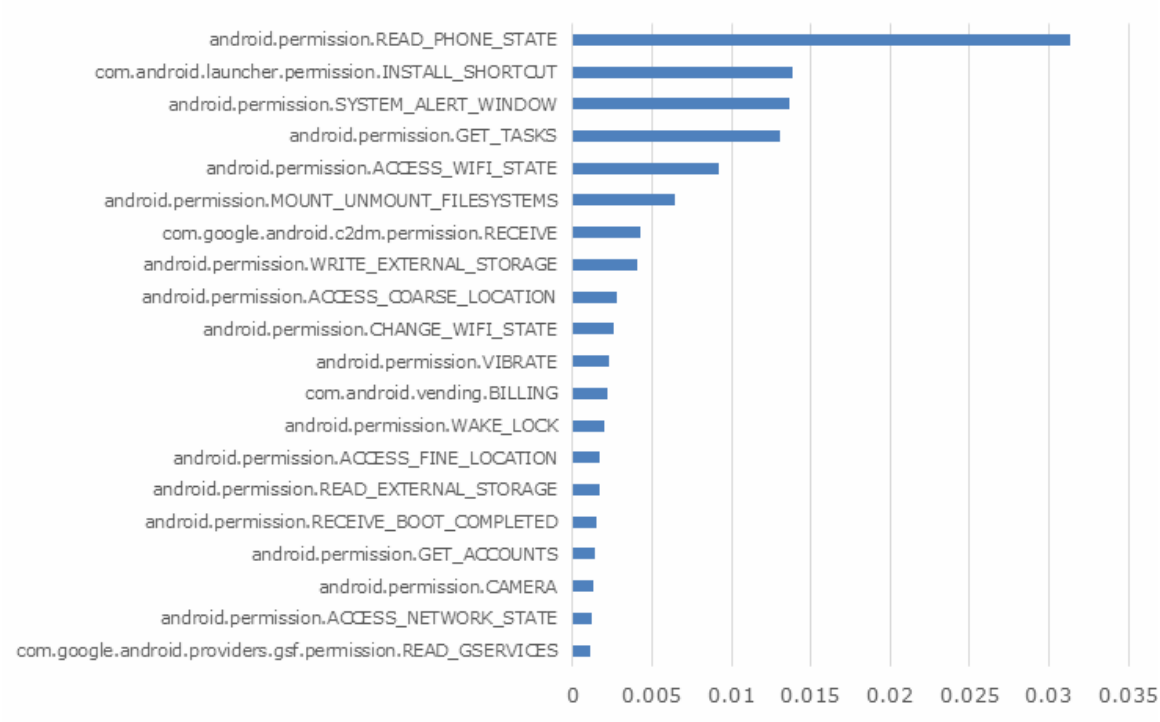

Figure 4. The top 20 permissions in order of feature importance

Based on the feature importance, we select the top N APIs and the top M permissions as a feature for machine learning. We perform a grid search for the best combination of N and M. Incrementing N from 5 to 987 and M from 5 to 79 with a step of 5 respectively, M+N features were tested. Note that the maximum value of N is 987 (excluding 861 APIs with zero importance). We found that when N=405 and M=25, Random Forest shows the highest accuracy for detecting Android malware with the least computational overhead.

3.4 Machine Learning Models

We developed the machine learning model for classifying Android apps into malicious or benign using the features selected in Section 3.3. We choose the Random Forest algorithm and use grid search to determine hyper-parameters. Random Forest has the

following advantages [54-55]: (1) it has a relatively small number of parameters that should be controlled, and removes the need for pruning the trees, (2) it can achieve high classification accuracy, (3) it can overcome the problem of overfitting, and (4) feature importance is computed automatically. Random Forest takes several hyper-parameters. In our experiments we consider two important parameters among them: `max_depth` and `n_estimators`, which control the maximum depth of each tree and the number of trees in the forest, respectively. We perform a grid search to find out the parameter values with which the Random Forest model achieves the highest detection accuracy on our datasets.

4 Experiments and Analysis

4.1 Dataset

In our experiments, we leverage the `AndroZoo` dataset [37, 56], a well-known large-scale collection of Android apps. `AndroZoo` collects Android apps from several sources including `Google Play` and `VirusShare`, and is currently being updated. Recent research such as [57-58] used the `AndroZoo` dataset in their experiments.

To construct a balanced dataset, we collected a similar number of benign apps and malicious ones. For the benign dataset, we downloaded 27,364 benign apps from the `AndroZoo` website between 2017 and 2018. For the malware dataset, we also downloaded 26,438 malicious apps between 2014 and 2018. Then we removed apps from which we cannot extract any API calls or permissions. We also removed apps that belong to both datasets. The resulting dataset consists of 27,041 benign apps and 26,276 malware.

Table 3 compares our dataset with other well-known datasets in terms of the average number of used APIs. As mobile users require more useful and convenient functions, recent apps use more APIs. This fact makes extracting and selecting significant features more important for the efficiency and effectiveness of machine learning. The high dimensionality of features may lead to computational difficulty, classification noise, or overfitting.

Table 3. The average number of APIs used

	Dataset	Collection period	Average # of APIs
Benign apps	Google Play	2014 ~ 2016	2,618
	AndroZoo [37, 56] (our dataset)	2017 ~ 2018	3,508
Malicious apps	Drebin [33]	2010 ~ 2012	521
	AMD [40]	2010 ~ 2016	1,077
	AndroZoo [37, 56] (our dataset)	2014 ~ 2018	1,954

4.2 Metrics

We describe the performance of our machine learning model based on a confusion matrix (Table 4), commonly used in machine learning. The performance metrics we consider are recall (True Positive Rate), specificity (True Negative Rate), and accuracy, which can be derived from the confusion matrix. Their definitions are as follows.

Table 4. Confusion matrix

		Prediction	
		Malicious	Benign
Ground Truth	Malicious	TP (True Positive)	FN (False Negative)
	Benign	FP (False Positive)	TN (True Negative)

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

$$\text{Specificity} = \frac{TN}{TN + FP}$$

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

The ground truth indicates that we already know if the app is malicious or benign. Reliable ground truth is essential to verify malware detection models. For building a reliable ground truth dataset, we rely on `AndroZoo`'s classification and `VirusTotal` anti-virus decisions. The malicious dataset consists of the apps that three or more anti-virus software of `VirusTotal` judged to be malicious. The benign dataset consists of the apps that all anti-virus software judged to benign.

