Journal of Computer Virology and Hacking Techniques (2019) 15:209–218
https://doi.org/10.1007/s11416-019-00332-z

ORIGINAL PAPER

Experimental analysis of Android malware detection based on combinations of permissions and API-calls

Abhishek Kumar Singh[1] · C. D. Jaidhar[1] · M. A. Ajay Kumara[2]

Received: 7 June 2017 / Accepted: 23 April 2019 / Published online: 30 May 2019
© Springer-Verlag France SAS, part of Springer Nature 2019

Abstract
Android-based smartphones are gaining popularity, due to its cost efficiency and various applications. These smartphones provide the full experience of a computing device to its user, and usually ends up being used as a personal computer. Since the Android operating system is open-source software, many contributors are adding to its development to make the interface more attractive and tweaking the performance. In order to gain more popularity, many refined versions are being offered to customers, whose feedback will enable it to be made even more powerful and user-friendly. However, this has attracted many malicious code-writers to gain anonymous access to the user's private data. Moreover, the malware causes an increase of resource consumption. To prevent this, various techniques are currently being used that include static analysis-based detection and dynamic analysis-based detection. But, due to the enhancement in Android malware code-writing techniques, some of these techniques are getting overwhelmed. Therefore, there is a need for an effective Android malware detection approach for which experimental studies were conducted in the present work using the static features of the Android applications such as Standard Permissions with Application Programming Interface (API) calls, Non-standard Permissions with API-calls, API-calls with Standard and Nonstandard Permissions. To select the prominent features, Feature Selection Techniques (FSTs) such as the BI-Normal Separation (BNS), Mutual Information (MI), Relevancy Score (RS), and the Kullback-Leibler (KL) were employed and their effectiveness was measured using the Linear-Support Vector Machine (L-SVM) classifier. It was observed that this classifier achieved Android malware detection accuracy of 99.6% for the combined features as recommended by the BI-Normal Separation FST.

Keywords Android · Feature selection · Malware detection · Static analysis

1 Introduction

Android malware, also known as Android malicious application, which always is intended either to steal private or confidential information or to harm the Android system. Android-based smartphones have been gaining popularity since 2010 [1]. Due to the rapid evolution in Android-based smartphones, malware writers have focused on this area too [2]. There are varieties of Android malware available in the market, such as ransomware, spyware, backdoor, trojan, etc. In order to prevent malware penetration, Google provides some mechanism to detect malicious Android applications during the time of application submission in the application repository. Some of the Android applications download the malicious contents after installation, so it cannot be readily detected by Google's Android malware detection approach [3,4]. Two broad categories of security techniques used to detect malicious Android applications are static analysis-based detection and dynamic analysis-based detection.

One of the popular static analysis-based detection techniques is the signature-based detection technique. It accurately detects the known malware, but is unable to perceive new malware unless the corresponding signature is available

✉ C. D. Jaidhar
jaidharcd@nitk.edu.in

Abhishek Kumar Singh
asingh370@gmail.com

M. A. Ajay Kumara
ma_ajaykumara@blr.amrita.edu

1 Department of Information Technology, National Institute of Technology Karnataka, Surathkal, Mangalore 575025, India

2 Department of Computer Science and Engineering, Amrita School of Engineering, Amrita Vishwa Vidyapeetham, Bangalore, India

in the signatures repository. To address this major issue, many other Android malware detection techniques were proposed. Each technique has its own merits and demerits. However, Google always recommends the user to download Android applications only from the Google Play Store, which also provides security over the applications during submission time from the application developers. Due to the restricted nature and device compatibility issue of the Google Play Store, most of the users use the third party application stores to get access to the applications. However, very few of these stores allow the developers to place their applications after they have verified the absence of any threat. Thus, there is every possibility of malicious applications getting downloaded onto the users smartphones without their knowledge. Android applications, which are verified either from the Google Play Store or third party application stores does not necessarily guarantee the application to be malware free [3,4].

The rest of the paper is organized as follows. Section 2 provides a brief description of the research work related to static analysis of Android malware detection. Section 3 discusses the proposed work, where static analysis and FSTs have been described. Section 4 presents the performance evaluation. Finally, Section 5 concludes the paper and presents the future work.

2 Related work

As per the current literature survey, there are many approaches available to detect Android malware from Android applications. Some of these are behavioural-based [5–7] or on-device anomaly detection [8,9], which is different from the present approach. It performs off-device analysis using static features with data mining techniques due to the performance bottleneck issues of smartphones. The major advantage of using static analysis is that it is free from malware involvement during analysis, i.e., the malware cannot modify its behaviour during analysis [10–12].

Due to the failure of the signature-based Android malware detection methods as mentioned in the Introduction, many techniques such as the static and dynamic analysis have been evolved to detect malicious Android applications [13–17].

