DROIDSWAN: DETECTING MALICIOUS ANDROID APPLICATIONS BASED ON STATIC FEATURE ANALYSIS

Babu Rajesh V, Phaninder Reddy, Himanshu P and Mahesh U Patil

Centre for Development of Advanced Computing
cdac.in

ABSTRACT

Android being a widely used mobile platform has witnessed an increase in the number of malicious samples on its market place. The availability of multiple sources for downloading applications has also contributed to users falling prey to malicious applications. Classification of an Android application as malicious or benign remains a challenge as malicious applications maneuver to pose themselves as benign. This paper presents an approach which extracts various features from Android Application Package file (APK) using static analysis and subsequently classifies using machine learning techniques. The contribution of this work includes deriving, extracting and analyzing crucial features of Android applications that aid in efficient classification. The analysis is carried out using various machine learning algorithms with both weighted and non-weighted approaches. It was observed that weighted approach depicts higher detection rates using fewer features. Random Forest algorithm exhibited high detection rate and shows the least false positive rate.

KEYWORDS

Mobile Security, Malware, Static Analysis, Machine Learning, Android

1. INTRODUCTION

Android is a widely used mobile platform and due to it's dominance in consumer space, Android becomes a lucrative target for malware developers who are exploiting the popularity and openness of Android platform for various benefits. Malware developers use Android marketplaces as entry points for hosting thesir malicious applications into the android user space. According to RiskIQ [1] report, malicious applications in Play store have grown by 388 percent from 2011 to 2013, while the number of such applications removed annually by Google has dropped from 60 percent in 2011 to 23 percent in 2013. As a large number of applications are uploaded and updated regularly on these market places, Manual analysis of all the applications is difficult task. Scarcity of effective mechanisms to detect these malicious samples has fueled the rise of malware applications on Android market places. In this regard we present DroidSwan, a system for classifying applications as malware or benign, based on static analysis of Android APK. DroidSwan extracts various crucial features from an Android application, assigns weight to these features and builds a classifier model using machine learning algorithms. The classifier model is trained using the malware data set of 1260 malware acquired from Genome Malware

David C. Wyld et al. (Eds) : ACITY, DPPR, VLSI, WiMNET, AIAA, CNDC - 2015
pp. 163–178, 2015. © CS & IT-CSCP 2015 DOI : 10.5121/csit.2015.51315

Project [2] and popular benign applications obtained from Google Play Store. The model was then tested against 500 malware samples obtained from Virustotal malware intelligence service [3].

Android applications are installed on to a device using an Android application package (APK) file. In our analysis, This APK file is disassembled for extraction of necessary features which form feature set to be used during classification. Figure 1 depicts how features are extracted from an APK. The application resources such as XML layout-definition files and images are stored in the 'res' directory which the malware writers use to inject malicious binaries like '.sh', '.elf' or '.exe' inside the images or other resource files used by the application. In most of the cases, these malicious binaries are found embedded within image files (.jpg and .png) used by the application. The AndroidManifest.xml file contains the name, version number and access rights of the APK. AndroidManifest.xml is a binary XML file. ApkParser [4] can be used to convert this binary XML file into a readable XML file. A malware application may hide some of the permission it uses by not declaring them in the Manifest file.

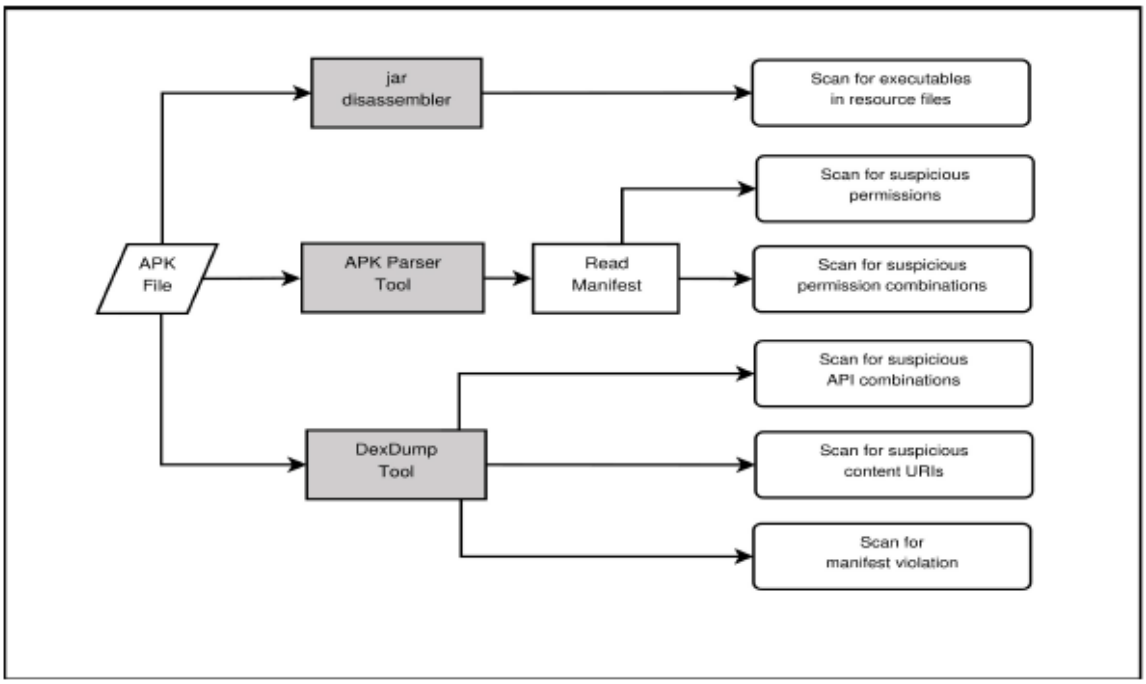

Figure 1. Feature extraction in DroidSwan

2. RELATED WORK

Androguard [5] statically extracts features from APK, but this tool shows high false positive rate as 80 out of 100 popular benign samples analyzed were assigned high androrisk score. Aubrey-Derrick Schmidt et al. [6] extracted function calls of an installed application using readelf command. These function calls were later compared with function calls of the malware executables present on a Remote Detection Server. In contrast to this, our approach does not analyze applications on an Android device because of limited resources like power, memory and data usage, but if needed it can be ported onto a mobile device. DroidRanger [7] detects

malicious applications of known malware families in popular Android marketplaces using permission-based behavioral foot printing. To detect malware from unknown families, DroidRanger uses heuristic-based filtering scheme. The drawback of DroidRanger is the requirement of manual operations while analyzing and collecting behavior of applications.

