Computers and Electrical Engineering 61 (2017) 266–274

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Machine learning aided Android malware classification☆

Nikola Milosevic[a], Ali Dehghantanha[b], Kim-Kwang Raymond Choo[c,*]

[a] School of Computer Science, University of Manchester, UK
[b] School of Computing, Science and Engineering, University of Salford, UK
[c] Department of Information Systems and Cyber Security, The University of Texas at San Antonio, San Antonio, TX 78249-0631, USA

ARTICLE INFO

Article history:
Received 9 November 2016
Revised 10 February 2017
Accepted 13 February 2017
Available online 22 February 2017

MSC:
00-01
99-00

Keywords:
Static malware analysis
OWASP
Seraphimdroid Android app
OWASP Seraphimdroid Android app
Machine learning

ABSTRACT

The widespread adoption of Android devices and their capability to access significant private and confidential information have resulted in these devices being targeted by malware developers. Existing Android malware analysis techniques can be broadly categorized into static and dynamic analysis. In this paper, we present two machine learning aided approaches for static analysis of Android malware. The first approach is based on permissions and the other is based on source code analysis utilizing a bag-of-words representation model. Our permission-based model is computationally inexpensive, and is implemented as the feature of OWASP Seraphimdroid Android app that can be obtained from Google Play Store. Our evaluations of both approaches indicate an F-score of 95.1% and F-measure of 89% for the source code-based classification and permission-based classification models, respectively.

1. Introduction

In our increasingly connected society, the number and range of mobile devices continue to increase. It is estimated that there will be approximately 6.1 billion mobile device users by 2020 [1]. The wealth of private information that is stored on or can be accessed via these devices made them an attractive target for cyber criminals [2]. Studies have also revealed that users generally do not install anti-virus or anti-malware app installed on their mobile devices, although the effectiveness of such apps is also unclear or debatable [3]. Hence, mobile devices are perceived by security professionals among the "weakest links" in enterprise security.

While all mobile operating systems/platforms have been targeted by malware developers, the trend is generally to focus on mobile operating systems with a larger market share. A bigger market share [4] along with Google's flexible publishing policy on Android's official application (also referred to as app) market – Google Play – resulted in Android users being a popular target for malware developers. It is also known that Android permission-based security model provides little protection as most users generally grant apps requested permissions [5]. There have also been instances where malicious apps were successfully uploaded to Google Play [6]. This suggests a need for more efficient Android malware analysis tools.

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Guest Editor Dr. A. K. Sangaiah.

* Corresponding author.
E-mail addresses: nikola.milosevic@manchester.ac.uk (N. Milosevic), a.dehghantanha@salford.ac.uk (A. Dehghantanha), raymond.choo@fulbrightmail.org (K.-K.R. Choo).

http://dx.doi.org/10.1016/j.compeleceng.2017.02.013
0045-7906/

Existing approaches for malware analysis can be broadly categorized into dynamic malware analysis and static malware analysis. In static analysis, one reviews and inspects the source code and binaries in order to find suspicious patterns. Dynamic analysis (behavioral-based analysis) involves the execution of the analyzed software in an isolated environment while monitoring and tracing its behavior [7].

Early approaches to mobile malware detection were based on the detection of anomalies in battery consumption [8]. Operating system events, such as API calls, Input/Output requests, and resource locks, have also been used in dynamic malware detection approaches. For example, TaintDroid is a malware detection system based on anomalies in the app's data usage behavior [9]. In [10], the authors created a system that monitored anomalies in Android Dalvik op-codes frequencies to detect malicious apps. Several approaches utilized machine learning to classify malware based on their behaviors. For example, the authors in [11] focused on run-time behavior and classified Android malware into the malware families using inter-process communications in combination with SVM. A random forest-based approach with set of 42 vectors including battery, CPU and memory usage as well as network behavior was also used for Android mawlare detection in [12]. In [13], the authors used system calls and regular expressions to detect data leakage, exploit execution and destructive apps.

In order to avoid degrading of mobile device's performance, solutions based on distributed computing and collaborative analysis for both static and dynamic malware analysis have also been proposed [7]. For example, M0Droid is an Android anti-malware solution that analyzes system calls of Android apps on the server and creates signatures which are pushed to the user devices for threat detection [14].

