ANDROID MALWARE DETECTION USING MACHINE LEARNING AND REVERSE ENGINEERING

Michal Kedziora, Paulina Gawin, Michal Szczepanik and Ireneusz Jozwiak

Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland

ABSTRACT

This paper is focused on the issue of malware detection for Android mobile system by Reverse Engineering of java code. The characteristics of malicious software were identified based on a collected set of applications. Total number of 1958 applications where tested (including 996 malware apps). A unique set of features was chosen. Five classification algorithms (Random Forest, SVM, K-NN, Nave Bayes, Logistic Regression) and three attribute selection algorithms were examined in order to choose those that would provide the most effective malware detection.

KEYWORDS

Malware Detection, Android, Random Forest, SVM, K-NN, Naive Bayes, Logistic Regression

1. INTRODUCTION

The basis for the effective detection of malware is the analysis of malware on a given platform. The main malware detection techniques consist of static and dynamic analysis [16]. Dynamic analysis techniques rely on monitoring the application in real time, working in an isolated environment [1]. Static analysis works on decompiled source code, without launching applications [2] analyzing the case of reporting rights, components, API calls. In this paper we are focused on static approach case based on the automatic analysis of decompiled mobile application code. Based on reference items [3], [4] and [5], a unique feature vector derived from the application Java code was constructed. The total number of features is 696. We divided them into three categories: First one is model implementation of onReceive() methods for BroadcastReceiver components. As demonstrated in [3], in malware applications, calls to certain methods more often occur in the overridden onReceive() method than in secure applications. The full list of wanted calls in the onReceive() method of components extending the BrodacastReceiver class can be found in first part of Table 1. Second one is Linux system commands - as the Android system uses the Linux kernel, there is an API available to execute Linux-specific commands on the Android mobile device. Some of the commands under examination relate to operations on the file system. There is also a group of commands that are used to obtain administrative access to the device (rooting), then to increase the possibilities of attack, and to hide the operation of malware on the device. The full list of searched Commands is available in second part of table 1. Selected on the basis of [6]. Third one is API Calls - the largest group of features (616). Includes methods from classes, some of which have been

Natarajan Meghanathan et al. (Eds) : SPPR, SCAI, CSIA, WiMoA, ICCSEA, InWeS, NECO, GridCom - 2018
pp. 95–107, 2018. © CS & IT-CSCP 2018 DOI : 10.5121/csit.2018.81709

indicated in [5]. Third part of Table 1 contains a list of classes, whose selected methods were included in the extraction of features. These were classes characteristic of the context of the mobile application, for objects of type Intention, for HTTP protocol operations, for telephone operations (SMS, connection, MMS), for device network settings, for data encryption or for dynamic code loading [19].

2. PREPARING TESTING ENVIRONMENT

Based on the literature on the subject, especially on [4], [7], [8], [9], [10], [11], [5], [12], [18], five classification algorithms were selected for testing under this work. These algorithms were among the most popular in the field of malware detection on the Android system. A collection of tested data consist of secure and malicious applications. Safe applications have been downloaded from two sources. First one is APKPure - an alternative Android application store. Second one is F-Droid - the directory of the FOSS Android application (Free and Open Source Software), which includes free and open software [17]. To increase the likelihood that applications are considered safe indeed they are not malicious, each of the downloaded files has been uploaded to the web application VirusTotal. Its task consists of analyzing the file using over 70 antivirus scanners, indicating whether the file is malicious, additionally revealing a label, indicating the species of malware to which the given scanner has classified a dangerous file. Secondly, VirusTotal provides additional information about the file, such as: the date of the first and last file upload operation to VirusTotal, the number of these operations, the results of static analysis (eg internal structure of the file), the results of dynamic analysis (behavioral characteristics of the application). The condition for joining the application to the test set of this work was not to detect malicious activity by any of the more than 70 scanners. Malicious applications also came from two sources: VirusShare - malware repository, currently containing over 24 million and Contagio Mobile - a blog that is part of the Contagio Dump project, which is a collection of malware samples. Contagio Mobile focuses on mobile malware, especially on Android and iOS. Applications downloaded from the above two sources have also been analyzed in VirusTotal. They were added to the test database if at least one of the scanners classified them as malware.