4.3 Experiments

First, we measure the performance using features after applying the domain knowledge-based feature selection. We construct the feature vector with relevant 1,848 APIs and 79 permissions as explained in Section 3.3.1. We evaluate our scheme using 5-fold cross-validation. The samples are randomly grouped into 5 disjoint subsets of equal size. The Random Forest is trained and tested five times using each subset as test data and the others as training data. The detection

accuracy is 96.33 % with the training time 38.89s and the testing time 1.12s on average. Table 5 shows the prediction results when the model performs best. The detection accuracy is 96.72%, the recall is 97.15%, and the specificity is 96.30%.

Table 5. The best prediction results with the domain knowledge-based feature selection

		Prediction	
		Malicious	Benign
Ground Truth	Malicious	5,105	150
	Benign	200	5,208

Then we measure the performance using features after applying the combined feature selection. The feature vector is composed of 405 APIs and 25 permissions as explained in Section 3.3.2. We also employ 5-fold cross-validation. The detection accuracy is 96.51 % with the training time 12.06s and the testing time 0.62s on average. Table 6 shows the prediction results when the model performs best. The detection accuracy is 96.85%, the recall is 97.09%, and the specificity is 96.61%.

Table 6. The best prediction results with the combined feature selection

		Prediction	
		Malicious	Benign
Ground Truth	Malicious	5,102	153
	Benign	183	5,225

To show the selection method is effective, we also measure the performance of the model before the feature selection. Before the feature selection, the total number of APIs is 133,271 and the total number of permissions is 474. If we use all the APIs as a feature, the training could take too long, thus, we applied the domain knowledge-based selection to APIs only. Table 7 summarizes the number of features, training time, and accuracy. The combined feature selection approach reduces the training time by 79.60% compared with the domain knowledge-based approach (only to APIs) and 69.00% the domain knowledge-based approach. Also, it achieves almost the same detection accuracy despite reduced features.

Table 7. Summary of experimental results

Feature selection	# of APIs	# of permissions	Training time	Detection accuracy
Domain knowledge-based (only to API)	1,848	474	59.11s	96.36%
Domain knowledge-based	1,848	79	38.89s	96.33%
Combined	405	25	12.06s	96.51%

To check if our model is overfitted we test it with a new dataset. We collected 200 benign apps from the AndroZoo (*AndroZoo2019*) and 200 malware from the DREBIN [33] (*Drebin*). AndroZoo2019 is a set of benign apps collected from AndroZoo during 2019. Note that we collected our original benign apps during 2017~2018 and malware during 2014~2018; both were collected from AndroZoo. No new apps are in our original datasets. We train our model with our original datasets and test it with the new dataset. The results are shown in Table 8. The detection accuracy is 96.0%, the recall is 97.5% and the specificity is 94.5%. This means that our model is not overfitted.

Table 8. Prediction results with the new test dataset

		Prediction	
		Malicious	Benign
Ground Truth	Malicious	195	5
	Benign	11	189

Adversarial machine learning is a technique that tries to deceive machine learning models into misclassification by modifying input data. One of the strategies of adversarial machine learning is an evasion attack. Attackers obfuscate their apps to hide or distort the features and behaviors and evade detection. We measure the performance of our model against evasion attacks. We conducted an experiment corresponding to `DexGuard`-based obfuscation attack in the attack scenarios of [75]. We train the model with our `AndroZoo` dataset, then test it with 200 benign apps collected from the `F-Droid` project [76] before and after obfuscation. We obfuscate the apps using `Obfusapk` [77] (with reflection). Out of 200 apps, our model misclassified 6 apps before obfuscation and 14 apps after obfuscation. The accuracy decreases from 97% to 93%.

4.4 Analysis of Misclassified Apps

This section analyzes some of the falsely classified apps in the worst performance experiment of the combined feature selection approach. They are 66 malicious apps (false negative) and 142 benign apps (false positive). We discuss the possible reasons for the misclassification in terms of code obfuscation, grayware, and cross-platform development tools.

4.4.1 Code Obfuscation

From a laborious manual analysis, we discover that all misclassified apps are obfuscated. Most obfuscators support identifier renaming and/or API hiding [6, 59-60]. Identifier renaming changes the names of packages, classes, and methods. If any of them is changed, the extracted APIs cannot be found in the list of the official APIs. API hiding hides the names of invoked APIs using the Java reflection mechanism. API invocation codes are replaced with the codes for

finding and calling APIs via Java reflection-related APIs. These types of code obfuscation can transform the functional parts of the apps by altering the API invocations. Therefore, code obfuscation can significantly degrade the performance of API call-based malware detection.

4.4.2 Grayware

Grayware is an unwanted application that is not classified as malware by most anti-malware products but behaves in an undesirable manner or causes security risks. Grayware is neither benign nor malicious. Grayware includes spyware, adware, remote access tools, etc. Some grayware tagged as malware are predicted as benign, and vice versa.

We investigate the 66 undetected malicious apps. They are divided into 14 malware families as shown in Figure 5. We found that about 75 % of them (50 out of 66) are adware. Their families are `Dowgin`, `Kuguo`, `Jfpush`, `Feiwo`, and unknown adware. A typical adware program displays advertising sentences in the notification bar. If a user touches the notification, an advertisement is displayed in a `WebView` component. No permission is required to display a sentence in the notification bar. And the ranks of `WebView`-related APIs in our API ranking are 270 ~ 325 as shown in Table 9, which means that the importance of `WebView`-related APIs is relatively low.

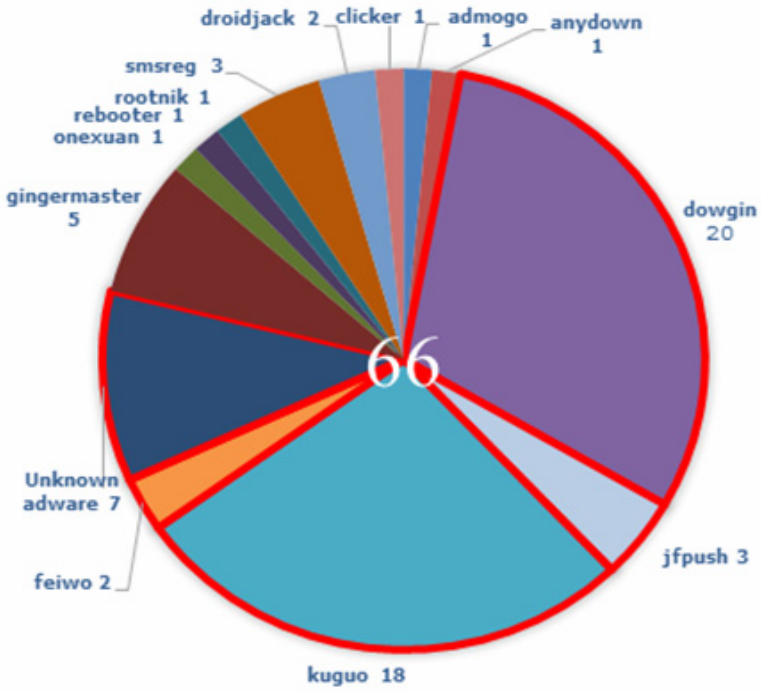

Figure 5. Malware families of undetected malicious apps

Table 9. Example of WebView-related APIs. The column *Rank* denotes the importance rank in the API list