Many authors have proposed static analysis. Majority of the existing static analysis based Android malware detection techniques use permissions and API-calls features, hence, these features have also been considered for the present work. Xiaoyan et al. [18] proposed an Android malware detection technique using permissions as features. Principal Component Analysis was used to select the prominent features. Permissions-based Android malware detection techniques are not efficient in detecting all kinds of Android malware because while these are used in the manifest file, it may not be required during runtime.

Zhu et al. [19] proposed an Android malware detection technique considering the API-calls feature, which is used to construct the API control flow graph. Three classification algorithms ID3, Naive Bayes, and SVM were used, out of which the SVM outperformed the others.

In static analysis, apart from considering individual features, many authors have proposed Android malware detection techniques using a combination of features. Peiravian and Zhu [20] and Chan and Song [21] proposed malware detection techniques using permissions and API-calls features of an Android application. The results have proved that the combination of permissions and API-calls delivered more detection accuracy compared with using them as independent features.

Qiao et al. [22] have focused on static analysis by considering permissions and API-calls features, which were used as binary and numerical features. The FSTs are used to improve efficiency as well as for dimensionality reduction. Three classifiers, Random Forest, Artificial Neural Network, and SVM were used to assess the performance. According to their results, the API-call features outperformed permissions features in both binary and numerical representations. Also, the numerical features provided better detection accuracy compared with the binary features.

Aswini and Vinod [15] have used features namely permissions, count of permission, hardware and software features and API-calls to uncover the android malware. The FSTs such as BNS, MI, RS, and KL were used in their work to measure the detection ability of their proposed approach.

Some of the authors have also considered both, the static and dynamic analysis both in their proposed methods. Su et al. [23] proposed an Android malware detection technique, which used both static and dynamic analysis. For static analysis, permissions and API-calls were used, while the dynamic analysis used the system log files. Static analysis was performed in on-device, while the dynamic analysis was performed in the off-device sandbox server.

By considering the aforementioned approaches for Android malware detection, an experimental work was performed to compare the present method's performance with other existing methods (see Table 1). This is summarized in the following paragraph.

- Using static analysis, standard and nonstandard permissions were segregated from the manifest file, which was combined with API-calls to prepare a combined features set. Standard permissions are predefined in the official Android library, whereas nonstandard permissions are defined by the application developers.
- To select prominent features as to enhance efficiency, four different FSTs as described in Section 3.3, have been used.

Table 1 Comparison of the Proposed Approach with Earlier Approaches

Method	Features	Feature Selection Technique	Malware Dataset	Detection Rate Accuracy (%)
Our approach	API + Standard Permissions + Nonstandard-Permissions	BNS, KL, MI, RS	DREBIN	99.6
Android Malware Analysis [15]	Permissions, Count of Permissions, Hardware Features, Software Features, and API-calls	BNS, KL, MI, RS	Not Mentioned	99.4
AndroDialysis [14]	Android Intent and Android Permissions	Not Mentioned	DREBIN	95.5
Aided Android Malware Classification [16]	Permissions and Source Code Analysis	Not Mentioned	MoDroid	95.6

```
|-- AndroidManifest.xml
|-- META-INF // Signature Data
|   |-- CERT.RSA
|   |-- CERT.SF
|   `-- MANIFEST.MF
|-- classes.dex //  java byte code  file generated after the compilation
|-- res // resource files
|   |-- drawable
|   |   `-- icon.png
|   `-- layout
|       `-- main.xml
`-- resources.arsc
```

Fig. 1 Android apk file content [24]

3 Proposed methodology

Figure 1 shows the structure of the unpacked contents of the Android application, where at the top is the *AndroidManifest.xml* file, which provides a road map to the Java Virtual Machine regarding the sequential running of the various activities. Furthermore, it provides package information and other components like broadcast receivers, services, activities, content providers, etc. From the *AndroidManifest.xml* file, the standard permissions and nonstandard permissions features will be extracted.

META-INF contains the MANIFEST, Certificate, and SHA-1 digest for the corresponding line of MANIFEST.MF file. The *classes.dex* is where the actual java source code of an Android application is stored in bytecode format. The API-call features will be extracted from this file. The *res* provides a multimedia file for the application to support Graphical User Interface rendering and the *resources.arsc* contains precompiled resources.

3.1 Brief overview of proposed work

Figure 2 depicts the proposed flow diagram, where *static analysis* has been used for Android malware detection. This technique has been used, because an efficient as well as a faster detection module has been proposed, which can be achieved by monitoring the code characteristics of an Android application. In contrast, to extract features from an Android application using the dynamic analysis, the application needs to execute, which takes significantly more time compared with static analysis. In *static analysis*, the very first step is to select a dataset consisting of uniform distribution of benign and malware Android applications. To extract features from an Android application, the application needs to first decompiled. There are various decompiler tools available like AXMLPrinter2, apktool, Androguard, etc. For the proposed work, *apktool* [25] was used for the decompilation.