2.1. Java Code Features extraction

To be able to extract the features, the files should be prepared properly. To this end, BASH shell scripts have been developed and auxiliary scripts that organize files. Scripts gets the .apk file to the input, returning the corresponding .jar file. For this purpose, the dex2jar tool is used. It's used for work with .dex and .class files. It enables: reading and writing to a .dex file (Dalvik Executable format, executable format for Dalvik), conversion from a .dex file to .class files (compressed as a .jar file), disassembling the .dex file to a smali format, as well as the decryption of code strings present in the code, which have been obfuscated through encryption and whose decryption was to take place only after execution.

Table 1. Methods, commands and classes from which methods where extracted belonging to the feature vector from java code.

BroadcastReceiver	psneuter	StringBuilder
startService	wpthis	Process
bindService	exploid	Context
schedule	rageagainstthecage	Intent
startForegroundService	motofail	ActivityManager
registerReceiver	GingerBreak	PackageManager
goAsync	**Classes**	SmsManager
startActivity	ContentResolver	TelephonyManager
startActivieties	Cipher	DexClassLoader

Commands	Class	BaseDexClassLoader
su	File	ClassLoader
mount	FileOutputStream	Runtime
insmod	DataOutputStream	System
rebot	psneuter	ConnectivityManager
chown	wpthis	NetworkInfo
pm install	exploid	WifiManager
zergRush	rageagainstthecage	HttpURLConnection
m7	motofail	Socket
fre3vo	GingerBreak	handler

A Java program was prepared for the extraction of the tests. The external library used for the extraction of features is JD-core-java - a package decompiling Java decompiler called Java Decompiler (Java Decompiler authors did not provide a tool in the form of a library that can be used inside the code, only a decompiler as a plug-in for selected programming environments or graphics tool). Necessary to obtain the application code from the .jar file, on which the analysis of malware in the second research case is based. To focus on the analysis of features selected by all three selection methods, in Table 2 there are features present in the top characteristics for each selection method and their participation in malware applications and secure applications. All features from Table 2 are much more common in malware than in secure applications.

Table 2. Percentage of occurrences of features derived from java code, which are in the top features chosen by 3 selection algorithms in the malware and safe application sets.

Feature	**Percentage in malware**	**Percentage in safe apps**
startService	0.34809	0.00118
getString	0.35412	0.04471
setPackage	0.25553	0
putExtra	0.27767	0.01059
startActivity	0.19215	0.01882
getSystemService	0.17907	0.02000
append	0.20121	0.04588
indexOf	0.11268	0.00941
getInputStream	0.09558	0.00235

Methods from the String class, such as append or indexOf, which are much more prevalent in malicious applications than in safe ones, show a legal manipulation on the string of characters to obfuscate the code. Operations on strings are used to avoid detection by dynamically creating URLs, providing parameters to the reflection mechanism API, or to hide Linux commands. The proof of probable manipulations with Linux commands is that the method for calling them appeared in 1% of malicious applications, while the extractor detected ten times less true Linux commands (at the level of 0.1%) - thus the vast majority of applications that calls the method to Linux commands do not contain these explicit commands (or they are very unpopular and unusual commands - but this alternative is less likely). In addition, the Context.getString method allows to extract a string from application resources that are outside the Java code. Therefore, this is a great opportunity to save a dangerous string of characters, e.g. the URL of a malicious server, in the application resources, so that it will not be detected during Java code analysis, and this method allows you to download this string to the code.

The Context.startService method, the frequency of which in malware applications was mentioned in [5], is used to start the service. In the dataset of this work, it occurred about 350 times more often in malware. Knowing that the service is a component running in the background, possibly

without the user's knowledge, it seems clear that malicious applications will want to reach for the described method in order not to alert the user about malicious activity by using a component that will work in hiding.

The getSystem method. The Context class service, appearing in malicious applications 9 times more often than in safe applications, allows access to given system services. Without examining the parameters of this method, it is difficult to conclude what specifically access was requested for. However, among the system services that this method gives access to, there are those that can be potentially dangerous: window visibility management, network connections, Wi-Fi connectivity, HTTP download process, location data.

The getInputStream method from the HttpURLConnection class, occurring in almost 10% of malicious applications, and only in 0.2% of secure applications, is important in the process of data transfer (both sending and receiving) via the HTTP protocol. Thanks to this method, on the one hand, the application can receive harmful packages (payload), on the other hand, send sensitive data about the user to the external server.