Rank	API
270	Landroid/webkit/WebView;.setWebViewClient:(Landroid/webkit/WebViewClient;)V
271	Landroid/webkit/WebViewClient;.shouldInterceptRequest:(Landroid/webkit/WebView;Ljava/lang/String;)Landroid/webkit/WebResourceResponse;
279	Landroid/webkit/WebView$HitTestResult;.getType:()I
285	Landroid/webkit/WebView;.setFocusable:(Z)V
...	...
325	Landroid/webkit/WebView;.removeJavascriptInterface:(Ljava/lang/String;)V

We submit the 142 misclassified benign apps to `VirusTotal` [61] in June 2019. `VirusTotal` judged nine of them as malware (Table 10), but only one or two of about 70 anti-malware products classified them as malware. We found that these apps are grayware. These apps request unnecessary permissions or use APIs for the subsidiary functionality such as advertisements or information sharing. However, the relevant features rank high. These features may cause our approach to misclassify apps as malware. Figure 6, for example, shows the screenshot of '`com.unicrios.funnyskeleton`' app. This app provides live wallpapers. Users can set animation speed and send feedback to `Google Play` Store. Its functionality is simple, but it requires an unnecessary permission `WRITE_EXTERNAL_STORAGE` and contains `WebView`-related APIs that are irrelevant to its functionality.

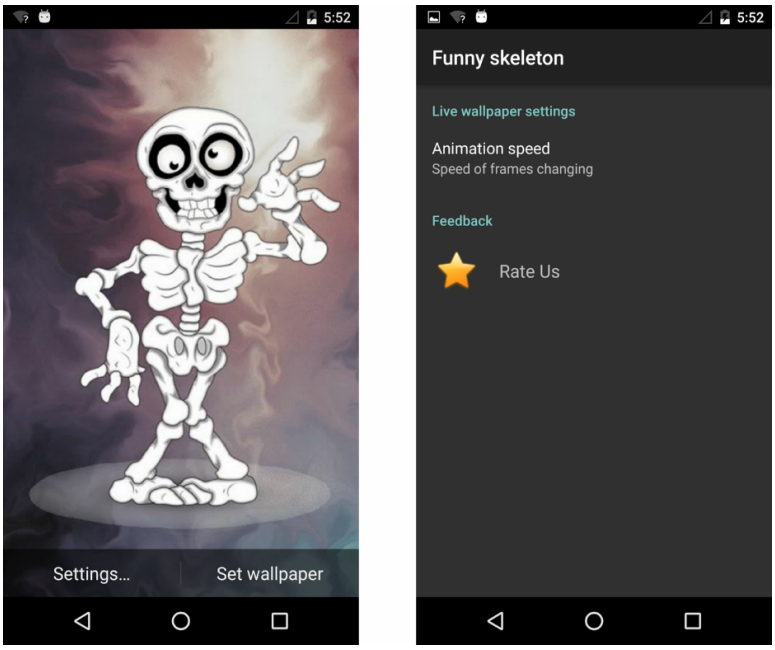

Figure 6. Screen shots of `com.unicrios.funnyskeleton`

Table 10. Scanning result for 9 misclassfied apps

Applications	# of anti-malware products that classify the app as malware
de.resolution.yf_android	1
net.kilho.CandleLight	1
com.virtualanimalsworld.chihuahuahomesimulator	2
com.unicrios.funnyskeleton	2
mfmotasouthwestregion6.org	2
com.saklalabs.vitalsecuritytoolkit	2
com.thunkable.android.devbid9.iKiwi	2
com.webroot.security.sme	1
com.ringer.ui	1

Figure 7 shows the screenshots of another game app. In Figure 7, the left figure displays a game scene, the middle one advertisement, and the right one "privacy policy". This app collects IMEI information, network information (IP address and Wi-Fi information), and location information for advertisements and service improvement. So this app contains several permissions and APIs, which rank high, as shown in Table 11. These permissions and APIs have little to do with the functionality of the game but may cause our model to classify the app as malware.

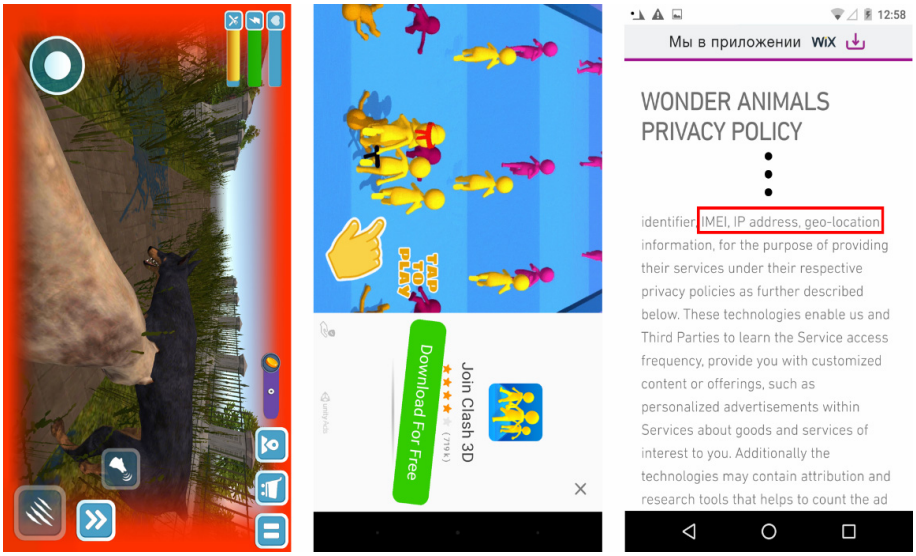

Figure 7. Screenshots of `com.virtualanimalsworld.chihuahuahomesimulator`

Table 11. Several features of com.virtualanimalsworld.chihuahuahomesimulator

Category		Rank	Feature
Permission	Device	1	READ_PHONE_STATE
	Display	3	SYSTEM_ALERT_WINDOW
	Network	4	ACCESS_WIFI_STATE
	Location	9	ACCESS_COARSE_LOCATION
API	Device	4	Landroid/telephony/TelephonyManager;.getDeviceId:()Ljava/lang/String;
	Location	9	Landroid/location/LocationManager;.getBestProvider:(Landroid/location/Criteria;Z)Ljava/lang/String;
	Network	11	Ljava/net/HttpURLConnection;.getHeaderField:(Ljava/lang/String;)Ljava/lang/String;
	Network	15	Landroid/net/wifi/WifiManager;.getConnectionInfo:()Landroid/net/wifi/WifiInfo;

4.4.3 Cross-Platform Development Tools

Cross-platform development tools such as `Xamarin` [62], `Unity` [63], `PhoneGap` [64], `Titanium` [65], and `Cocos2D` [66] are employed by many mobile app developers to reduce the development cost and easily distribute apps across multiple platforms [67-69]. Malware writers also employ those cross-platform development tools to develop malware at a low cost and infect as many devices as possible [67-68, 70].