Static malware analysis techniques mainly rely on manual human analysis, which limits the speed and scalability of investigation. Different approaches to automate the static analysis process have also been proposed. Mercaldo et al. [15] suggested transforming malware source code into Calculus for Communicating Systems (CCS) statements and utilized formal methods for checking the software's behavior. However, their approach requires human analysts to formally describe the unwanted behavior, which could still be time-consuming. The authors in [16] proposed a methodology to generate fingerprints of apps that captures binary and structural characteristics of the app. Machine learning techniques can be used to automate static malware analysis process. In [17], pattern recognition techniques are used to detect malware, while other works used standard machine learning algorithms such as perception, SVM, locality sensitive hashing and decision trees to assist in malware analysis (see [18]). In [19], the authors extracted network access function calls, process execution, string manipulation, file manipulation and information reading, prior to applying different machine learning algorithms to classify malicious programs. In [20], the authors extracted 100 features based on API calls, permissions, intents and related strings of different Android apps and applied Eigen space analysis to detect malicious programs. Sahs and Khan used Androguard to obtain permissions and control flow graphs of Android apps and created a SVM-based machine learning model to classify Android malware [21].

In this paper, we demonstrate the utility of employing machine learning techniques in static analysis of Android malware. Specifically, techniques such as manifest analysis and code analysis are utilized to detect malicious Android apps. The contributions of this paper are two-folded:

1. We present a machine learning model for Android malware detection based on app permissions. This approach is lightweight and computationally inexpensive, and can be deployed on a wide range of mobile devices.
2. We then present a new approach to perform code analysis using machine learning, which provides higher accuracy and is capable of revealing more granular app behaviors. Static code analysis of malware is a task generally undertaken by forensics and malware analysts. However, our research results indicate the potential to automate several aspects of the static code analysis, such as detecting malicious behavior within the code.

The structure of this paper is as follows. In the next section, we present the research methodology used in this paper. Research results are then presented, followed by a discussion of the findings. Finally, the paper is concluded and several future directions are suggested.

2. Methodology

Combination of permissions may give a clear indication about the capabilities of the analyzed app(s). From combining the permissions, it can be induced weather the app may cause harm or behave maliciously. We hypothesize that malicious apps will have certain patterns and common permission combinations, which can be learned by a machine learning algorithm. On the other hand, the app code reflects the app's behavior and, therefore, is a common choice for static malware analysis. We utilize two machine learning techniques, namely: classification and clustering. As apps can be classified into malware and goodware, the task of malware detection can be modeled as a classification problem.

Classification is a supervised machine learning technique, which can be used to identify category or sub-population of a new observation based on labeled data. Clustering is an unsupervised machine learning technique that is capable of forming clusters of similar entities. Clustering algorithms are useful when only a small portion of dataset is labeled. The labeled examples can be utilized to infer the class of unlabeled data. Labels obtained through clustering can be subsequently used to retrain a classification model with more data.

Also, in this research, we conducted four experiments, namely: permission-based clustering, permission-based classification, source code-based clustering, and source code-based classification. For the training and testing of our machine learning models, we utilize M0Droid dataset, which contains 200 malicious and 200 benign Android apps [14].

2.1. Permission-based analysis

Since Android security model is based on app permissions, we use permission names as features to build a machine learning model. Every app has to acquire the required privileges to access the different phone features. During an app installation, a user is asked whether to grant the app access to the permissions requested. Malicious apps usually require certain permissions. For example, in order to access and exfiltrate sensitive information from the SD card, a malicious app would require access to both the SD card and Internet. Our approach is to model combinations of the Android permissions requested by such malicious apps. We propose an approach that uses the appearance of specific permissions as features for a machine learning algorithm.