DroidMat [8] combines static and dynamic analysis approaches. It extracts features like permissions and intents using static analysis and API calls using dynamic analysis. In contrast to this, we perform static analysis to extract all the necessary features as a tool based on static analysis can be deployed on a gateway device with greater ease as compared to tool based on dynamic analysis. Adrieene et al. [9] proposed an approach to identify over privileged applications by comparing API calls invoked with permissions declared in the Manifest. William Enck et al. [10] proposed an approach where a certificate is generated during an application's installation. This certificate gives complete information about the application by rating them using Kirin security rules which are based on the combinations of permissions extracted from Manifest file. DroidAnalytics [11] is a signature based system for detecting repackaged applications. The drawback of this technique is it requires large and balanced data set of malware and benign samples. Shabtai et al. [12] applied machine learning classifier techniques like decision tree, Naive Bayes (NB), Bayesian Networks (BN) etc. to classify Android applications as games and utilities citing the non availability of malware applications.

They collected around 22,000 features initially and later reduced to 50 features for the purpose of classification. Our approach uses 24 features for classification.

3. APPROACH

This section explains our approach. The following subsections describe feature selection for feature set, weight assignment to the features, selection of feature vector and finally the working of DroidSwan.

3.1. Features

3.1.1. Suspicious Permissions and Permission Combinations

A permission is a restriction limiting the access of an application to the device to protect critical data and code that could be misused to distort or damage the user experience. We considered the patterns of suspicious permissions in malware samples as discovered by Y.Zhou et.al. [13]. For extracting permissions used by an application we use APKParser tool. The permissions extracted were analyzed and cross verified for high occurrence across malware samples available in our training dataset. Out of all the permissions specified as suspicious by Y.Zhou et.al, we discarded those permissions which were present in large numbers in benign samples as these would not significantly contribute during classification process. The presence or absence of the remaining suspicious permissions was then considered as a feature. Our findings are shown in Figure 2.

I.Rassameeroj [14] states that certain permission combinations enable an application to perform dangerous actions posing threat to user's data and privacy. We considered these combinations as features for our feature set. Table 1 depicts the permissions and permission combinations considered as features.

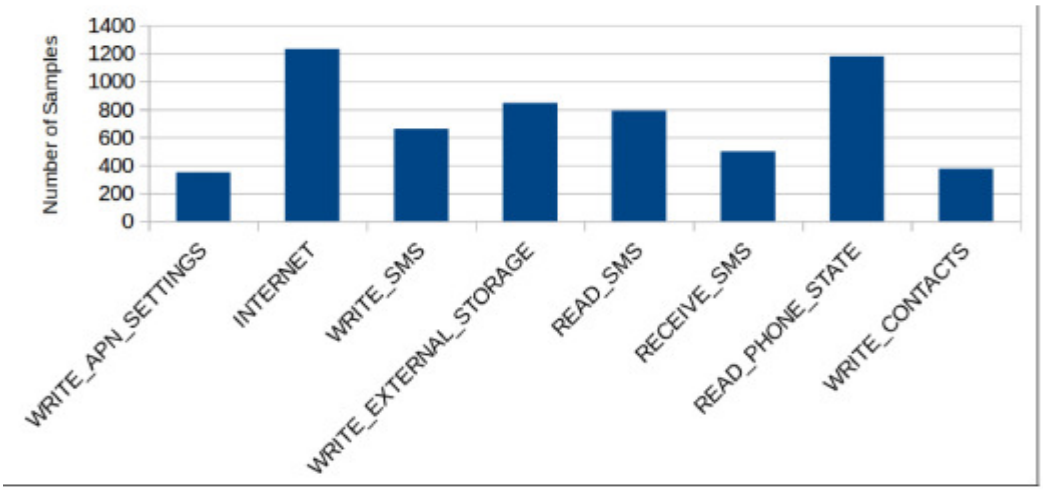

Figure 2. Frequency of suspicious permissions among malware samples

3.1.2. Suspicious API Combinations

APIs used by an application determines the actual functionality and capability of the application. Static analysis of APIs used in an application hence becomes important to understand what the application actually intends to do. In the similar direction of selecting permissions as features, our approach contributes by evaluating APIs extensively used by malware applications. APIs were broadly classified according to their usage by the application. From the list of APIs which are found in large number of malware samples, combinations were derived which could pose a threat to the user. Two main types of threats considered are financial losses and leakage of user's personal information. For example APIs for accessing user's personal information (network details, device ID, line number, etc.) in combination with APIs for sending SMS enables an application to transmit user's personal information to a predefined source. This leads to both breach of privacy as well as monetary loss. The monetary loss here is due to cost incurred when the SMS is sent.\par APIs for evaluation are extracted by disassembling classes.dex file using dexdump tool present in Android SDK [15]. Figure 3 depicts the a snapshot of classes.dex when disassembled using dexdump tool. Table 2 lists the API combinations considered as a feature for our feature set.

```
8582 018370: 0c09                                    |0010: move-result-object v9
8583 018372: 7100 d800 0008                          |0011: invoke-static {}, Landroid/telephony/SmsManager;.getDefault:()Landroid/telephony/
     SmsManager; // method@00d8
8584 018378: 0c0c                                    |0014: move-result-object v12
8585 01837a: 7210 5b00 0908                          |0015: invoke-interface {v9}, Landroid/database/Cursor;.getCount:()I // method@005b
8586 018380: 0a01                                    |0018: move-result v1
8587 018382: 3d01 0800                               |0019: if-lez v1, 0021 // +0008
8588 018386: 7210 5d00 0908                          |001b: invoke-interface {v9}, Landroid/database/Cursor;.moveToNext:()Z // method@005d
8589 01838c: 0a01                                    |001e: move-result v1
8590 01838e: 3901 0300                               |001f: if-nez v1, 0022 // +0003
8591 018392: 0e00                                    |0021: return-void
8592 018394: 1a01 b007                               |0022: const-string v1, "_id" // string@07b0
8593 018398: 7220 5a00 1908                          |0024: invoke-interface {v9, v1}, Landroid/database/Cursor;.getColumnIndex:(Ljava/lang/
     String;)I // method@085a
8594 01839e: 0a01                                    |0027: move-result v1
8595 0183a0: 7220 5c00 1908                          |0028: invoke-interface {v9, v1}, Landroid/database/Cursor;.getString:(I)Ljava/lang/
     String; // method@005c
8596 0183a6: 0c0a                                    |002b: move-result-object v10
8597 0183a8: 1a01 c80b                               |002c: const-string v1, "has_phone_number" // string@0bc8
8598 0183ac: 7220 5a00 1908                          |002e: invoke-interface {v9, v1}, Landroid/database/Cursor;.getColumnIndex:(Ljava/lang/
     String;)I // method@085a
```