Standard Permissions, Nonstandard Permissions and API-calls features were extracted from the decompiled Android applications. To select prominent features, four different FSTs were used as depicted in Section 3.3. These features were combined to prepare training files to train the classifiers.

3.2 Static analysis

In this analysis, the most popular features of an Android application were taken in to consideration such as permissions and API-calls, which were extracted from the *AndroidManifest.xml* and *classes.dex* file, respectively. Before extracting these features, firstly, the Android application needed to be decompiled using the *apktool* [25]. Permissions were extracted from the *AndroidManifest.xml* file under the <uses-permission> tag. The Fig. 3 describes it for a sample Android application file.

For the present work, standard and nonstandard permissions were segregated from permissions. Permissions, which start with android.permission keyword, are treated as standard permissions or otherwise as nonstandard permissions. Standard and nonstandard permissions segregated from Fig. 3 are mentioned as follows.

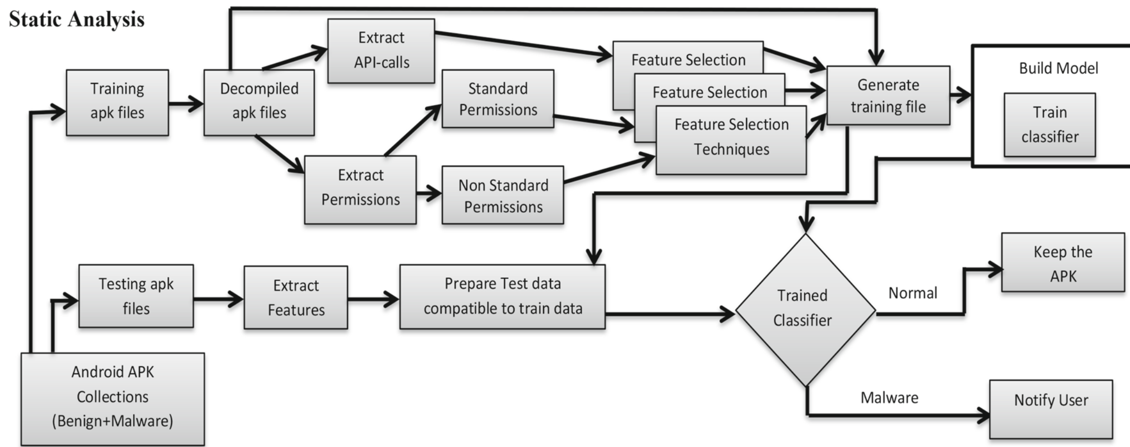


Fig. 2 Proposed operational flow diagram

Fig. 3 Permissions in AndroidManifest.xml file

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="com.appsbuilder537608.permission.C2D_MESSAGE"/>
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
<uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT"/>
```

Standard Permissions

INTERNET
ACEESS_NETWORK_STATE
ACEESS_FINE_LOCATION
ACEESS_COARSE_LOCATION

Nonstandard permissions

C2D_MESSAGE
RECEIVE
INSTALL_SHORTCUT

To extract API-calls, traverse each subdirectory of *smali* directory inside root package to get .smali files. From each .smali file, search for an *invoke* keyword under the Section .method and .end method. Figure 4 depicts the aforementioned approach to get API-calls. First part of API-calls represents class name, whereas second part represents method name.

From Fig. 4, we have extracted two API-calls, which are as follows.

android/app/Activity/getApplicationContext()
android/widget/RelativeLayout/init()

Following combinations of features have been considered to prepare training files.

1. API-calls + Standard Permissions.
2. API-calls + Nonstandard Permissions.
3. API-calls + Standard Permission and Nonstandard Permission.

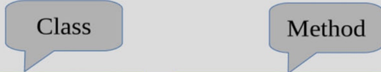
3.3 Feature selection technique

To achieve better results, prominent features should be selected using the FST. For the proposed work, four different feature selection techniques were used, which are briefly discussed as follows:

1. **BI-Normal Separation (BNS)** [15]
   - This technique models frequency of a feature in each document by using a random variable, which exceeds the hypothetical threshold. The widespread rate of the feature corresponds to the Area Under the Curve, which crosses the threshold. The classification of a feature for a particular class is decided by finding the distance between the threshold, i.e., if the feature is more commonly occurs in one class compared with other classes, then its threshold will be further from the tail of the curve.
   - BNS = $|F^{-1}(TPR) - F^{-1}(FPR)|$
     True Positive Rate (TPR) = $\frac{(tp)}{(tp+fn)}$

Fig. 4 Extraction of API-calls from .smali file