In this approach, we first extract the permissions from our dataset and create a model. For training, we use Weka toolkit and evaluate several machine learning algorithms, including SVM, Naive Bayes, C4.5 Decision trees, JRIP and AdaBoost. Classification algorithms we chose differ in their underlying concept. *Support Vector Machines* is a non-probabilistic supervised machine learning binary classification algorithm. SVM is capable of nonlinear classification that maps inputs into high dimensional feature space. *C4.5 decision tree* is a statistical classifier that builds a decision tree based on information entropy. Each node of the tree algorithm selects a feature and splits its sets of samples into subsets until classes can be inferred. *Random forest* is an ensemble classification algorithm that combines a number of decision trees and returns the mode of individual decisions by decision trees. *Naive Bayes* is a simple probabilistic classifier that is based on applying Bayes theorem with strong independence assumption between features. *Bayesian network* is a probabilistic graphical model that represents a set of random variables and their inter-dependencies in directed acyclic graph. *JRIP* is a propositional rule learner that tries every attribute with every possible value and adds a rule which results to the greatest information gain. *Logistic regression* is a statistical regression model where dependent variable is used to estimate the probability of binary response based on multiple features. *AdaBoost* is a meta algorithm that can be used with many other algorithms to improve their performance by combining their outputs into a weighted sum which represents the final output.

We then used the modified Weka 3.6.6 library[1] for Android to develop the OWASP Seraphimdroid Android app, which is using support vector machines with sequential minimal optimization model.[2]

We also apply several clustering techniques in order to evaluate the performance of our unsupervised and supervised learning algorithms. Training, testing and evaluation of our model are performed using Weka Toolkit by applying the Farthest First, Simple K-means and Expectation maximization (EM) algorithms. *Simple K-means* is a clustering algorithm where samples are clustered into n clusters, in which each sample belongs to a cluster with the nearest mean. *Farthest First* algorithm uses farthest-first traversal to find k clusters that minimize the maximum diameter of a cluster, and *Expectation maximization (EM)* assigns a probability distribution to each instance which indicates the probability of it belonging to each of the clusters.

2.2. Source code-based analysis

The second approach is a static analysis of the app's source code. Malicious codes generally use a combination of services, methods, and API calls in a way that is not usual for non-malicious app [11]. Machine learning algorithms are capable of learning common combinations of malware services, API and system calls to distinguish them from non-malicious apps.

In this approach, Android apps are first decompiled and then a text mining classification based on bag-of-words technique is used to train the model. Bag-of-words technique has already showed promising results for classification of harmful apps on personal computers [22]. Decompiling Android apps to conduct static code analysis involves several steps. First, it is necessary to extract the Dalvik Executable file (dex file) from the Android application package (APK file). The second step is to transform the Dalvik Executable file into a Java archive using the dex2jar tool.[3] Afterwards, we extract all .class files from the Java archive and utilize Procyon Java decompiler (version 0.5.29) to decompile .class files and create .java files. Then, we merge all Java source code files of the same app into one large source file for further processing.

Since Java and natural language text have some degree of similarity, we apply the technique used in natural language processing, known as "a bag-of-words". In this technique, the text, or Java source code in our case, is represented as a bag or set of words which disregards the grammar or word order. The model takes into account all words that appear in the code. Our approach considers the whole code including import statements, method calls, function arguments, and instructions. The source code obtained in the previous step is then tokenized into unigrams that are used as a bag-of-words. We use several machine learning algorithms for classifications, namely: C4.5 decision trees (in Weka toolkit, it is known as J48), Naive Bayes, Support Vector Machines with Sequential Minimal Optimization, Random Forests, JRIP, Logistic Regression and AdaBoostM1 with SVM base. We performed our training, testing and evaluation using Weka Toolkit. For source code analysis, we also utilized ensemble learning with combinations of three and five algorithms and majority voting decision system. Ensemble learning combines multiple machine learning algorithms over the same input, in hope to improve the classification performance. The number of algorithms is chosen in a way that system is able to unambiguously choose the output class based on majority of votes.