Figure 3. Disassembled dex file

3.1.3. Manifest Violation

All the permissions required by an application should be declared in the AndroidManifest.xml. These permissions determine what are all the capabilities the application has. During application installation, all the permissions declared by the application are not cross verified by the package manager. Thus, at the run time if the application needs to perform a certain action and it does not have corresponding permission, run time exceptions occur. Malware developers take advantage of this flaw to perform collusion attacks [16]. The collusion attack requires at least 2 applications to work in collaboration. In this type of attack, an over privileged application provides an under privileged application with necessary permissions at runtime. Soundcomber [17] is one such application which aims at collecting user's information by capturing audio from device's microphone and then sends it over the network with help of another application having necessary permissions. Figure 4 depicts a scenario where two applications combine their permissions to read contacts and send them over the network.

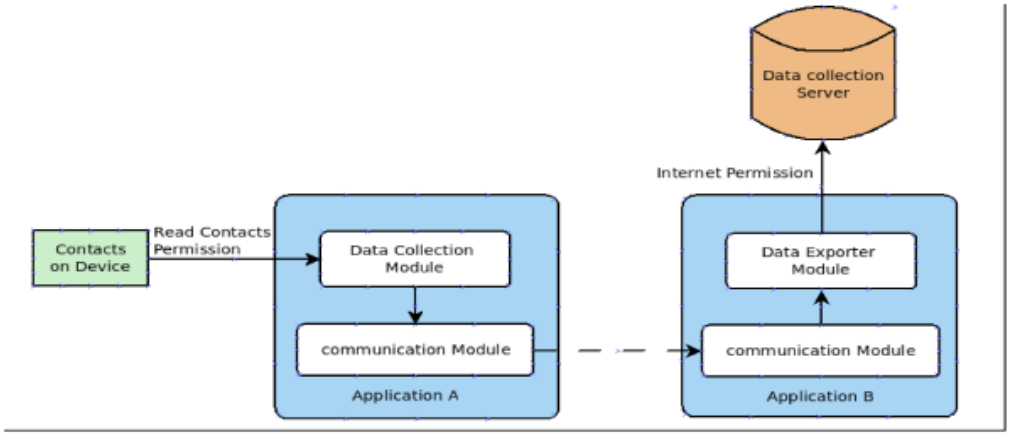

Figure 4. A collusion attack scenario

One way to detect the possibility of collusion attack is to look for application which has declared more permissions than what it requires (over privileged applications), but the drawback with this approach is the high false positive rate. The reason for high false positive rate is that many developers declare majority of the permissions available irrespective of their usage by the application.

We devised a different approach for detecting possible collusion attack. Rather than looking for over privileged applications we detect under privileged applications, that is the application declaring less permissions than what it actually required. The under privileged application then gets required privileges at runtime with the help of another application. To detect under privileged applications applications, we look for the permissions that will be used by the application at run time but are not present in application's manifest file. To derive permissions required by application at run time, permission required for executing each API present in application's dex file is extracted. If any permission required for execution of an API is not found in the application's manifest file, it is considered as a manifest violation.

We derive the permissions required by an API with the help of Android's developer guide and Pscout [18].

Each occurrence of manifest violation is assigned a weight of 7. A summation of these permission's weights was considered as the weight of the feature (Manifest violation).

3.1.4. Suspicious Content URI

A content URI (used for data access) can be called suspicious if by using that URI an application can leak user's personal data or can access another application's data. For example, an application can get access to contacts by using URI: content://com.Android.contacts. Such suspicious URIs were identified and their presence was checked among various malware and benign samples available in the training set. Suspicious content URIs which were detected in most of the malware samples and few benign samples were considered as a feature for feature set. Figure 5 shows the content URIs extensively used by malware applications.

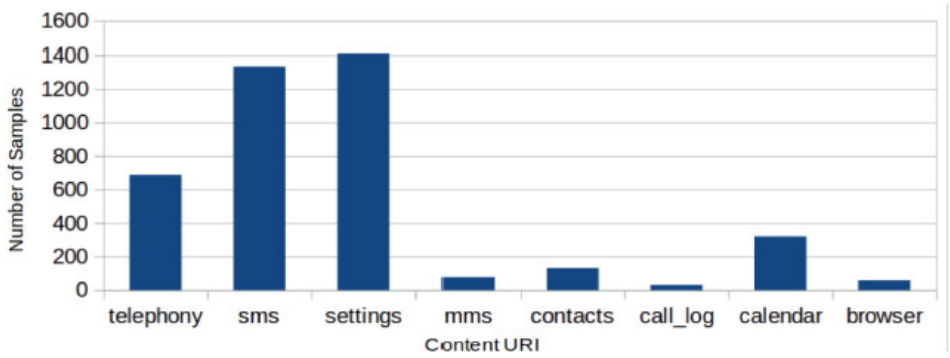

Figure 5. Frequency of suspicious content URIs among malware samples

To collect the content URIs used by the application, we parse the dalvik bytecode of disassembled classes.dex. The presence of content URIs that provide access to MMS, Browser and telephony data were seen among majority of malware applications.