```
# direct methods
.method static constructor <clinit>()V
   .locals 2
   .prologue
   .line 90
    const/16 v1, 0xe
    if-lt v0, v1, :cond_0
    .line 91
    invoke-direct {v0}, Landroid/app/Activity;->getApplicationContext()V
    sput-object v0,
    invoke-direct {v0}, Landroid/widget/RelativeLayout;-><init>()
.end method
```



False Positive Rate (FPR) $= \frac{(fp)}{(tn+fp)}$
where F^{-1} is the inverse Normal distributions cumulative probability function.

- The BNS formula can be understand in expanded form as follows:

$$BNS\,(t, C_i) = \left| F^{-1}\left(\frac{N_{tC_i}}{N_{C_i}}\right) - F^{-1}\left(\frac{N_{t\bar{C}_i}}{N_{\bar{C}_i}}\right) \right| \quad (1)$$

where again F^{-1} is the inverse Normal distributions cumulative probability function.
Since this function explodes if $N_{tC_i}/N_{C_i} = 0$ or $N_{t\bar{C}_i}/N_{\bar{C}_i} = 0$ so we set minimum rate to 5/10000, and maximum rate to 1 - 5/10000.

2. **Mutual Information (MI)** [15,26].

- Following formula depicts mutual information:

$$\sum_{c\epsilon\{\bar{C}_i, C_i\}} \sum_{t} P(t,c) log\left(\frac{P(t,c)}{P(t)P(c)}\right) \quad (2)$$

- It gives the extent to, which an attribute t reduces the uncertainty in determining the appropriate class C. P(t,c) is the joint cumulative distribution function, P(t) and P(c) are the marginal probability for variables t and c respectively.

3. **Relevancy Score (RS)** [15,27].

- The RS formula for feature t w.r.t. class C_i is shown as follows.

$$RS\,(t, C_i) = log\left(\frac{P\,(t|C_i) + n}{P\,(\bar{t}|\bar{C}_i) + n}\right) \quad (3)$$

- This technique is based on the conditional probabilities of a feature that are considered in the training set. $P\,(t|C_i)$ represents the fraction of files in class C_i contains feature t, $P\,(\bar{t}|\bar{C}_i)$ represents the fraction of files in class C_i not containing feature t and n represents number of files in class C_i contains feature t.

4. **Kullback-Leibler (KL)** [15,28].

- The KL formula for feature t has been shown as follows.

$$KL\,(t) = -P\,(t|\bar{C}_i)\,log\left(\frac{P\,(t)}{P\,(t|\bar{C}_i)}\right) - P\,(t|C_i)\,log\left(\frac{P\,(t)}{P\,(t|C_i)}\right) \quad (4)$$

- $P\,(t|\bar{C}_i)$ represents the fraction of files not in class C_i contains feature t, $P\,(t|C_i)$ represents the fraction of files in class C_i contains feature t and P(t) is the probability of an feature t considering all class.

4 Performance evaluation

4.1 Datasets used and classifier

A dataset is mandatory to carry out the experimental work. For the malware, we have used Drebin dataset [29] and polymorphic and metamorphic malware dataset [30] containing 5600 and 10500 samples, respectively, were used. For benign, the third party applications were crawled to collect 8000 samples. From each category (benign and malware), 1500 samples were used for training the classifier.

Support Vector Machine is a supervised machine learning algorithm and it is widely used in malware detection [16,17,31]. It classifies the input test file based on the concept of decision planes that define decision boundaries. Support Vector Machine works very well even for unstructured and semi structured data like text, images and trees. One of the main strengths of the Support Vector Machine is kernel trick which can be used to solve complex classification task with an appropriate a kernel function. It scales relatively well to

Table 2 Confusion Matrix

Class Name	Prediction	
	Benign	Malware
Benign	True Positive (TP)	False Negative (FN)
Malware	False Positive (FP)	True Negative (TN)

high dimensional data and also less susceptible to overfitting. In this empirical study, Linear Support Vector Machine (L-SVM) [32] was used as a classifier and 10-fold cross-validation tests were used to measure the performance of the L-SVM.

- Training file was provided for training 3000 samples containing a uniform distribution of benign and malware Android applications.
- The training file was randomly partitioned into 10 equal sized subsamples. From these subsamples, 9 subsamples were used for training and one subsample for testing. This procedure was repeated for 10 rounds and each of the subsample was used only once for testing the trained model. Finally, the results from the 10-fold cross-validation tests were averaged to declare the final detection accuracy.