Android apps developed using the cross-platform development tools usually have additional folders and files that are not found in native apps as shown in Figure 8. This means that we need to analyze those additional folders/files as well as `classes.dex` for malware detection. Table 12 lists the additional files that are contained in apps developed using each cross-platform development tools. Among them, `*.so`, `*.dll`, and `*.js` files are program files that cannot be decompiled using Android reverse engineering tools. Thus, our Android API and permission-based approach cannot extract suitable features from these files for detection. This increases the false negative instances.

Table 12. Additional files in apps developed using cross-platform development toos

Tools	Additional files
PhoneGap	Index.html Index.js
Titanium	Index.js
Unity	Assembly-Csharp.dll System.dll System.core.dll libunity.so libmain.so libmono.so
Xamarin	App.dll
Cocos2D	Libcocos2dcpp.so

We investigate the structure of each undetected malware and identify its development tool. Figure 9 shows the development tools of the 66 undetected malware. There are 47 malware instances written in Java, and 13 malware instances written with

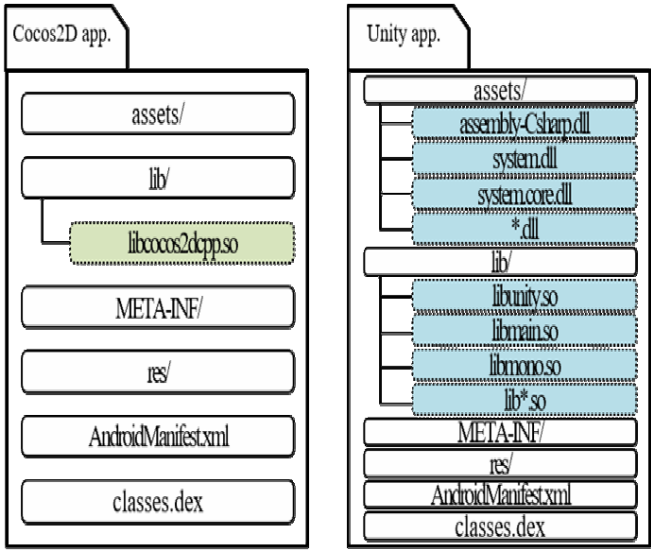

Figure 8. Structure of APK written with Cocos2D and unity

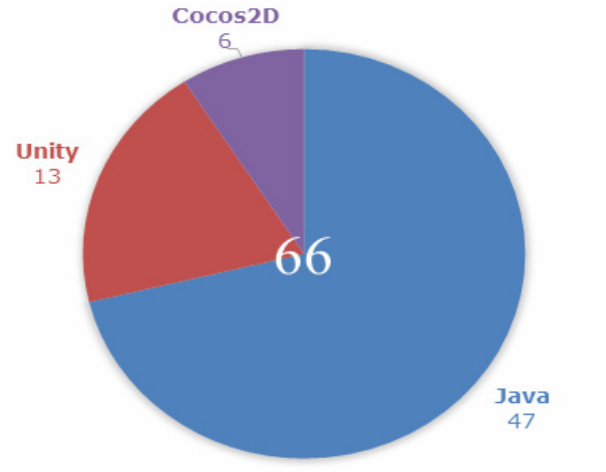

Figure 9. Development toos of undetected malware. 'Java' denotes native apps

`Unity`/C#. The remaining 6 malware instances are written with `Cocos2D`/C++. We check each `lib*.so` and `*.dll` files of `Unity` apps and `Cocos2D` apps using `VirusTotal`. According to the results, most malicious codes are found in `classes.dex`. For only two malware instances of `Gingermaster` family have malicious codes in

libieunh.so, a malicious advertisement library (Figure 10). We conclude that the effect of cross-platform development tools on our malware detection approach is relatively small.

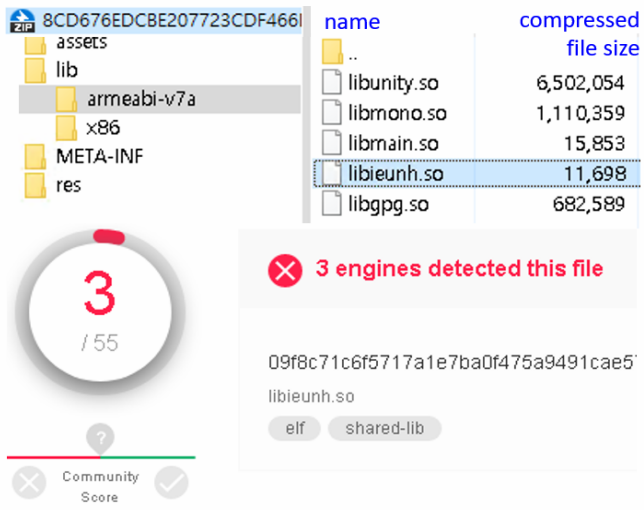

Figure 10. Malware detection result on `libieunh.so`

5 Related Work

In this paper, we mainly aim at performing static analysis for detecting Android malware using machine learning. Static analysis is an approach that evaluates Android apps by scanning their executable code without runtime analysis. The static features are obtained without executing the sample apps. On the contrary, dynamic analysis conducts malware detection by executing sample apps and monitoring their behavior.

Dynamic analysis need to mimic the actual runtime environment and simulate effectively human operations to achieve high code coverage. Static analysis has several advantages over dynamic analysis. It does not need any execution scenario as well as the notions of test case. It can be implemented in a lightweight manner for deployment on computing resource-limited devices and operate on a stand-alone basis on a mobile device. In addition, there is no possibility for mobile devices to be infected by malware during its analysis. In this work, therefore, we focus on static analysis.