Each Suspicious Content URI was assigned a weight of 6. Summation of the weights for frequency of such suspicious content URIs is considered as the weight of the feature.

3.1.5. Detection of Executable code

Embedding malicious code into documents has been successful technique for distributing malware. Desktop malware like Pidief, ZBOT, SillyD have been distributed as malicious PDF, JPEG, mp3 files. Based on Shafiq [19] and Stolfo's [20] findings which stated that detection of embedded malware requires parsing the bytecode of the documents, We employed a mechanism to find embedded executables by parsing the bytecode of all the files present in the resources directory of an APK. Many malware samples show the presence of executables and shell scripts embedded within image and music files. Presence of image files embedded with executable code can be found in samples from malware families like DroidKungFu1 and RougePush. Malware samples from DroidKungFu3 and GingerMaster families show presence of music files embedded with executable code.\par As this behavior was detected only in malware samples, presence of embedded executables was assigned a maximum weight of 10. Summation of the weights for frequency of such files is considered as weight of the feature.

3.2. Assigning Weight to Features

The weight assigned to a feature represents the impact that presence or absence feature makes on an application's classification. Weights are assigned to each feature on a scale of 1 to 10 using heuristics based approach such that higher the weight of a feature, more the feature contributes during classification. The highest weight of 10 was assigned to presence of executables embedded in image or music files. Presence of embedded executables is the strongest indicator in our feature set of an application being malicious as only malware samples are found to have resource files injected with executable code. All other features were assigned weights relative to the weight of 'presence of embedded executables' feature. Manifest violations are assigned a weight of 7. This is because unlike a malicious application, a benign application declares all the permissions being used. When compared to 'suspicious Permission combinations' or 'suspicious API combinations', 'manifest violation' has more impact during classification but it is not as influential as 'presence of embedded executables'. Thus it is assigned a weight lower than 'presence of embedded executables' and higher than 'suspicious Permission combinations' and 'suspicious API combinations'. Presence of suspicious content URI in an application is assigned a weight of 6. The presence of these content URI was seen in both malicious and benign samples, but number of malicious samples containing these URIs was much greater than number of benign samples. Weights for suspicious content URIs, manifest violations, presence of executable code are frequency based. Thus the total weight for these features in the feature set is multiple of the frequency of the feature occurrence and the weight assigned to the feature. \par Permission combinations and API combinations are assigned a moderate weight of 5 as the presence of these leads to suspicious behaviors, but their presence cannot conclude an application of being a malware or benign. We assigned suspicious permissions the lowest weight of 3 as these permissions can be found in large number in both benign and malware samples. Table 3 depicts the assignment of weights to the features selected.

3.3. Feature Vector Selection

After deciding upon the application's attributes to be considered as features, we considered and evaluated three categories of feature vectors with a set of machine learning algorithms. All the three categories of feature vectors constituted of similar features, but represented in different way. The first and second categories of feature vectors were weighted feature vector where as the third category was a non weighted feature vector. The first category of feature vector contained weights for each feature along with the Euclidean distance as an additional feature. The second category of feature vector was derived by excluding Euclidean distance from the first feature vector. For the third category of feature vector, rather than considering the frequency and weight of a feature, we check only presence of a feature. Representation in feature vector is done as either 1 or 0 to depict the presence or absence of a specific feature in the sample.

3.3.1. Evaluation of model for Feature Vector Selection

K-fold cross validation was carried out in order to evaluate the efficiency of the classification model. The default implementation of cross validation provided by WEKA was used for this purpose. The efficiency of the classifier models generated using all three categories of feature vectors were compared based on cross validation. One round of cross-validation of a two class classifier model involves segregating a sample of the training data set into two complementary

subsets, subset for performing the analysis (the training set) and subset for validating the analysis (the validation set). Inconsistency is reduced by multiple rounds of cross-validation using different segregations. Finally the average of all validation results is presented as true positive rate and false positive rate. We used WEKA [21] implementation for both model generation and cross validation. The true positive rate and false positive rate are deduced as follows :

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

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

Figure 6 (a) and Figure 6 (b) show variations in true positive rates and variations in false positive rates respectively for models generated using three categories of feature vectors.