For the present work, Accuracy, Precision, and Recall were considered as the evaluation metrics as shown in Eq. 5. Confusion metrics values shown in Table 2 are used to compute the evaluation metrics.

$$\text{Recall} = \frac{TP}{TP + FN} \qquad \text{Precision} = \frac{TP}{TP + FP}$$
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

True Positive (TP) represents the benign samples that are correctly classified as benign. True Negative (TN) represents the malware samples that are correctly classified as malware. False Positive (FP) represents the malware samples that are misclassified as benign. False Negative (FN) represents the benign samples that are misclassified as malware.

4.2 Experimental results

Evaluation and results analysis of the proposed approach are discussed in this Section. Each of the chosen FSTs were evaluated with three different combinations of features 1) API-calls + Standard Permission (A_S), 2) API-calls + Nonstandard Permission (A_N), and 3) API-calls + Standard and Nonstandard Permission features (A_M). Experiments were conducted separately for the topmost features of count (FC) in the range of 50 to 500 and it was in increments of 50 as shown in Table 3. Finally, imperative features sets of sizes 50 to 500 of A_S, A_N, and A_M recommended by the each of the FSTs were considered to verify which feature set influence the L-SVM classifier to achieve highest accuracy. Table 3 depicts the performance achieved by the L-SVM classifier for the combined features.

The first set of experiments was conducted with most significant features of A_S, A_N, and A_M advised by the BNS FST. From Table 3 we can notice that the L-SVM achieved the highest accuracy of 99.6% for FC = 150, FC = 200, and FC = 450 features, respectively for suggested A_S feature of BNS FST. However, when the FC = 250, FC = 400, and FC = 500 we noticed the second highest accuracy of 99.5% was attained for the A_S features recommended by BNS FST. Further, for the same A_S features, the accuracy yielded i.e., 96.8%, 99.1%, 99.4%, and 99.4%, when FC = 50, FC = 100, FC = 300, and FC = 350, respectively, was not appreciable. Similarly, the same classifier performed pretty well on the A_N features that are suggested by BNS FST while yielding an accuracy of 99.6% for FC = 200, FC = 300, FC = 350, and FC = 500. Contrarily, when the FC set with 100, 400, and 450, there was competitive performance and L-SVM classifier attained the accuracy of 99.5%. The accuracy obtained was not appreciable when FC set with 50, 150, and 250 for A_N features, (see Table 3). Moreover, when the relevant A_M features of different FC are considered as suggested by BNS FST, the L-SVM recorded the highest accuracy of 99.6% for the FC = 300 features. We can notice that L-SVM classifier was equipotent and attained the accuracy of 99.5% when FC set with 200, 250, and 500 A_M features were considered. Meanwhile, the accuracy gained when FC = 50, FC = 100, FC = 150, FC = 350, FC = 400, and FC = 450 was not remarkable as depicted in Table 3.

The second sets of experiments were performed to examine the efficiency of L-SVM classifier for the best relevant A_S, A_N, and A_M features suggested by KL FST. Apparently, the highest accuracy of 98.8% was recorded for A_S features when FC = 450 are considered. In addition, the experiments were pursued and when FC = 500 L-SVM classifier obtained the second highest accuracy of 98.6% as shown in Table 3. Similarly the experiments were conducted for the different FC of 50, 100, 150, 200, 250, 300, 350, and 400 of A_S features. However, the accuracy accomplished was less when compared with the FC = 450 and FC = 500 of A_S features. Further, we could observe from the Table 3 that when the topmost prominent A_N features of different FC are taken into account, the maximum accuracy obtained by L-SVM classifier was 99.1% when FC = 400 and FC = 450. The second highest accuracy gained was 98.9% when FC = 350 and FC = 500. Later, we can see from the Table 3 that when FC = 250 the Lear-SVM classifier achieved the very less accuracy of 98.7%. Subsequently, the L-SVM classifier attained the minimum accuracy of 96.4%, 97.7%, 98.3%, and 98.5%, when FC = 50, FC = 100, FC = 150, and

Table 3 Accuracy achieved by the L-SVM classifier under 10-fold Cross-Validation Tests

FST	L-SVM Classifier BNS			KL			MI			RS		
FC	A_S	A_N	A_M	A_S	A_N	A_M	A_S	A_N	A_M	A_S	A_N	A_M
50	96.8	96.9	93.6	96.1	96.4	95.9	96.1	96.3	95.9	92.4	91.6	92.6