[1] http://www.pervasive.jku.at/Teaching/lvaInfo.php?key=346&do=uebungen

[2] https://www.owasp.org/index.php/OWASP_SeraphimDroid_Project https://github.com/nikolamilosevic86/owasp-seraphimdroid

[3] https://github.com/pxb1988/dex2jar

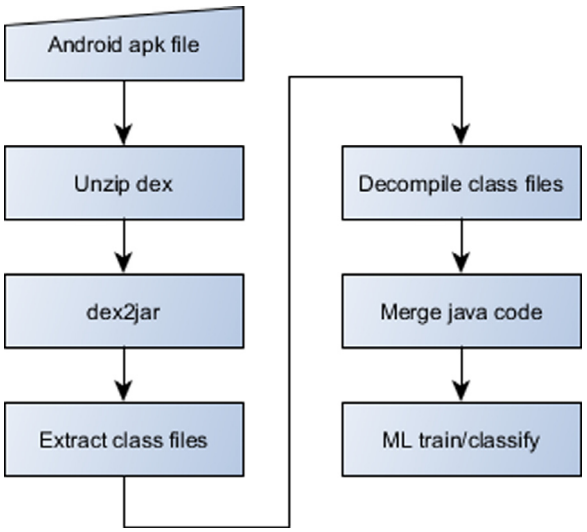

Fig. 1. Workflow of Android file decompiling and machine learning-based malware detection methodology.

We also experiment with clustering on the source code. Clustering algorithms we use include the Farthest First, Simple K-means and Expectation maximization (EM). A flow diagram of the process is presented in Fig. 1.

2.3. Ensemble learning

To improve the performance of our learning algorithms, our tests were performed using ensemble learning with voting for both permission-based and source code-based analysis. Ensemble methods use multiple classification algorithms to obtain better performance than could be obtained from any of the constituent algorithms individually. The final prediction is chosen as the label that was predicted by the majority of classifiers. We also experiment with ensembles that contained combinations of three and five algorithms. Odd number of algorithms allow us to unambiguously choose the class with majority voting. For classification algorithms, we use SVM, C4.5, Decision Trees, Random Tree, Random Forests, JRIP, and Linear Regression.

3. Evaluation and discussion

We evaluated the performance of our approaches using 10-fold cross validation. In 10-fold cross validation, the original sample was randomly partitioned into ten equal sized sub-samples. A single sub-sample was retained for the testing, while the remaining nine were used for training. The process was repeated ten times, and each time using a different sub-sample for testing. The results were then averaged to produce a single estimation. The main advantage of this method is that all samples were used once only for validation. The metrics we used for the evaluation of the algorithms are precision, recall and F-measure, which are widely used in the text mining and machine learning communities. Classified items can be true positive (TP – items correctly labeled as belonging to the class), false positive (FP - items incorrectly labeled as belonging to a certain class), false negative (FN - items incorrectly labeled as not belonging to a certain class), and true negative (TN - items correctly labelled as not belonging to a certain class).

Given the number of true positives and false negatives, recall is calculated using the following formula:

$$Recall = \frac{TP}{(TP + FN)}$$

The recall is sometimes referred to as “sensitivity” or the “true positive rate”.

Given the number of true positive and false positive classified items, precision (also known as “positive predictive rate”) is calculated as follows:

$$Precision = \frac{TP}{(TP + FP)}$$

The measure that combines precision and recall is known as F-measure, given as:

$$F = \frac{(1 + \beta^2) \times Recall \times Precision}{\beta^2 \times (Precision + Recall)},$$

where β indicates the relative value of precision. A value of $\beta = 1$ (which is usually used) indicates the equal value of recall and precision. A lower value indicates a larger emphasis on precision and a higher value indicates a larger emphasis on recall [23].

Table 1
Evaluation results of permission-based classification using single machine learning algorithms.

Algorithm	Precision	Recall	F-score
C4.5 decision trees	0.827	0.827	0.827
Random forest	0.871	0.866	0.865
Bayes Networks	0.747	0.747	0.747
SVM with SMO	0.879	0.879	0.879
JRip	0.821	0.819	0.819
Logistic regression	0.823	0.822	0.821

Table 2
Evaluation results of permission-based classification using ensemble learning.

Algorithm	Precision	Recall	F-score
Random tree + Random forest + C4.5	0.878	0.876	0.876
Random tree + Random forest + SVM with SMO	0.885	0.884	0.884
SVM with SMO + Logistic regression + Random forest	0.892	0.891	0.891
Bayes Nets + SVM with SMO + Logistic Regression	0.879	0.876	0.876
C4.5 + Random forests + Random tree + SVM with SMO + Logistic regressing	0.895	0.894	0.894

3.1. Evaluation of permission-based classification

The evaluation of machine learning algorithms performing permission-based classification is presented in Table 1.