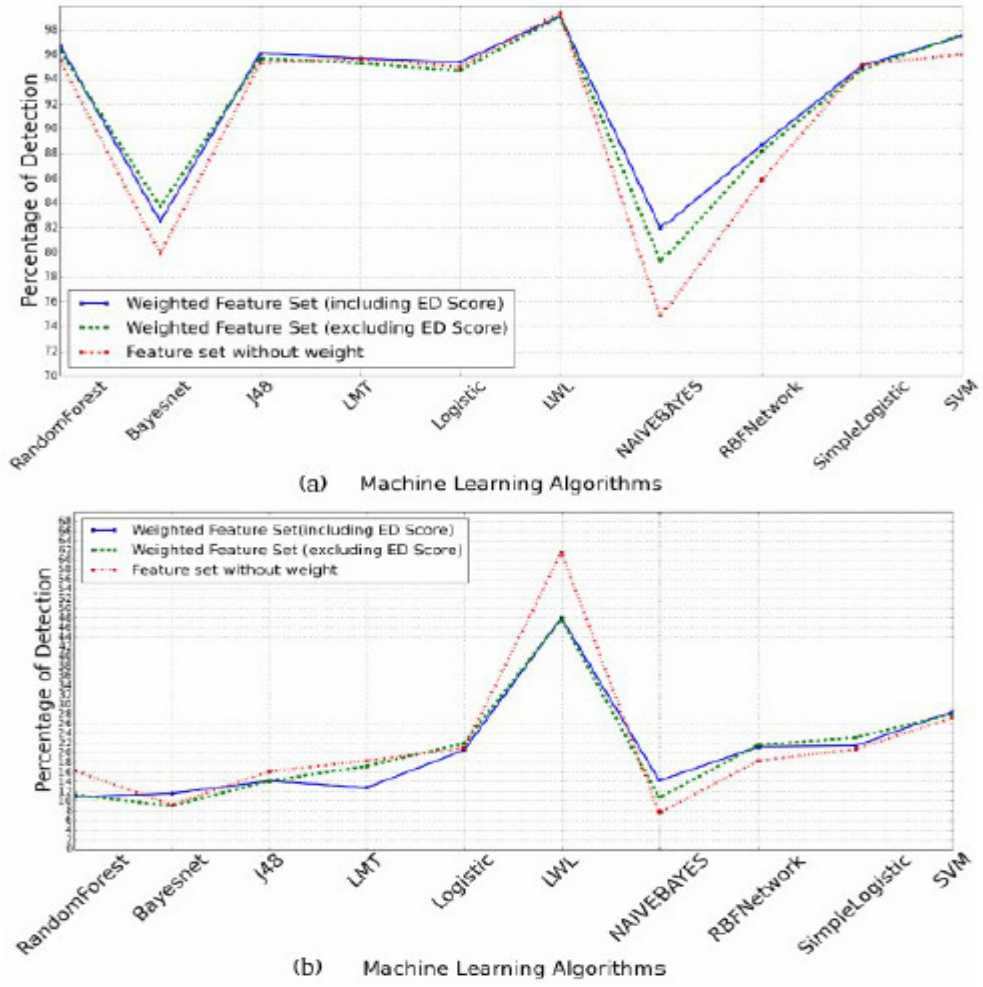

Figure 6. Variation inTPR (a) and FPR (b) for various models

High true positive and low false positive rates are observed for the second category of feature vector, that is a feature vector with weights and excluding Euclidean distance. Thus the second category of feature vector was considered for providing features to the machine learning algorithms. The reason for omitting Euclidean distance from the feature set was its last rank among the features on applying Chi-Square attribute ranking mechanism. This illustrated that excluding it as a feature would not affect the detection rates. Figure 7 shows variation in Euclidean distance across all the samples present in our dataset.

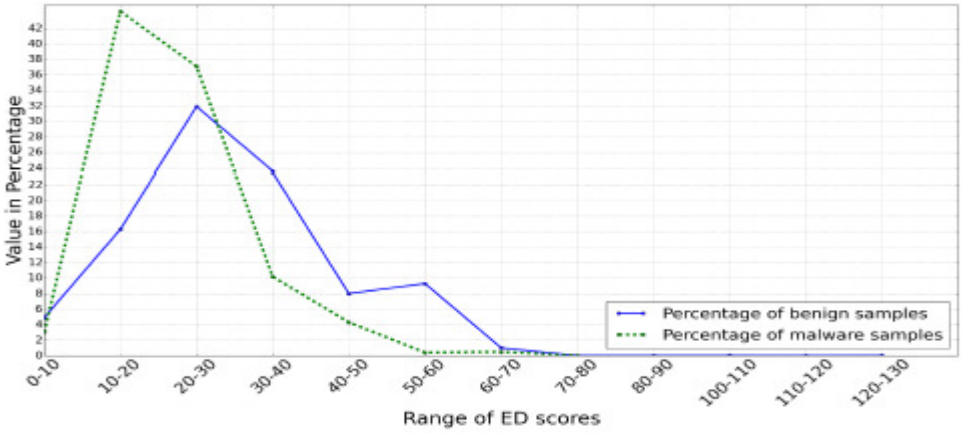

Figure 7. Variation in ED scores among benign and malware samples

Figure 8 shows the receiver operating characteristic (ROC) graph for the classification model built using second category of feature set. This graph illustrates the performance of a binary classifier system built using various machine learning algorithms and the weighted feature set. Random Forest algorithm depicts the maximum ROC space in the ROC curve which proves that for the given training set, classifier model built using Random Forest is more efficient than models generated using other machine learning algorithms. We used model built using Random Forest algorithm as the classifier in DroidSwan implementation.

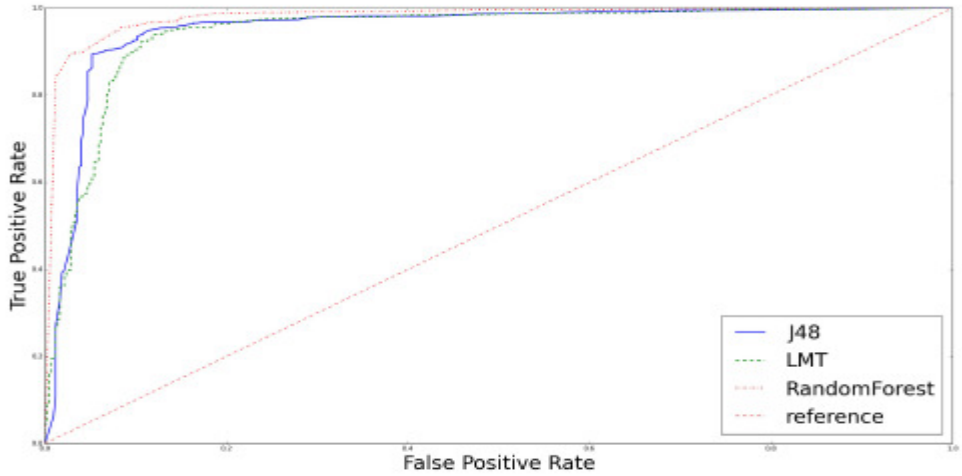

Figure 8. ROC Curve for classifier models based on various algorithms

3.4. DroidSwan working

Working of DroidSwan is carried out in two phases. Phase1 is the knowledge building phase. In this phase, DroidSwan extracts specific features and builds feature set of all the samples from the training set. These feature sets are then provided to the machine learning algorithm using WEKA implementation of machine learning algorithms. A two class classifier model is thus generated. \par Classifier model generated during phase 1 can be used for classification of samples without updating the model every time a new sample is provided for analysis.