100	99.1	99.5	99.4	97.7	97.7	96.7	96.2	97.4	96.3	94.4	96.9	94.9
150	99.6	99.4	99.3	98.4	98.3	97.4	97.5	97.9	96.9	96.6	98.1	96.9
200	99.6	99.6	99.5	98.4	98.5	98.1	98.3	98.5	97.9	96.8	98.4	97.7
250	99.5	99.4	99.5	98.3	98.7	98.4	98.4	98.5	98.3	97.4	98.4	97.8
300	99.4	99.6	99.6	98.3	98.8	98.8	97.9	98.6	98.4	97.1	98.6	98.2
350	99.4	99.6	99.4	98.5	98.9	98.7	98.2	98.8	98.4	97.7	98.8	98.2
400	99.5	99.5	99.4	98.5	99.1	98.6	98.7	99.1	98.6	98.0	98.8	98.2
450	99.6	99.5	99.3	98.8	99.1	98.8	98.8	99.1	98.8	98.1	99.0	98.4
500	99.5	99.6	99.5	98.6	98.9	98.7	98.5	98.9	98.8	98.5	98.9	98.6

Table 4 Precision Achieved by the L-SVM classifier under 10-fold Cross-Validation Tests

FST	L-SVM Classifier BNS			KL			MI			RS		
FC	A_S	A_N	A_M	A_S	A_N	A_M	A_S	A_N	A_M	A_S	A_N	A_M
50	0.968	0.969	0.936	0.961	0.964	0.959	0.961	0.963	0.959	0.924	0.916	0.926
100	0.991	0.995	0.994	0.977	0.977	0.967	0.962	0.974	0.963	0.944	0.969	0.949
150	0.996	0.994	0.993	0.984	0.983	0.974	0.975	0.979	0.969	0.966	0.981	0.969
200	0.996	0.996	0.995	0.984	0.985	0.981	0.983	0.985	0.979	0.968	0.984	0.977
250	0.995	0.994	0.995	0.983	0.987	0.984	0.984	0.985	0.983	0.974	0.984	0.978
300	0.994	0.996	0.996	0.983	0.988	0.988	0.979	0.986	0.984	0.971	0.986	0.982
350	0.994	0.996	0.994	0.985	0.989	0.987	0.982	0.988	0.984	0.977	0.988	0.982
400	0.995	0.995	0.994	0.985	0.991	0.986	0.987	0.991	0.986	0.980	0.988	0.982
450	0.996	0.995	0.993	0.988	0.991	0.988	0.988	0.991	0.988	0.981	0.990	0.984
500	0.995	0.996	0.995	0.986	0.989	0.987	0.985	0.989	0.988	0.985	0.989	0.986

Table 5 Recall Achieved by the L-SVM classifier under 10-fold Cross-Validation Tests

FST	L-SVM Classifier BNS			KL			MI			RS		
FC	A_S	A_N	A_M	A_S	A_N	A_M	A_S	A_N	A_M	A_S	A_N	A_M
50	0.968	0.969	0.936	0.961	0.964	0.959	0.961	0.963	0.959	0.924	0.916	0.926
100	0.991	0.995	0.994	0.977	0.977	0.967	0.962	0.974	0.963	0.944	0.969	0.949
150	0.996	0.994	0.993	0.984	0.983	0.974	0.975	0.979	0.969	0.966	0.981	0.969
200	0.996	0.996	0.995	0.984	0.985	0.981	0.983	0.985	0.979	0.968	0.984	0.977
250	0.995	0.994	0.995	0.983	0.987	0.984	0.984	0.985	0.983	0.974	0.984	0.978
300	0.994	0.996	0.996	0.983	0.988	0.988	0.979	0.986	0.984	0.971	0.986	0.982
350	0.994	0.996	0.994	0.985	0.989	0.987	0.982	0.988	0.984	0.977	0.988	0.982
400	0.995	0.995	0.994	0.985	0.991	0.986	0.987	0.991	0.986	0.980	0.988	0.982
450	0.996	0.995	0.993	0.988	0.991	0.988	0.988	0.991	0.988	0.981	0.990	0.984
500	0.995	0.996	0.995	0.986	0.989	0.987	0.985	0.989	0.988	0.985	0.989	0.986

FC = 200, respectively. For the A_M features recommended by the KL FST, the L-SVM classifier yielded the highest accuracy of 98.8% with FC = 300 and FC = 450 features. Further, there was no significant improvement in the L-SVM classifier accuracy after FC was set with 50, 100, 150, 200, 250, 350, 400, and 500.

The accuracy obtained by the L-SVM classifier for different features (A_S, A_M, and A_N) recommended by

MI FST is provided in the Table 3. The L-SVM classifier achieved the greatest accuracy of 98.8% when FC = 450 for the topmost features of A_S, suggested by MI FST. When FC = 400 the accuracy attained was 98.7%. Additionally, the same classifier produced the least accuracy on other A_S features of different FC such as 50, 100, 150, 200, 250, 300, 350, and 500. When predominant A_N features of FC = 400 and FC = 450 are considered, L-SVM classifier was able to obtain maximum accuracy of 99.1%. However, in this case, also accuracy obtained by L-SVM classifier on other A_N features (for different FC) was not remarkable. Relatively, when A_M features were considered the L-SVM classifier outperformed by gaining the highest accuracy of 98.8% when FC = 450 and FC = 500, respectively. Consecutively, L-SVM classifier performed well for FC = 400 of A_M features recommended by MI FST and achieved the accuracy of 98.6%. In comparison, L-SVM classifier produced the least accuracy for various FC such as 50, 100, 150, 200, 250, 300, and 350 of A_M features, as depicted in Table 3.

Similar to the above experiments we examined the efficiency of the L-SVM classifier by considering the features such as A_S, A_N, and A_M recommended by RS FST in obtaining better accuracy. From the obtained experimental results, we could describe that the maximum stable accuracy achieved by the L-SVM classifier was significantly less compared to the greatest accuracy achieved by the L-SVM classifier for the A_S, A_N, and A_M features recommended BNS, KL, and MI FST as depicted in the Table 3.