As observed from Table 1, support vector machines with sequential minimal optimization has the best performance with a F-measure value of 0.879. In other words, this algorithm correctly classified 87.9% of test instances in 10-fold cross validation. The algorithm is also efficient, in terms of speed, as it took only 0.04 s to train the model. Instances were also classified faster; thus, making this approach suitable for real-time classification of (malicious) apps. We then integrated this model for classification based on permissions with SVM in the OWASP Seraphimdroid Android app,[4] which can be obtained from Google Play Store.[5]

On the other hand, Bayesian algorithms such as Naive Bayes and Bayesian networks have the worst performance. This could be due to the small dataset (comprising only 387 instances) used in this study. Bayesian algorithms usually require much larger datasets than SVM to train the model with a higher accuracy. A larger dataset may also improve SVM model performance.

SVM algorithm outperforms Naive Bayes, Bayesian Network, JRip and Logistic regression on statistical *t*-test with a confidence interval of 0.05. However, it is not significantly better than decision trees and random forests.

In Table 2, we present the results of ensemble learning using majority voting. We experimented with ensembles of three algorithms in order to determine which algorithm(s) contribute to the best results in ensembles. The best three performing algorithms are SVM with SMO, Logistic Regression and Random Forest with an F-measure of 0.891. This is only a slight improvement compared to using the SVM algorithm in isolation. The *t*-test suggested that ensemble learning is not significantly better with a confidence interval of 0.05.

On the other hand, ensemble algorithms were much slower as more time is needed to apply multiple machine learning algorithms (in our case, three or five) and post-process results. Since the significance test showed that the performance of the ensemble learning algorithm is not significantly better than the single machine learning algorithm, there is no benefit in using these algorithms in production.

Both results from the single classifier and ensemble method present a promising performance that can be used in anti-malware systems. This method would be able to detect unknown and new malware samples since it does not rely on signatures, but rather on learned dangerous permission combination. Our findings echoed the findings of previous studies such as Kolter and Maloof [24], which demonstrated the potential of machine learning algorithms in achieving a high detection rate, even on new malware samples.

There are, however, limitations with this approach. For the permission-based approach, we reported an F-measure of 87.9% for single machine learning algorithms. In other words, some malware samples would be undetected and some non-malicious apps classified as malicious. In our case, 340 apps were correctly classified, while 47 were incorrectly classified. Using ensemble learning, the number of misclassified instances was reduced to 42. Our reported performance is higher than those reported in [25]. Also, our permission-based analysis model is not computationally expensive and when implemented in the OWASP Seraphimdroid app, we were able to scan and classify all 83 installed apps on the test device (i.e., a Nexus 5 device with Quad-core 2.3 GHz, 2GB RAM) in under 8 s.

[4] https://github.com/nikolamilosevic86/owasp-seraphimdroid
[5] https://play.google.com/store/apps/details?id=org.owasp.seraphimdroid

Table 3
Evaluation results of permission-based clustering.

Algorithm	Correctly clustered instances	Incorrectly clustered instances
SimpleKMeans	229 (59.17%)	158.0 (40.83%)
FarthestFirst	199 (51.42%)	188.0 (48.58%)
EM	250 (64.6%)	137.0 (35.4%)

Table 4
Evaluation results of source code-based classification using single machine learning algorithm.

Algorithm	Precision	Recall	F-score
C4.5 decision trees	0.886	0.886	0.886
Random forest	0.937	0.935	0.935
Naive Bayes	0.825	0.821	0.820
Bayesian networks	0.825	0.821	0.819
SVM with SMO	0.952	0.951	0.951
JRip	0.916	0.916	0.916
Logistic regression	0.935	0.935	0.935

Table 5
Evaluation results of source code-based classification using ensemble learning.