Phase2 is the classification phase. In this phase, features are extracted from test application which needs to be classified and a corresponding feature set is built. Now this feature set is provided to the classification model generated during phase 1. The classification model then classifies the sample as either malicious or benign and generates a Json report for the same. This output report contains details regarding the presence or absence of all the features under consideration. The report also specifies all the suspicious content URIs and embedded executables present in the application. Figure 9 depicts DroidSwan's architecture.

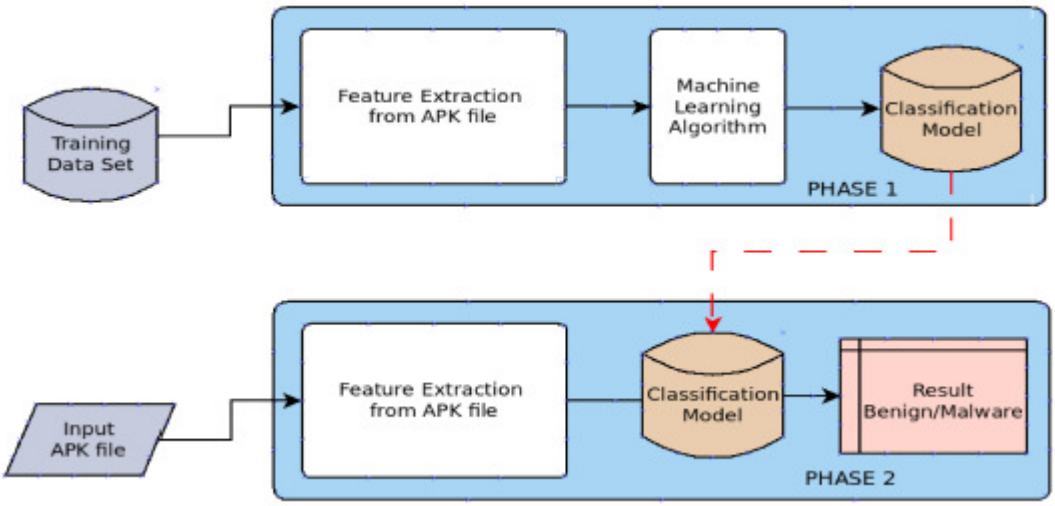

Figure 9. DroidSwan architecture

4. RESULTS

The efficiency of DrooidSwan classification model was tested by analyzing 500 malware samples obtained from Virustotal malware intelligence service \cite{hispasec2011virustotal} and 800 benign samples from ApkDrawer \cite{apkDrawer}. Collectively these samples constituted of our test-set. It was verified beforehand that the test-set does not contain any samples in common with the training-set by comparing the hashcode of each sample in test set against hashcodes of samples from training set. Figure 10 and Figure 11 depict the detection rates of malware samples and benign samples respectively by using DroidSwan.

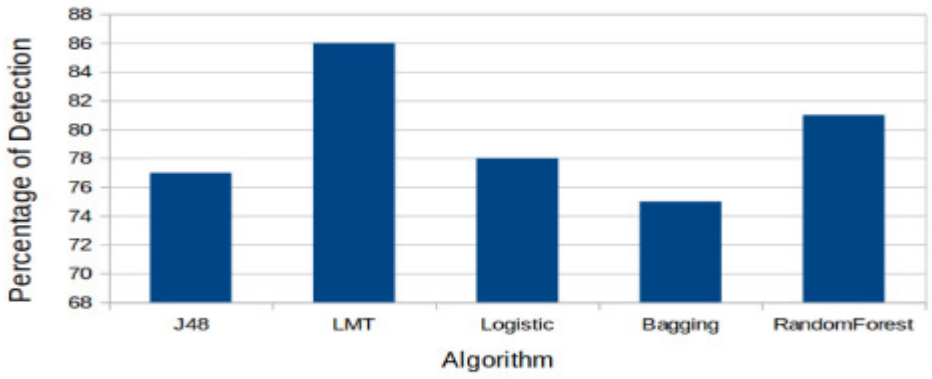

Figure 10. Detection rate of DroiidSwan for malware samples

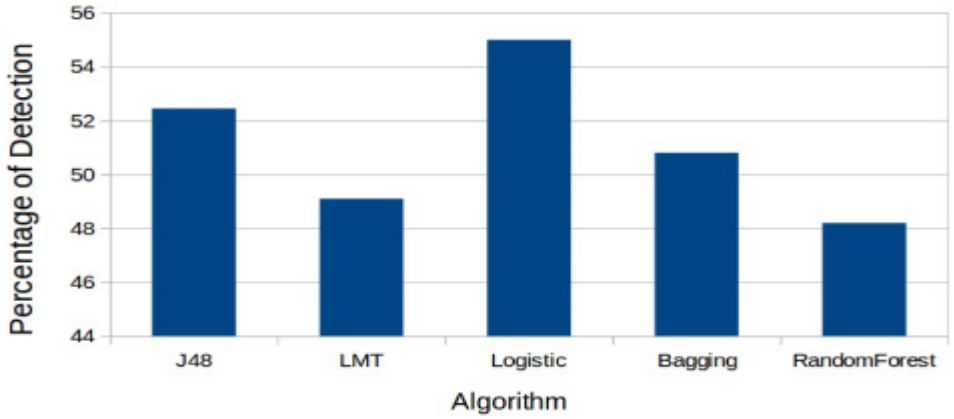

Figure 11. Detection rate of DroiidSwan for benign samples

The detection rate of DroidSwan was compared with the detection rates of four other antivirus solutions for the same set of malware samples. Figure 12 shows the detection rate of DroidSwan in comparison with Kaspersky (version 12.0.0.1225) [23], McAfee (version 6.0.5.614) [24], Avast (version 8.0.1489.320) [25] and TrendMicro (version 9.740.0.1012) [26].