Several studies on Android malware detection adopt machine learning algorithms and use APIs and permissions as the features. These studies have considered various criteria in selecting APIs and permissions for efficient malware detection. Table 13 summarizes those studies.

Table 13. Comparison of our study and existing studies on Android malware detection

	Features	Static/ Dynamic	Feature selection (Feature refinement)	Dataset (Malware/ Benign)	Acc.	Classifier
Peiravian et al. [19]	APIs, Permissions	Static analysis	None	1,260 / 1,250	96.88%	SVM, J48, Bagging
Arp et al. [33]	APIs, Permissions, Network addresses, Filtered intents, etc.	Static analysis	feature weight	5,560 / 123,453	94%	SVM
Aafer et al. [34]	APIs (with arguments)	Static analysis	frequency analysis + data flow analysis	3,987 / 16,000	99%	k-NN, ID3 DT, C4.5 DT, SVM
Chan et al. [72]	APIs, Permissions,	Static analysis	information gain	175 / 621	92.36%	NB, SVM, RBF Network, MLP, Liblinear, J48, RF
Qiao et al. [73]	APIs, Permissions	Static analysis	ANOVA, SVM-RFE	1,260 / 5,000	94.41%	SVM, RF, NN
Zhu et al. [74]	Sensitive APIs, Permission rate	Static analysis	TF-IDF, cosine similarity	1,065 / 1,065	88.26%	Rotation Forest, SVM
Li et al. [71]	Permissions	Dynamic analysis	multilevel data pruning (PRNR, SPR, PMAR)	5,494 / 310,926	95.63%,	FT, RF, Random Committee, SVM, Rotation Forest, PART
Salah et al. [79]	Symmetric patterns	Static analysis	*FF_AF* based on TF_IDF	5,560 / 123,453	99%	SVM, Logistic regression, SGD AdaBoost, LDA
Our study	APIs (with arguments, return type), Permissions,	Static analysis	Gini importance based method	26,276 / 27,041	96.51%	RF

Peiravian et al. [19] employed three machine learning models, Bagging, J48 and Support Vector Machine (SVM) with API calls and permissions as features. They performed experiments using a total of

2510 samples including 1260 malicious and 1250 benign apps, and the experiments demonstrated that Bagging achieved the best performance in classifying the datasets. They used a relatively small dataset compared to our work. Their scheme differs from ours in that it does not have feature selection step. The reduced number of permissions and APIs make our scheme perform efficiently.

Arp et al. [33] developed the machine learning technique called DREBIN which resorted to static analysis for malware detection on Android mobile device. From Android apps, DREBIN extracted APIs, permissions, hardware components, filtered intents, network addresses, etc. The extracted features were presented as strings and organized as eight different feature sets. They embedded the features into a high-dimensional vector space. After representing Android apps as feature vectors, DREBIN learned a linear SVM algorithm to classify. A dataset of about 120,000 apps is used for training and detection. The evaluation results indicated that DREBIN could achieve a detection accuracy rate of 94% by incorporating numerous features. However, utilizing too many features can increase the computational overhead [71].

Li et al. [71] presented a permission usage-based malware detection system SigPID. Through three-levels of permission pruning methods, they identified 22 significant permissions. Then they experimented SigPID using 67 machine learning models and found that Functional Tree (FT) yielded the highest recall with the shortest processing time. They also compared SigPID with other malware detection approaches such as DREBIN [33] and showed that SigPID+FT achieved a high detection rate in spite of a small number of features (22 permissions).

Aafer et al. [34] proposed DroidAPIMiner that used API call information including parameter values. They deployed four classifiers: SVM, k-NN, C4.5, and ID5. They collected around 20,000 apps (3,987 malware and around 16,000 benign apps) and the classifiers achieved a high accuracy (up to 99%).

Chan et al. [72] also considered permissions and APIs. The authors selected permissions and API calls with a positive information gain. They conducted the experiments using WEKA using several machine learning algorithms. On 796 apps (621 benign and 175 malicious), the classifiers achieved the accuracy of 92.36%.

Qiao et al. [73] utilized the patterns of API calls and permissions. They considered APIs that were controlled by permissions. They classified benign and malicious apps using SVM, RBF kernels, Random Forest, and Artificial Neural Networks. Using 6260 apps (5,000 benign and 1,620 malware), the classifiers with the feature selection achieved an accuracy of about 78~94%.

Zhu et al. [74] presented DroidDet. The information considered in this work is permission requests, APIs, permission-rate and monitoring system events. They scored each feature through methods such as TF-IDF or cosine similarity to select top features. At classification stage, an ensemble classifier Rotation Forest is employed. With 2,130 samples (1,065 benign and 1,065 malware), the classifier achieves an accuracy of 88.26%, which is higher than SVM by 3.33% under the same experimental conditions.

Salah et al. [79] found out symmetric features across malicious Android applications. They took into account different types of static features and chosen the most important features to detect Android malware. They introduced a frequency-based feature selection method called the *feature frequency-application frequency (FF - AF)* to reduce the feature space size, and merged Android app URLs into a single feature called the *URL_score*. The proposed method was evaluated using five machine learning classifiers with the DREBIN dataset. They used 349 features from the six feature categories such as APIs, permissions, app components, etc. The linear SVM of the five classifiers showed the highest accuracy up to 99%.

All the aforementioned studies selected features based on domain knowledge. For example, DREBIN [33] analyzed malware, selected relevant APIs, and used them as feature. Other approaches selected feature(s) based on statistical analysis or data mining with domain knowledge [34, 71-74]. For example, in [34], after selecting APIs related to malicious behavior, the authors analyzed the frequency of APIs in normal apps and malware and selected APIs with the large difference in the frequency. In this paper, we select features using minimal domain knowledge, and then select relevant features among them using Gini importance-based method. Specifically, features are selected based on the algorithm of decision trees in Random Forest, which is a kind of statistical analysis method, and the experimental results before and after the analysis are presented. Most of all, we analyze the falsely classified apps and suggest future work.

Su et al. [80] constructed the behavioral portrait of Android malware to depict behaviors of malware samples and detect them based on both static and dynamic analysis. They defined several dimensions of behavioral features to depict malware, and defined behavioral tags to generalize meta-data of the features. They then analyzed the correlation of the behavior tags to construct a behavioral portrait of Android malware. Finally a random forest algorithm was combined with the behavior portrait of malware for Android malware detection.

Alswaina et al. [78] reviewed the literature over the past 10 years related to Android malware families by surveying on Android malware family detection, identification, and categorization techniques. The survey was conducted using three dimensions: analysis type (static, dynamic, hybrid), feature (static, dynamic),

and techniques (model-based, analysis-based). They introduced a new taxonomy that could categorize malware familial classification-related studies in terms of the three dimensions. The limitations of the related studies and future trends have been highlighted too.