Algorithm	Precision	Recall	F-score
C4.5 decision tree + random tree + random forests	0.950	0.948	0.948
Logistic regression + C4.5 + SVM with SMO	0.947	0.946	0.946
Random tree + Random Forest + SVM with SMO	0.825	0.821	0.820
SVM with SMO + Logistic regression + Random forest	0.942	0.940	0.940
SVM with SMO + Logistic regression + AdaBoostM1 with SVM base	0.952	0.951	0.951
Logistic regression + JRip + Random Forests + C4.5 + SVM with SMO	0.950	0.948	0.948
SVM with SMO + Logistic regression + Simple Logistic regression + AdaBoostM1 with SVM base	0.958	0.957	0.956

3.2. Evaluation of permission-based clustering

Clustering refers to the grouping of similar items together, without any knowledge of how the grouping should be performed. Clustering is different from supervised learning, where the training set is defined in a way that shows how to perform classification. In clustering, there are no labels or training sets. The set of elements is clustered into a certain number of groups, which are usually formed based on the element's similarity.

Table 3 presents the results of our permission-based clustering approach. In our case, apps will be grouped according to whether they use a similar set of permissions. However, if an app uses a similar set of permissions as some malware, it does not mean that the app is malicious. As it can be seen from Table 3, the results are not as good as classification. The best algorithm incorrectly clustered more than 35% of the instances while permission-based classification only incorrectly classified around 10.5% of the instances. In our permission-based analysis, clustering had a higher error rate than classification.

3.3. Evaluation of source code-based classification

Of the 400 apps in our data set, we were unable to decompile 32 of them (10 non-malicious and 22 malicious), perhaps due to code encryption and obfuscation or instability of our Java decompiler. Nevertheless, the remaining 368 source files were sufficient to train a good model.

The evaluation of the classification for the analysis of the app's source code is presented in Table 4.

As Table 4 shows, over 95% of instances were correctly classified using SVM. The high accuracy of source code-based classification reveals that the machine can infer app behavior from its source code. Even though the bag-of-words model disregards grammar and word order in text (in our context, the source code), it is possible to train a successful machine learning model that is able to distinguish malicious app from non-malicious app. Other machine learning algorithms such as Random Forests, Logistic Regression and JRip were also evaluated and had an F-score of over 90%. Therefore, source code appears to be a viable source of information for a machine learning classification algorithm. Also, with the machine learning-based source code analysis, it is possible to analyze whether an Android package (apk) is malicious in less than 10 s, which is significantly faster than a human analyst.

In Table 5, we present the results of ensemble learning methods. Ensemble learning with voting had a slight improvement compared to the best results from using single machine learning algorithms (the best F-measure of ensemble learning was 0.956; the F-measure of SVM was 0.951) by combining SVM with SMO, logistic regression, LogitBoost with simple regression functions as base learners (simple logistic regression) and AdaBoostM1 with SVM as a base. Some of the ensembles (e.g. C4.5 decision tree+random tree+random forests or SVM with SMO+Logistic regression+Random Forest) performed worse

Table 6
Evaluation results of permission-based clustering.

Algorithm	Correctly clustered instances	Incorrectly clustered instances
SimpleKMeans	303 (82.3%)	65 (17.66%)
FarthestFirst	296 (80.44%)	72 (19.56%)
EM	300 (81.53%)	68 (18.47%)

than SVM with SMO. Since the F-measure of C4.5 decision trees was 0.886, it had a negative impact on the ensembles. Ensembles that contained SVM may have misclassified some instances if the majority of algorithms voted for the wrong class. The combination of algorithms in one case (SVM with SMO+Logistic regression+Simple Logistic regression+AdaBoostM1 with SVM base) had slightly improved classification performance (by 0.5% in F-measure), but it was not statistically significant. Our source code analysis approach allows successful classification of new malware in 95.1% cases with a single machine learning algorithm.

3.4. Evaluation of source-based clustering

Table 6 presents the results of source code clustering. These results were more promising than the those obtained from permission-based clustering since the best performance of correctly clustered instances increased from 64.6% to 82.3%. The increase in performance is due to the fact that source code provides a greater amount of data based on which clustering can be done. However, there were still 17.6% incorrectly clustered instances. Since clustering is a type of unsupervised machine learning algorithm, it creates clusters that are based on code similarity. This is not necessarily a good indication of the code's malicious behavior. The way clustering maps instances in the absence of any supervision is the main reason that its performance is worse than the classification algorithms. The results for non-supervised learning can be used for creating larger labeled data sets. Classification (SVM) performed 14% better than the best clustering method, which indicates that clustering should not be used for detecting malware but only for expanding small datasets if necessary.