From Table 3, we could reasonably infer that BNS FST is superior when compared to other three FSTs such KL, MI, and RS that are considered in this experimental work. As proof of concept, we achieved the highest detection accuracy of 99.6% was accomplished for the A_S features (when FC = 150, FC = 200, and FC = 450), A_N features (when FC = 200, FC = 300, and FC = 500), and A_M features (when FC = 300), respectively.

Tables 4 and 5 depict the Precision and Recall achieved by the L-SVM classifier, respectively. "Appendix A" shows computation of Precision and Recall.

5 Conclusion

In this work, an Android malware detection technique was implemented using static features, i.e., standard permissions, Nonstandard Permissions and API-calls. These features were extracted by using the *apktool*. After feature extraction, FSTs were used to select prominent features. Finally, the training files were built using a set of 3000 equal sized samples of benign and malware to check the performance of the proposed work.

Experimental results suggested that the combined features were a better choice for Android malware detection compared with individual features. For combined features, three possibilities of combinations were considered for this work, namely, the API-calls + Standard Permissions, API-calls + Nonstandard Permissions, and API-calls + Standard Permissions + Nonstandard Permissions. The highest detection accuracy of 99.6% was achieved by considering all the combined features using the BI-Normal Separation FST with the L-SVM classifier.

For future work, our whole concentration will be to detect newly evolving malwares by adding up them into our dataset.

Appendix A

See Tables 6 and 7.

Table 6 Confusion Matrix values

Class Name	No. of Files	Prediction	
		Benign	Malware
Benign	1500	1492 (TP)	8 (FN)
Malware	1500	6 (FP)	1494 (TN)

Table 7 Evaluation Metrics Values

Class Name	TP Rate	FP Rate	Precision	Recall
Benign	0.995	0.004	0.996	0.995
Malware	0.996	0.005	0.995	0.996
Weighted Average	0.995	0.005	0.995	0.995

Calculation of precision

$$\text{Precision Weighted Average} = \frac{X * XT + Y * YT}{XT + YT}$$

Where X = Benign Files Precision, XT = Total number of Benign Files, Y = Malware Files Precision, and YT = Total number of Malware Files.

$$\text{Precision Weighted Average} = \frac{0.996 * 1500 + 0.995 * 1500}{1500 + 1500}$$

$$\text{Precision Weighted Average} = 0.995$$

Calculation of recall

$$\text{Recall Weighted Average} = \frac{U * UT + V * VT}{UT + VT}$$

Where U = Benign Files Recall, UT = Total number of Benign Files, V = Malware Files Recall, and VT = Total number of Malware Files.

$$\text{Recall Weighted Average} = \frac{0.995 * 1500 + 0.996 * 1500}{1500 + 1500}$$

Recall Weighted Average = 0.995.

References

1. http://www.businessinsider.in/This-Chart-Shows-The-Massive-Pricing-Problem-Facing-Apples-iPhone-6/articleshow/39197536.cms. Accessed Oct 2016
2. https://techcrunch.com/2013/04/16/symantec-mobile-malware/. Accessed Nov 2016
3. http://www.darkreading.com/mobile/android-app-permission-in-google-play-contains-security-flaw/d/d-id/1328834. Accessed Jan 2017
4. https://www.eset.com/int/about/newsroom/research/fake-android-apps-bypass-google-play-store-security-installed-200000-times-in-a-month/. Accessed Jan 2017
5. Chuang, H.-Y., Wang, S.-D.: Machine learning based hybrid behavior models for Android malware analysis. In: IEEE International Conference on Software Quality, Reliability and Security, pp. 201–206 (2015). https://doi.org/10.1109/QRS.2015.37
6. Qin, Z., Xu, Y., Di, Y., Zhang, Q., Huang, J.: Android malware detection based on permission and behavior analysis. In: International Conference on Cyberspace Technology (CCT 2014), pp. 1–4 (2014). https://doi.org/10.1049/cp.2014.1352
7. Vinayakumar, R., Soman, K.P., Poornachandran, P.: Deep android malware detection and classification. In: Advances in Computing, Communications and Informatics (ICACCI 2017), pp. 1677–683 (2017)
8. Ariyapala, K., Do, H.G., Anh, H.N., Ng, W.K., Conti, M.: A host and network based intrusion detection for Android smartphones. In: 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 849–854 (2016). https://doi.org/10.1109/WAINA.2016.35
9. Sanz, B., Santos, I., Ugarte-Pedrero, X., Laorden, C., Nieves, J., Bringas, P.G.: Instance-based anomaly method for Android malware detection. In: International Conference on Security and Cryptography (SECRYPT 2013)