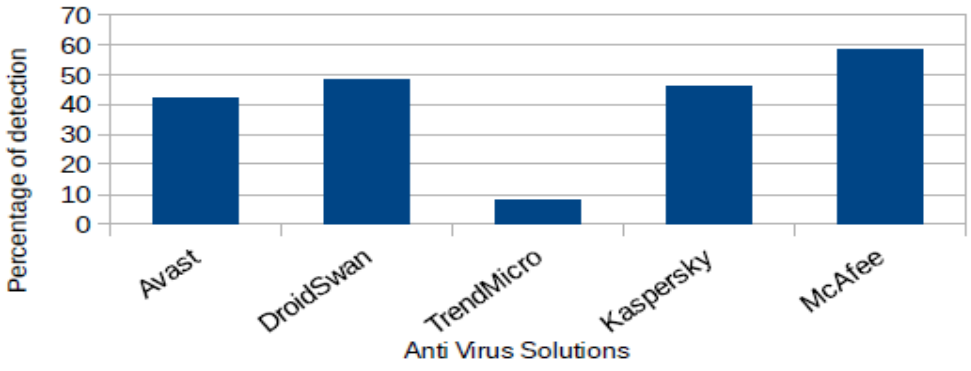

Figure 12. Detection rates of DroiidSwan in comparison with other AV solutions

Recall rate of DroidSwan with Random Forest based classifier for malwares from various malware families is shown in Figure 13.

5. CONCLUSION

We present DroidSwan, an approach for detecting malicious Android applications wholly based on static analysis of their respective APK files. The process of classification comprises of extracting 24 features, assigning weights to the features and finally using the collection of feature weights as a feature set. The feature set along with Random Forest classifier model is then used to classify the given sample as either malware or benign.\par We observed that classifier model built using Random Forest shows higher TPR and lower FPR when compared to other machine learning algorithms.

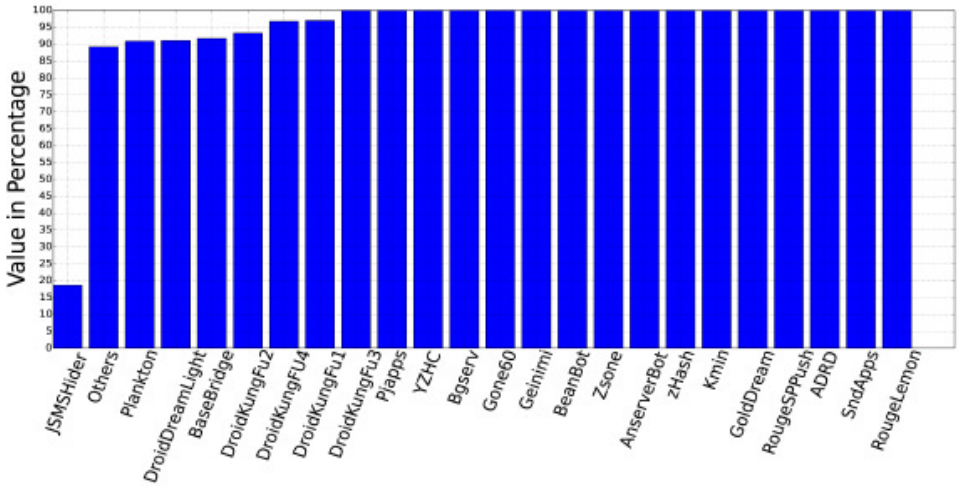

Figure 13. Recall rate of DroidSwan for various malware families

REFERENCES

[1] RiskIQ, Feb 19 2014, Research Also Shows Steady and Significant Drop in Number of Malicious Apps Being Removed in Past Three Years. Available: http://www.riskiq.com/company/press-releases/riskiqreports-malicious-mobile-apps-google-play-have-spiked-nearly-400

[2] Genome Project. Android malware samples. http://www.malgenomeproject.org.

[3] S. Hispasec Sistemas. Virustotal malware intelligence service, 2011.

[4] J. Erdfelt. Apkparser tool. https://code.google.com/p/xml-apk-parser.

[5] A. Desnos. Androguard. Available at https://code.google.com/p/androguard/.

[6] Schmidt, A-D., Rainer Bye, H-G. Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. "Static analysis of executables for collaborative malware detection on android." In Communications, 2009. ICC'09. IEEE International Conference on, pp. 1-5. IEEE, 2009.

[7] Zhou, Yajin, Zhi Wang, Wu Zhou, and Xuxian Jiang. "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets." In NDSS. 2012.

[8] Wu, Dong-Jie, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and KuoPing Wu. "Droidmat: Android malware detection through manifest and API calls tracing."In Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on, pp. 62-69. IEEE, 2012.

[9] Felt, Adrienne Porter, et al. ”Android permissions demystified.”Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011.
[10] Enck William, Machigar Ongtang, and Patrick McDaniel. ”On lightweight mobile phone application certification.” Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.
[11] Zheng, Min, Mingshen Sun, and John Lui. ”Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware.” Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on IEEE, 2013.
[12] Shabtai, Asaf, Yuval Fledel, and Yuval Elovici. "Automated static code analysis for classifying Android applications using machine learning." Computational Intelligence and Security (CIS), 2010 International Conference on. IEEE, 2010.
[13] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.
[14] Rassameeroj, Ittipon, and Yuzuru Tanahashi. "Various approaches in analyzing Android applications with its permission-based security models." Electro/Information Technology (EIT), 2011 IEEE International Conference on. IEEE, 2011.
[15] Google Inc. Official Page for android developers. http://developer.android.com.
[16] Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android." In NDSS. 2012.
[17] Schlegel, Roman and Zhang, Kehuan and Zhou, Xiao-yong and Intwala, Mehool and Kapadia, Apu and Wang, XiaoFeng. 'Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.'NDSS, 2011
[18] Au, Kathy Wain Yee, Yi Fan Zhou, Zhen Huang, and David Lie. "Pscout: analyzing the android permission specification." In Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217-228. ACM, 2012.
[19] Shafiq, M. Zubair, Syed Ali Khayam, and Muddassar Farooq. "Embedded malware detection using markov n-grams." In Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 88-107. Springer Berlin Heidelberg, 2008.
[20] Stolfo, Salvatore J., Ke Wang, and Wei-Jen Li. ”Towards stealthy malware detection.” Malware Detection. Springer US, 2007. 231-249.
[21] Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. ”The WEKA data mining software: an update.” ACM SIGKDD explorations newsletter 11, no. 1 (2009): 10-18.
[22] Z. Jay. Apkdrawer.com. http://www.apkdrawer.com.
[23] Kaspersky mobile security. Available at http://www.kaspersky.co.in/downloads/android-security.
[24] Mcafee mobile security. Available at https://www.mcafeemobilesecurity.com/.
[25] Avast mobile security. Available at http://www.avast.com/en-in/free-mobile-security.
[26] Trendmicro mobile security. Available at http://www.trendmicro.com/us/enterprise/product-security/mobile-security/.