4. Conclusion and future research directions

We presented two machine learning aided (classification and clustering) approaches based on app permissions and source code analysis to detect and analyze malicious Android apps. The use of machine learning allows our algorithms to detect new malware families with high precision and recall rates. Our approach complements existing signature-based anti-malware solutions, as the latter is not capable of detecting malicious software until the appropriate signatures are released. Specifically, we demonstrated that the permission-based method was able to classify malware from goodware in 89% of cases while source code analysis classification performance was over 95%. Accuracy rates of 95.1% using SVM, and 95.6% using the ensemble learning method are comparable with existing state-of-the art solutions.

The majority of existing approaches need to perform analysis on the remote server or they require the Android device to be rooted. However, our permission-based approach can run on Android devices without root access and offers a relatively good accuracy in malware detection. Source code-base analysis approach, to the best of our knowledge, is the only automated Android static malware analysis method that uses machine learning to scan the entire source code of an app. Other static malware detection approaches are usually limited to monitoring a set of API or system calls, ignoring import code snippets such as operator statements and other code features. Our source code-based classification is computationally more expensive as it requires decompilation of the files prior to analysis. However, detailed analysis of decompiled code does not take more than 10 s per app. In practice, this approach can be used to scan and classify any apps including those on Google Play and other app stores.

Future research includes the evaluation of the proposed approaches using a significantly bigger labeled balanced data sets and utilizing online learning. Another research focus is combining static and dynamic software analysis in which multiple machine learning classifiers are applied to analyze both source code and dynamic features of apps in run-time.

References

[1] Boxall A. The number of smartphone users in the world is expected to reach a giant 6.1 billion by 2020; 2015 http://www.digitaltrends.com/mobile/smartphone-users-number-6-1-billion-by-2020/.
[2] Dehghantanha A, Franke K. Privacy-respecting digital investigation. In: 2014 twelfth annual international conference on privacy, security and trust (PST). IEEE; 2014. p. 129–38.
[3] Walls J, Choo K-KR. A review of free cloud-based anti-malware apps for android. In: 14th IEEE international conference on trust, security and privacy in computing and communications (TrustCom/BigDataSE/ISPA). IEEE; 2015. p. 1053–8.
[4] Kitagawa M, Gupta A, Cozza R, Durand I, Glenn D, Maita K, et al. Market share: final pcs, ultramobiles and mobile phones, all countries, 2q15 update Technical Report; 2015.
[5] Chia C, Choo K-KR, Fehrenbacher D. How cyber-savvy are older mobile device users?. In: Au MH, Choo K-K R, editors. Mobile security and privacy; Chapter 4. Waltham, MA: Syngress/Elsevier; 2017. p. 67–83.
[6] Viennot N, Garcia E, Nieh J. A measurement study of google play. In: ACM SIGMETRICS performance evaluation review, 42. ACM; 2014. p. 221–33.
[7] Schmidt A-D, Clausen JH, Camtepe A, Albayrak S. Detecting symbian os malware through static function call analysis. In: 4th international conference on malicious and unwanted software (MALWARE), 2009. IEEE; 2009. p. 15–22.