10. Apvrille, A., Strazzere, T.: Reducing the window of opportunity for Android malware Gotta catchem all. J. Comput. Virol. **8**, 61–71 (2012)
11. Ham, H.-S., Choi, M.-J.: Analysis of Android malware detection performance using machine learning classifiers. In: ICTC (2013)
12. Vinayakumar, R., Soman, K.P., Poornachandran, P., Sachin Kumar, S.: Detecting Android malware using long short-term memory (LSTM). J. Intell. Fuzzy Syst. **34**, 1277–1288 (2018)
13. Tong, F., Yan, Z.: A hybrid approach of mobile malware detection in Android. J. Parallel Distrib. Comput. **103**, 22–31 (2017). https://doi.org/10.1016/j.jpdc.2016.10.012
14. Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S.: Androdialysis: analysis of Android intent effectiveness in malware detection. Comput. Secur. **65**, 121–134 (2017). https://doi.org/10.1016/j.cose.2016.11.007
15. Aswini, A.M., Vinod, P.: Android malware analysis using ensemble features. In: International Conference on Security, Privacy and Applied Cryptographic Engineering (SPACE 2014), LNCS 8804, pp. 303–318 (2014)
16. Milosevic, N., Dehghantanha, A., Choo, K.-K.R.: Machine learning aided Android malware classification. Comput. Electr. Eng. **61**, 266–274 (2017). https://doi.org/10.1016/j.compeleceng.2017.02.013
17. Kim, H.-H., Choi, M.-J.: Linux kernel-based feature selection for Android malware detection. In: The 16th Asia-Pacific Network Operations and Management Symposium, pp. 1–4 (2014). https://doi.org/10.1109/APNOMS.2014.6996540
18. Xiaoyan, Z., Juan, F., Xiujuan, W.: Android malware detection based on permissions. In: International Conference on Information and Communications Technologies (ICT 2014), pp. 1–5 (2014). https://doi.org/10.1049/cp.2014.0605
19. Zhu, J., Wu, Z., Guan, Z., Chen, Z.: API sequences based malware detection for Android. In: IEEE 12th International Conference on Ubiquitous Intelligence and Computing and IEEE 12th International Conference on Automatic and Trusted Computing and IEEE 15th International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), pp. 673–676 (2015). https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.135
20. Peiravian, N., Zhu, X.: Machine learning for Android malware detection using permission and API calls. In: IEEE 25th International Conference on Tools with Artificial Intelligence (2013). https://doi.org/10.1109/ICTAI.2013.53
21. Chan, P.P.K., Song, W.-K.: Static detection of Android malware by using permissions and API calls. In: International Conference on Machine Learning and Cybernetics, vol. 1, 82–87 (2014). https://doi.org/10.1109/ICMLC.2014.7009096
22. Qiao, M., Sung, A.H., Liu, Q.: Merging permission and API features for Android malware detection. In: 5th International Congress on Advanced Applied Informatics (IIAI-AAI 2016), pp. 566–571 (2016). https://doi.org/10.1109/IIAI-AAI.2016.237
23. Su, M.-Y., Fung, K.-T., Huang, Y.-H., Kang, M.-Z., Chung, Y.-H.: Detection of Android malware: combined with static analysis and dynamic analysis. In: 2016 International Conference on High Performance Computing & Simulation (HPCS), pp. 1013–1018 (2016). https://doi.org/10.1109/HPCSim.2016.7568448
24. http://stackoverflow.com/questions/18717286/what-are-thecontents-of-an-android-apk-file. Accessed Feb 2017
25. APKTool. https://ibotpeaches.github.io/Apktool/. Accessed Sept 2016
26. Battiti, R.: Using mutual information for selecting features in supervised neural net learning. IEEE Trans. Neural Netw. **5**(4), 537–550 (1994)
27. Ling, X.F.: Feature selection. http://courses.washington.edu/ling572/winter2013/slides/class7featureselection.pdf. Accessed Sept 2016

28. Bonev, B.I.: Feature selection based on information theory. http://www.dccia.ua.es/~boyan/papers/TesisBoyan.pdf. Accessed Sept 2016
29. Drebin Dataset. https://www.sec.cs.tu-bs.de/~danarp/drebin/. Accessed Oct 2016
30. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: an extended insight into the obfuscation effects on Android malware. Comput. Secur. **51**, 16–31 (2015). https://doi.org/10.1016/j.cose.2015.02.007
31. Shahzad, F., Shahzad, M., Farooq, M.: In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS. Inf. Sci. **231**, 45–63 (2013). https://doi.org/10.1016/j.ins.2011.09.016
32. Hearst, M.A., Dumais, S.T., Osuna, E., Platt, J., Scholkopf, B.: Support vector machines. IEEE Intell. Syst. Appl. **13**(4), 18–28 (1998)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.