A meta-classifier or classifier fusion approach extracts features from Android apps, trains several base classifiers with the features, and collates their detection results, and selects a final model [81-82]. The performance of this approach depends upon the accuracy of individual base classifiers. If base classifiers cannot detect malware accurately, the performance of the final classification is limited. Hence, studies on effective base classifiers, like our work, are significant.

6 Conclusions

In this paper, we proposed feature extraction and selection techniques that use API call and permission information as features of a machine learning model for classifying efficiently and effectively Android apps into malicious or benign. For the API call information, we used as features class name, method name, and arguments and return data type of each method. Since Android apps contains a very large number of features, it is necessary to reduce the number of features. By combining a minimal domain knowledge-based and Gini importance-based methods, we finally selected 405 APIs and 25 permissions out of 133,271 APIs and 474 permissions, respectively. We constructed a dataset that is balanced and large enough to build a generalized machine learning model. We downloaded the latest Android sample apps, 27,041 benign apps and 26,276 malware, from the AndroZoo dataset. We then conducted some experiments on the sample apps. The experiment results showed that our technique had the classification accuracy of 96.51% using the features selected by the combined methods. It reduced the training time by 68.99% without degrading the classification accuracy.

In addition, we demonstrated the superiority of our model by performing another experiment with a new test dataset, where no apps in the new dataset are in the aforementioned dataset. The experiment results achieved the accuracy of 96%. This implies that our model is not overfitted.

We finally investigated the misclassified 66 malicious apps and 142 benign apps in detail and discovered that the performance of our model can be degraded by code obfuscation, grayware, and cross-platform development tools. Specially, API hiding using Java reflection can be a major obstacle to Android malware detection based on API calls because it conceals the functional parts of the sample app by hiding the API calls in the app. Meanwhile, about 75% of the undetected malicious apps and 6.3% of misclassified apps were greyware such as adware, spyware, etc. Our experiment results showed that many anti-malware products of `VirusTotal` could not detect grayware correctly. In order to correctly detect Android grayware using machine learning, it is necessary to build reliable ground truth dataset for current grayware. Therefore, we plan to construct a reliable ground truth dataset for grayware in the future.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2018R1A2B2004830).

References

[1] Android Things Home Page, https://developer.android.com/things/get-started, March, 2020.

[2] M. Chibuye, J. Phiri, A Remote Sensor Network using Android Things and Cloud Computing for the Food Reserve Agency in Zambia, *International Journal of Advanced Computer Science and Applications (IJACSA)*, Vol. 8, No. 11, pp. 411-418, 2017.

[3] W. Song, H. Lee, S.-H. Lee, M.-H. Choi, M. Hong, Implementation of Android Application for Indoor Positioning System with Estimote BLE Beacons, *Journal of Internet Technology (JIT)*, Vol. 19, No. 3, pp. 871-878, May, 2018.

[4] B. Sharma, M. S. Obaidat, Comparative analysis of IoT based products, technology and integration of IoT with cloud computing, *IET Networks*, Vol. 9, No. 2, pp. 43-47, March, 2020.

[5] J. Qi, P. Yang, M. Hanneghan, D. Fan, Z. Deng, F. Dong, Ellipse fitting model for improving the effectiveness of life-logging physical activity measures in an Internet of Things environment, *IET Networks*, Vol. 5, No. 5, pp. 107-113, September, 2016.

[6] T. Cho, H. Kim, J. H. Yi, Security Assessment of Code Obfuscation based on Dynamic Monitoring in Android Things, *IEEE Access*, Vol. 5, pp. 6361-6371, April, 2017.

[7] H. S. Ham, H. H. Kim, M. S. Kim, M. J. Choi, Linear SVM-based Android Malware Detection for Reliable IoT Services, *Journal of Applied Mathematics*, Vol. 2014, Article ID 594501, September, 2014.

[8] A. K. Sikder, H. Aksu, A. S. Uluagac, 6thSense: A context-aware sensor-based attack detector for smart devices, *The 26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, Canada, 2017, pp. 397-414.

[9] A. K. Sikder, H. Aksu, A. S. Uluagac, A context-aware framework for detecting sensor-based threats on smart devices, *IEEE Transactions on Mobile Computing*, Vol. 19, No. 2, pp. 245-261, February, 2020.

[10] E. B. Karbab, M. Debbabi, A. Derhab, D. Mouheb, MalDozer: Automatic framework for android malware detection using deep learning, *Digital Investigation*, Vol. 24, No. Supplement,

pp. S48-S59, March, 2018.

[11] McAfee, *McAfee Mobile Threat Report*, https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf, March, 2019.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A survey of mobile malware in the wild, *Proceedings the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, Chicago, Illinois, USA, 2011, pp. 3-14.

[13] M. Chandramohan, H. B. K. Tan, Detection of mobile malware in the wild, *IEEE Computer*, Vol. 45, No. 9, pp. 65-71, September, 2012.

[14] K. Shaerpour, A. Dehghantanha, R. Mahmod, Trends in android malware detection, *Journal of Digital Forensics, Security and Law*, Vol. 8, No. 3, pp. 21-40, 2013.

[15] S. H. Seo, A. Gupta, A. M. Sallam, E. Bertino, K. Yim, Detecting mobile malware threats to homeland security through static analysis, *Journal of Network and Computer Applications*, Vol. 38, pp. 43-53, February, 2014.

[16] M. Christodorescu, S. Jha, *Static analysis of executables to detect malicious patterns*, Technical Report at the Computer Sciences Department of the University of Wisconsin, 2006.

[17] R. W. Lo, K. N. Levitt, R. A. Olsson, MCF: A malicious code filter, *Computers & Security*, Vol. 14, No. 6, pp. 541-566, 1995.

[18] J. Sahs, L. Khan, A machine learning approach to android malware detection, *IEEE European Intelligence and Security Informatics Conference*, Odense, Denmark, 2012, pp. 141-147.

[19] N. Peiravian, X. Zhu, Machine learning for android malware detection using permission and api calls, *IEEE 25th international conference on tools with artificial intelligence*, Herndon, VA, USA, 2013, pp. 300-305.

[20] F. A. Narudin, A. Feizollah, N. B. Anuar, A. Gani, Evaluation of machine learning classifiers for mobile malware detection, *Soft Computing*, Vol. 20, No. 1, pp. 343-357, January, 2016.

[21] M. G. Schultz, E. Eskin, F. Zadok, S. J. Stolfo, Data mining methods for detection of new malicious executables, *IEEE Symposium on Security and Privacy (S&P 2001)*, Oakland, CA, USA, 2000, pp. 38-49.