[8] Buennemeyer TK, Nelson TM, Clagett LM, Dunning JP, Marchany RC, Tront JG. Mobile device profiling and intrusion detection using smart batteries. In: Proceedings of the 41st annual Hawaii international conference on system sciences. IEEE; 2008. 296–296.
[9] Enck W, Gilbert P, Han S, Tendulkar V, Chun B-G, Cox LP, et al. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst (TOCS) 2014;32(2):5.
[10] Canfora G, Mercaldo F, Visaggio CA. Mobile malware detection using op-code frequency histograms. In: Proceedings of international conference on security and cryptography (SECRYPT); 2015.
[11] Dash SK, Suarez-Tangil G, Khan S, Tam K, Ahmadi M, Kinder J, et al. Droidscribe: classifying android malware based on runtime behavior. Mobile Secur Technol (MoST 2016) 2016;7148:1–12.
[12] Alam MS, Vuong ST. Random forest classification for detecting android malware. In: 2013 IEEE and internet of things (iThings/CPSCom), IEEE international conference on and IEEE cyber, physical and social computing, green computing and communications (GreenCom),. IEEE; 2013. p. 663–9.
[13] Isohara T, Takemori K, Kubota A. Kernel-based behavior analysis for android malware detection. In: 2011 seventh international conference on computational intelligence and security (CIS). IEEE; 2011. p. 1011–15.
[14] Damshenas M, Dehghantanha A, Choo K-KR, Mahmud R. M0droid: an android behavioral-based malware detection model. J Inf Privacy Secur 2015;11(3):141–57.
[15] Mercaldo F, Nardone V, Santone A, Visaggio CA. Download malware? No, thanks: how formal methods can block update attacks. In: Proceedings of the 4th FME workshop on formal methods in software engineering. ACM; 2016. p. 22–8.
[16] Karbab EB, Debbabi M, Mouheb D. Fingerprinting android packaging: generating dnas for malware detection. Digital Invest 2016;18:S33–45.
[17] Nataraj L, Karthikeyan S, Jacob G, Manjunath B. Malware images: visualization and automatic classification. In: Proceedings of the 8th international symposium on visualization for cyber security. ACM; 2011. p. 4.
[18] Nath HV, Mehtre BM. Static malware analysis using machine learning methods. Recent Trends Comput Netw Distrib Syst Secur 2014:440–50.
[19] Afonso VM, de Amorim MF, Grégio ARA, Junquera GB, de Geus PL. Identifying android malware using dynamically obtained features. J Comput Virol Hacking Tech 2015;11(1):9–17.
[20] Yerima SY, Sezer S, Muttik I. Android malware detection: an eigenspace analysis approach. In: Science and information conference (SAI), 2015. IEEE; 2015. p. 1236–42.
[21] Sahs J, Khan L. A machine learning approach to android malware detection. In: 2012 European intelligence and security informatics conference (EISIC). IEEE; 2012. p. 141–7.
[22] Benjamin VA, Chen H. Machine learning for attack vector identification in malicious source code. In: 2013 IEEE international conference on intelligence and security informatics (ISI). IEEE; 2013. p. 21–3.
[23] Hersh W. Evaluation of biomedical text-mining systems: lessons learned from information retrieval. Brief Bioinform 2005;6(4):344–56.
[24] Kolter JZ, Maloof MA. Learning to detect and classify malicious executables in the wild. J Mach Learn Res 2006;7:2721–44.
[25] Cyveillance. Cyveillance testing finds av vendors detect on average less than 19% of malware attacks; 2010 http://www.businesswire.com/news/home/20100804005348/en/Cyveillance-Testing-Finds-AV-Vendors-Detect-Average.

Nikola Milosevic is a PhD student at the University of Manchester, School of Computer Science, where his research topics focus around machine learning and natural language processing. Also he is involved with OWASP (Open Web Application Security Project) as a founder of OWASP Serbia local chapter, OWASP Manchester local chapter leader and a project leader of OWASP Seraphimdroid mobile security project.

Ali Dehghantanha is a Marie-Curie International Incoming Fellow in Cyber Forensics, a fellow of the UK Higher Education Academy (HEA) and an IEEE Sr. member. He has served for many years in a variety of research and industrial positions. Other than Ph.D in Cyber Security, he holds several professional certificates such as GXPN, GREM, CISM, CISSP, and CCFP.

Kim-Kwang Raymond Choo holds the Cloud Technology Endowed Professorship at The University of Texas at San Antonio. He is the recipient of ESORICS 2015 Best Paper Award, Germany's University of Erlangen-Nuremberg Digital Forensics Research Challenge 2015, Fulbright Scholarship, and British Computer Society's Wilkes Award, etc, and is an Australian Computer Society Fellow, and a Senior Member of IEEE.