Intent: setPackage and putExtra methods, used mostly in malware in comparison with secure applications, may have their justification in intentional intentions. Intentional intentions are those that do not indicate a specific component that the intention can pick up. Therefore, it is possible that the intention will be received by another application. The threat occurs when a secure application uses implicit intent, does not specify which component can perform the action, and then such an intention intercepts the malware. Then he will be able to send the application in response to inaccurate data or send information about the success of the operation at the moment when the operation did not take place. The result of intention can be saved using the putExtra function. The setPackage function is used to determine which components can receive the intention. Perhaps such a large presence in the malware serves to specify exactly which component should be responsible for the actions in order to have full control over the course of malicious activity, so that no other application could accidentally intercept the expected event.

It is noted that only two patterns listed at all appear in the code of the tested applications - and these are the startService and schedule methods. Appear on the level of 3% and 1% respectively in the malware code. They are used to activate the service accordingly and scheduling the service. Statistical tests were performed on the characteristics of Table 2 using the Mann-Whitney U-test. The confidence level at 0.05 was assumed. For each of the features in Table 2, the null hypothesis of median equality was rejected and an alternative hypothesis with a larger median in the malware population was adopted than in the population of safe applications.

2. PRACTICAL MALWARE DETECTION

The research will cover features obtained from the application code. Three methods of feature selection where be tested. Then, for each classifier, its selected parameters will be tested, and with the adopted determined parameters, the classification will be determined depending on the number of features taken into account. Next, the most common features in malware will be listed. The best results in terms of the number of correctly classified instances will be compiled for the 5 tested classifiers, along with the time of the algorithm's operation and the time of the pre-processing process and the extraction of features. For testing each classifier, 10-fold cross validation will be used, repeated 10 times. The stages are as follows: First is selection of features, second is characteristics of malware, third is classifiers and their parameters (see Table 3), fourth is summary of the best results and time data for classifiers and last each of the 5 classifiers will be tested for selected parameters (see second column of Table 3).

Table 3. Parameters tested for classifiers.

Classifier	**Parameter**
Random Forest	Iterations / max depth
Naive Bayes	-
logistic regression	Iterations
k-NN	number of neighbors
SVM	Kernel function

2.1. Random Forest

First test case included changing of the maximum depth of the tree with assumptions: number of iterations: 100, number of features: 696. Table 4 shows that after reaching the maximum depth of 50, the percentage of correctly classified instances was stable - and in the range of maximum depth from 80 to 200 and equal to infinity even identical. Some of the other indicators also remained at the same level from a depth greater than or equal to 80: TP, FN and F-measure. Interestingly, among these research cases, the lowest time was recorded for a maximum depth of 100. Measures TN and FP had the best result for a depth of 20.

Table 4. Results of the examination of the influence of the maximum depth for a random forest on the percentage of correctly classified instances, the root of mean square error and the time of learning and testing.

Max Depth	Correctly classified	mean square error	Learning and testing time [ms]	TP	TN	FP	FN	F-measure
10	78,1598	0,4268	15227,6102	0,6305	0,9582	0,0418	0,3695	0,7555
20	80,1345	0,3917	31206,3691	0,6564	0,9708	0,0292	0,3436	0,7797
30	80,2212	0,3731	46360,0157	0,667	0,9604	0,0396	0,333	0,7833
40	80,5359	0,3656	62293,7623	0,677	0,9554	0,0446	0,323	0,7885
50	80,6118	0,3636	93321,374	0,68	0,9536	0,0464	0,32	0,7899
100	80,6444	0,3637	104305,4938	0,6808	0,9534	0,0466	0,3192	0,7904
200	80,6444	0,3637	122231,6517	0,6808	0,9534	0,0466	0,3192	0,7904

However, it is worth noticing a very large difference between the matrices of the confusion matrix. The average value of TP was 68%, while TN - 96%. Similarly, the average FP value was 4%, and FN - 32%. The high FN value, i.e. the second type error, seems to be more dangerous in the field of malware detection - not detecting software malware and using it may result in the unconscious infection of the system or even the network of systems, while the error of the first type, i.e. the recognition of a safe application as dangerous is a false alarm, which can be further examined and reversed the diagnosis, and then use the application securely.