APPENDIX

Table 1. Suspicious permissions and permission combinations

Suspicious permissions and permission combinations	Weight assigned
READ SMS	3
WRITE SMS	3
RECEIVE SMS	3
WRITE CONTACTS	3
WRITE APN SETTINGS	3
SEND SMS	3
ONLY INTERNET	3
ONLY WRITE EXTERNAL STORAGE	3
WRITE SMS and RECEIVE SMS	5
SEND SMS and WRITE SMS	5
INTERNET and WRITE EXTERNAL STORAGE	5
INTERNET,RECORD AUDIO, READ PHONE STATEand MODIFY PHONE STATE	5
ACCESS FINE LOCATION or ACCESS COARSE LOCATION, RECEIVE BOOT COMPLETED and INTERNET	5
INTERNET,RECORD AUDIO and PROCESS OUTGOING CALLS	5

Table 2. Suspicious API combinations

Suspicious API combinations	Weight assigned
"landroid/telephony/telephonymanager;.getdeviceid" "landroid/location/locationmanager;.getlastknownlocation" "landroid/location/location;.getlatitude" "landroid/location/location;.getlongitude" "landroid/telephony/smsmanager;.sendtextmessage" "landroid/net/uri;.parse","landroid/location/locationmanager;.getbestprovider"	5
"ljava/net/urlencoder;.encode" "ljava/net/uri;.getquery" "ljava/net/httpurlconnection;.connect" "ljava/net/httpurlconnection;.geturl" "ljava/net/httpurlconnection;.getheaderfield" "landroid/location/locationmanager;.getbestprovider" "landroid/location/location;.getlatitude" "landroid/location/location;.getlongitude" "landroid/telephony/gsm/smsmanager;.sendtextmessage"	5
"landroid/net/uri;.parse" "landroid/content/contentresolver;.query" "landroid/database/cursor;.movetonext" "landroid/database/cursor;.getcolumnindex"	5

"landroid/database/cursor;.getstring" "landroid/database/cursor;.close" "landroid/database/cursor;.movetolast" "landroid/database/cursor;.movetoprevious"	
"landroid/net/uri;.parse" "ljava/net/urlencoder;.encode" "ljava/net/url;.openstream" "landroid/telephony/telephonymanager;.getdeviceid" "landroid/telephony/telephonymanager;.getline1number" "landroid/telephony/telephonymanager;.getnetworkcountryiso" "landroid/telephony/telephonymanager;.getnetworkoperatorname" "ljava/io/bufferedreader;.readline" "landroid/content/pm/packagemanager;.hassystemfeature"	5
"ljava/net/inetaddress;.getlocalhost" "ljava/net/inetaddress;.gethostname" "ljava/net/url;.openstream" "ljava/net/inetaddress;.getbyname" "ljava/net/inetaddress;.equals" "ljava/net/inetaddress;.hashcode" "landroid/net/uri;.parse" "landroid/telephony/smsmanager;.getdefault" "landroid/telephony/smsmanager;.dividemessage" "landroid/telephony/smsmanager;.sendtextmessage" "landroid/telephony/telephonymanager;.getdeviceid" "landroid/telephony/telephonymanager;.listen"	5
"ljava/net/urlencoder;.encode" "ljava/net/uri;.<init>" "landroid/location/location;.hasaccuracy" "landroid/location/location;.distanceto" "landroid/location/location;.gettime" "landroid/location/location;.getaccuracy" "landroid/location/location;.getlatitude" "landroid/location/location;.getlongitude" "landroid/location/location;.getprovider" "landroid/location/locationmanager;.requestlocationupdates" "landroid/location/location;.<init>" "landroid/location/location;.setaccuracy"	5
"ljava/net/urlencoder;.encode" "ljava/net/url;.<init>" "ljava/net/url;.openconnection" "landroid/telephony/telephonymanager;.getline1number" "landroid/telephony/smsmanager;.getdefault" "landroid/telephony/smsmanager;.sendtextmessage" "landroid/telephony/smsmessage;.getdisplayoriginatingaddress" "landroid/telephony/smsmessage;.getmessagebody" "landroid/telephony/smsmessage;.createfrompdu"	5

Table 3. Weight assignment to various features

Feature type	Weight assigned
Suspicious permissions	3
Suspicious permission combinations	5
Suspicious API combinations	5
Suspicious content URI	6
Manifest violation	7
Presence of executable	10

AUTHORS

Babu Rajesh V has been working for three years in the field of mobile security and malware analysis. His areas of interests include mobile security and embedded security.

Phaninder Reddy has been working for two years in the field of mobile security and malware analysis. His areas of interests include machine learning and data analytics.

Himanshu Pareek has around six years of experience in developing and design of security solutions related to small sized networks. He has research papers published on topics like malware detection based on behavior and application modeling.

Mahesh U Patil received master degree in electronics and communication. Presently he is working as Principal Technical Officer at Centre for Development of Advanced Computing. His research interests include Mobile Security and Embedded Systems.