[22] Z. Markel, M. Bilzor, Building a machine learning classifier for malware detection, *IEEE Second Workshop on Anti-malware Testing Research (WATeR)*, Canterbury, UK, 2014, pp. 1- 4.

[23] J. Saxe, K. Berlin, Deep neural network based malware detection using two dimensional binary program features, *IEEE 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, Puerto Rico, 2015, pp. 11-20.

[24] A. Feizollah, N. B. Anuar, R. Salleh, A. W. A. Wahab, A review on feature selection in mobile malware detection, *Digital investigation*, Vol. 13, pp. 22-37, June, 2015.

[25] N. B. Anuar, H. Sallehudin, A. Gani, O. Zakari, Identifying false alarm for network intrusion detection system using hybrid data mining and decision tree, *Malaysian journal of computer science*, Vol. 21, No. 2, pp. 101-115, December, 2008.

[26] M. Hassen, M. Carvalho, P. Chan, Malware classification using static analysis based features, *IEEE Symposium Series on Computational Intelligence (SSCI)*, Honolulu, HI, USA, 2017, pp. 1-7.

[27] Z. Zhu, T. Dumitraş, Featuresmith: Automatically engineering features for malware detection by mining the security literature, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 767-778.

[28] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, G. Giacinto, Novel feature extraction, selection and fusion for effective malware family classification, *Proceedings of the sixth ACM conference on data and application security and privacy*, New Orleans, Louisiana, USA, 2016, pp. 183-194.

[29] S. Ranveer, S. Hiray, Comparative analysis of feature extraction methods of malware detection, *International Journal of Computer Applications*, Vol. 120, No. 5, pp. 1-7, June, 2015.

[30] S. Khalid, T. Khalil, S. Nasreen, A survey of feature selection and feature extraction techniques in machine learning, *IEEE Science and Information Conference*, London, UK, 2014, pp. 372-378.

[31] B. N. Narayanan, O. Djaneye-Boundjou, T. M. Kebede, Performance analysis of machine learning and pattern recognition algorithms for malware classification, *IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, Dayton, OH, USA, 2016, pp. 338-342.

[32] Android developer, Dangerous permission group, https://developer.android.com/guide/topics/permissions/overview#permission-groups and https://developer.android.com/training/permissions/requesting#normal-dangerous, March, 2019.

[33] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket, *Network and Distributed System Security (NDSS)*, San Diego, California, USA, 2014, pp. 23-26.

[34] Y. Aafer, W. Du, H. Yin, DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android, *International conference on security and privacy in communication systems*, Sydney, NSW, Australia, 2013, pp. 86-103.

[35] B. H. Menze, B. M. Kelm, R. Masuch, U. Himmelreich, P. Bachert, W. Petrich, F. A. Hamprecht, A comparison of random forest and its Gini importance with standard chemometric methods for the feature selection and classification of spectral data, *BMC bioinformatics*, Vol. 10, No. 1, pp. 1-16, July, 2009.

[36] Y. Qi, Random forest for bioinformatics, in: C. Zhang, Y. Ma (Eds.), *Ensemble machine learning*, Springer US, 2012, pp. 307-323.

[37] K. Allix, T. F. Bissyandé, J. Klein, Y. L. Traon, Androzoo: Collecting millions of android apps for the research community, *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, Austin, Texas, USA, 2016, pp. 468-471.

[38] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, C. K. Nicholas, Malware detection by eating a whole exe, *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, New Orleans, Louisiana, USA, 2018, pp. 268-276.

[39] E. Raff, J. Sylvester, C. Nicholas, Learning the pe header, malware detection with minimal domain knowledge, *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, Dallas, Texas, USA, 2017, pp. 121-132.

[40] F. Wei, Y. Li, S. Roy, X. Ou, W. Zhou, Deep ground truth analysis of current android malware, *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Bonn, Germany, 2017, pp. 252-276.

[41] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, D. Poshyvanyk, API change and fault proneness: a threat to the success of Android apps, *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, Saint Petersburg, Russia, 2013, pp. 477-487.

[42] Z. Salehi, M. Ghiasi, A. Sami, A miner for malware detection based on API function calls and their arguments, *The 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012)*, Shiraz, Fars, Iran, 2012, pp. 563-568.

[43] Z. Salehi, A. Sami, M. Ghiasi, MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values, *Engineering Applications of Artificial Intelligence*, Vol. 59, pp. 93-102, March, 2017.

[44] Android Studio, SDK Platform release notes: Android 7.1 (API level 25), https://developer.android.com/studio/releases/platforms, January, 2020.

[45] Java Virtual Machine class file format - Method descriptors, https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.3, January, 2020.

[46] Manifest.permission, https://developer.android.com/reference/android/Manifest.permission, March, 2019.

[47] S. Liang, X. Du, Permission-combination-based scheme for android mobile malware detection, *IEEE international conference on communications (ICC)*, Sydney, NSW, Australia, 2014, pp. 2301-2306.

[48] AndroGuard Home Page, https://github.com/androguard/androguard, March, 2020.

[49] Android AAPT - Android packaging tool to create. APK file, https://androidaapt.com/, January, 2020.

[50] Android studio and Android SDK tools, https://developer.android.com/studio and https://developer.android.com/studio/command-line#tools-sdk, January, 2020.

[51] Wikipedia, Domain knowledge, https://en.wikipedia.org/wiki/Domain_knowledge, January, 2020.

[52] S. Ronaghan, The Mathematics of Decision Trees, Random Forest and Feature Importance in Scikit-learn and Spark, https://towardsdatascience.com/the-mathematics-of-decision-trees-random-forest-and-feature-importance-in-scikit-learn-and-spark-f2861df67e3, May, 2018.

[53] Scikit-learn, https://scikit-learn.org/, January, 2020.

[54] V. F. Rodriguez-Galiano, B. Ghimire, J. Rogan, M. Chica-Olmo, J. P. Rigol-Sanchez, An assessment of the effectiveness of a random forest classifier for land-cover classification, *ISPRS Journal of Photogrammetry and Remote Sensing*, Vol. 67, pp. 93-104, January, 2012.

[55] J. Ali, R. Khan, N. Ahmad, I. Maqsood, Random forests and decision trees, *International Journal of Computer Science Issues (IJCSI)*, Vol. 9, No. 5, pp. 272-278, September, 2012.

[56] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, Y. L. Traon, Androzoo++: Collecting millions of android apps and their metadata for the research community, *arXiv preprint arXiv:1709.05281*, https://arxiv.org/pdf/1709.05281.pdf, 2017.