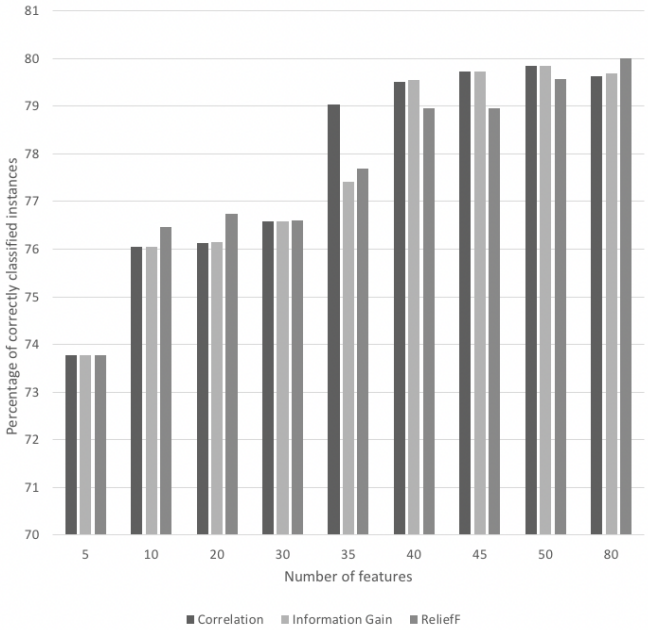

Figure 1. Percentage of correctly classified instances with changing number of attributes and attribute selection method (Random Forest)

Second test case included changing of the number of iterations with assumptions: maximum depth: 80, number of features: 696. The percentage of correctly classified instances, varying with the number of iterations. The highest percentage of correctly classified instances was recorded for 55 iterations and it was 80.6662%. 55 iterations are also the case of the highest TN index value (0.9542) and the lowest FP index value (0.0458). For the 80 iterations, the best results were obtained for FN (0.319) and F (0.7906). Again, the second type of error is much greater than the error of the first type, and the percentage of correctly classified instances never exceeded 81%, which proves the poor quality of the classification.

Figure 2. Percentage of correctly classified instances with varying number of attributes and attribute selection method (SVM)

Third test case included changing of the number of features with assumptions: maximum depth: 80, number of iterations: 55. Fig. 2 presents the summary results from the research on the change in the number of features (and the algorithm of feature selection) and its impact on the percentage of correctly classified instances. 5 to 80 best traits with step 5 were examined. Only at 80 guilds

selected using the ReliefF algorithm, this index was achieved at over 80%. It is not possible to distinguish a strong favorite among the algorithms of feature selection - they came out on the lead at different times all the algorithms studied.

2.2 Naive Bayian Classifer

This test case included changing of the number of features. Figure 3 shows how the percentage of correctly classified instances has changed depending on the number of features selected by each selection method. The highest score achieved over 80 attributes is 76.12% for 10 features selected by the ReliefF algorithm. In two cases - 10 and 35 attributes - it dominated competitors. Despite that, from 45 to 80 features, there was no significant improvement in the quality of the classification.