[57] H. Cai, N. Meng, B. Ryder, D. Yao, Droidcat: Effective android malware detection and categorization via app-level profiling, *IEEE Transactions on Information Forensics and Security*, Vol. 14, No. 6, pp. 1455-1470, June, 2019.

[58] A. Hamidreza, N. Mohammed, Permission-based analysis of Android applications using categorization and deep learning scheme, *MATEC Web of Conferences, Engineering Application of Artificial Intelligence Conference 2018 (EAAIC 2018), Sabah, Malaysia, 2018*, Vol. 255, Article No. 05005, January, 2019.

[59] J. H. Park, H. J. Kim, Y. S. Jeong, S. J. Cho, S. C. Han, M. K. Park, Effects of Code Obfuscation on Android App Similarity Analysis, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, Vol. 6, No. 4, pp. 86-98, December, 2015.

[60] M. Backes, S. Bugiel, E. Derr, Reliable third-party library detection in android and its security applications, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 356-367.

[61] VirusTotal – a free virus, malware and URL online scanning service, https://www.virustotal.com/, January, 2020.

[62] Xamarin homepage, https://dotnet.microsoft.com/apps/xamarin, January, 2021.

[63] Unity homepage, https://unity.com/, 2020.

[64] PhoneGap homepage, https://phonegap.com/, 2020.

[65] Titanium Mobile Development Environment, https://www.appcelerator.com/Titanium/, 2020.

[66] Cocos2D, https://cocos2d-x.org/, 2020.

[67] J. W. Shim, K. H. Lim, S. J. Cho, S. C. Han, M. K. Park, Static and Dynamic Analysis of Android Malware and Goodware Written with Unity Framework, *Security and Communication Networks*, Vol. 2018, Article ID 6280768, June, 2018.

[68] B. Zahran, S. Nicholson, A. Ali-gombe, Cross-Platform Malware: Study of the Forthcoming Hazard Adaptation and Behavior, *Proceeding of the International Conference on Security and Management (SAM), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp)*, Las Vegas, Nevada, USA, 2019, pp. 91-94.

[69] P. Feng, J. Ma, C. Sun, X. Xu, Y. Ma, A novel dynamic Android malware detection system with ensemble learning,

IEEE Access, Vol. 6, pp. 30996-31011, June, 2018.

[70] W. Lee, X. Wu, Cross-platform mobile malware, write once, run everywhere, *Proceedings of the International Virus Bulletin Conference*, Prague, Czech Republic, 2015, pp. 352-360.

[71] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, H. Ye, Significant Permission Identification for Machine-Learning-Based Android Malware Detection, *IEEE Transactions on Industrial Informatics*, Vol. 14, No. 7, pp. 3216-3225, July, 2018.

[72] P. P. Chan, W. K. Song, Static detection of Android malware by using permissions and API calls, *International Conference on Machine Learning and Cybernetics*, Lanzhou, China, 2014, pp. 82-87.

[73] M. Qiao, A. H. Sung, Q. Liu, Merging Permission and API Features for Android Malware Detection, *IEEE 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, Kumamoto, Japan, 2016, pp. 566-571.

[74] H. J. Zhu, Z. H. You, Z. X. Zhu, W. L. Shi, X. Chen, L. Cheng, DroidDet: Effective and robust detection of android malware using static analysis along with rotation forest model, *Neurocomputing*, Vol. 272, pp. 638-646, January, 2018.

[75] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, F. Roli, Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection, *IEEE Transactions on Dependable and Secure Computing*, Vol. 16, No. 4, pp. 711-724, July-August, 2019.

[76] F-droid - Free and Open Source Android App Repository, https://f-droid.org/, September, 2020.

[77] S. Aonzo, G. C. Georgiu, L. Verderame, A. Merlo, Obfuscapk: An open-source black-box obfuscation tool for Android apps, *SoftwareX*, Vol. 11, Article 100403, January-June, 2020.

[78] F. Alswaina, K. Elleithy, Android Malware Family Classification and Analysis: Current Status and Future Directions, *Electronics*, Vol. 9, No. 6, Article No. 942, June, 2020.

[79] A. Salah, E. Shalabi, W. Khedr, A Lightweight Android Malware Classifier Using Novel Feature Selection Methods, *Symmetry*, Vol. 12, No. 5, Article No. 858, May, 2020.

[80] X. Su, L. Xiao, W. Li, X. Liu, K. C. Li, W. Liang, DroidPortrait: Android Malware Portrait Construction Based on Multidimensional Behavior Analysis, *Applied Sciences*, Vol. 10, No. 11, Article No. 3978, June, 2020.

[81] S. Y. Yerima, S. Sezer, DroidFusion: A Novel Multilevel Classifier Fusion Approach for Android Malware Detection, *IEEE transactions on cybernetics*, Vol. 49, No. 2, pp. 453-466, February, 2019.

[82] W. Wang, Y. Li, X. Wang, J. Liu, X. Zhang, Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers, *Future Generation Computer Systems*, Vol. 78, pp. 987-994, January, 2018.

Biographies

Jaemin Jung received the B.S. degree in Software Science from Dankook University, Korea, in 2018. He received his M.E. degree in Computer Science and Engineering from Dankook University in 2019. His research interests include mobile security and machine learning.

Jihyeon Park is currently an undergraduate student at Dept. of Software Science in Dankook University, Korea. Her research interests include computer system security, mobile security and machine learning.

Seong-je Cho received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 1989, 1991 and 1996, respectively. In 1997, he joined the faculty of Dankook University, Korea, where he is currently a Professor in Department of Software Science. His current research interests include computer security, operating systems, and software intellectual property protection.

Sangchul Han received the B.S. degree in Computer Science from Yonsei University in 1998. He received his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now a professor of Dept. of Software Technology at Konkuk University. His research interests include real-time scheduling, and computer security.

Minkyu Park received the B.E., M.E., and Ph.D. degree in Computer Engineering from Seoul National University in 1991, 1993, and 2005, respectively. He is now a professor in Konkuk University, Rep. of Korea. His research interests include operating systems, embedded software, computer system security, and HCI. He has authored and co-authored several journals and conference papers.

Hsin-Hung Cho received the B.S. degree from the Department of Applied Mathematics at Hsuan Chuang University, Taiwan, R.O.C. in 2010, the M.S. degree from the Institute of Computer Science and Information Engineering at National I-Lan University in 2011, and the Ph.D. degree from the Department of Computer Science and Information Engineering at National Central University. He joined the Department of Computer Science and Information Engineering at National I-Lan University as an Assistant Professor since 2017.