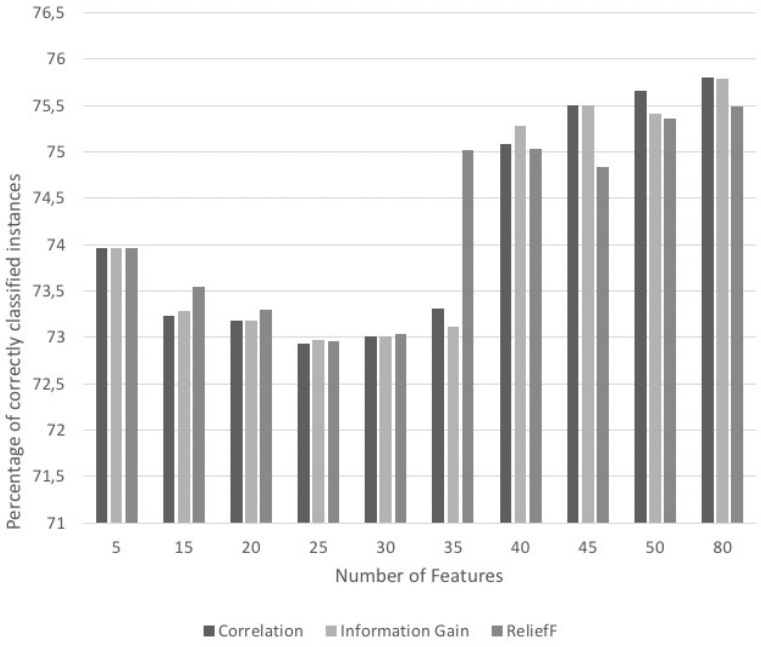

Figure 3. Percentage of correctly classified instances with varying number of attributes and attribute selection method (Naive Bayesian classifier, features from Java code)

2.2 Logistic Regression

First test case included changing of the number of iterations with assumptions: Number of features: 696. The quality of classification for logistic regression with the increase in the number of iterations decreased slightly. This is best seen in Fig. 4, where for the initial 10 and 20 iterations, more than 79.1% of correctly classified instances were achieved, and for each of the subsequent cases it was about 78.9%. With such small differences it can be said that changes in the records were negligible.

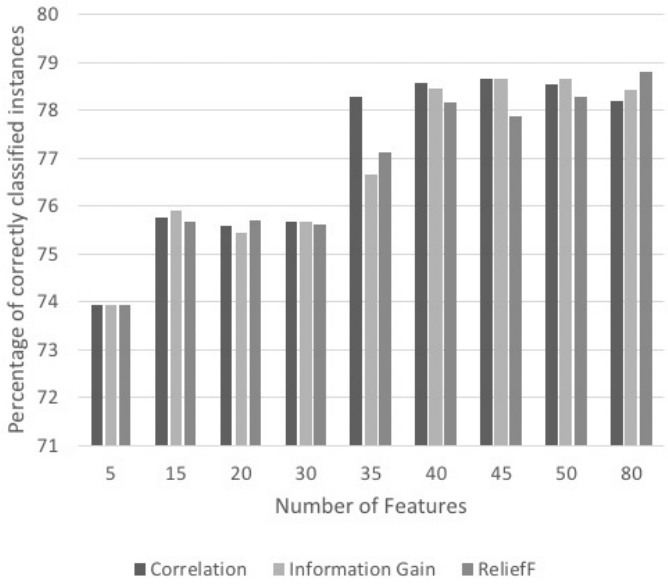

Figure 4. Percentage of correctly classified instances with varying number of attributes and attribute selection method (Logistic regression)

Second test case included changing of the number of features with assumptions: Number of iterations: 20. Figure 5 shows a deterioration in the quality of the classification with a number of features less than 35. For 35 and more features, the metric value was reached between 77% and 79%.

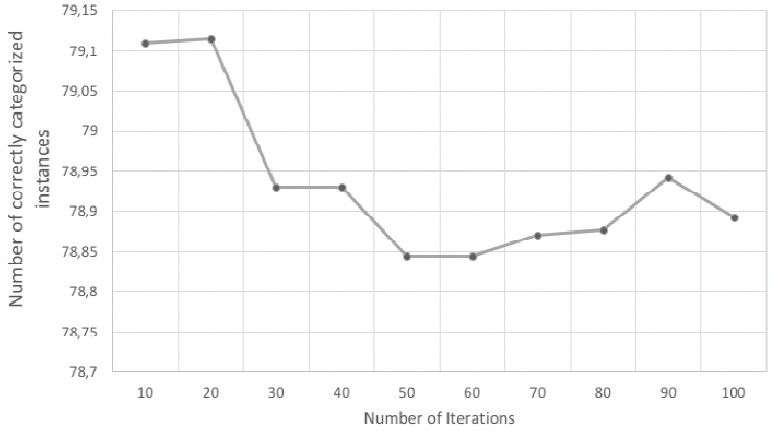

Figure 5. Percentage of correctly classified instances from the number of iterations (logistic regression)

2.5. K Nearest Neighbors

First test case included changing of the number of neighbors with assumptions: Number of features: 696. Euclidean distance function Fig. 6 show that the best results in percent of correctly classified instances, TP, FN and F, were achieved for k = 1, and these statistics deteriorated with increasing parameter k. When the number of neighbors is equal unity, there is a risk of overfitting, which is too much a fit of the model to the learning data.

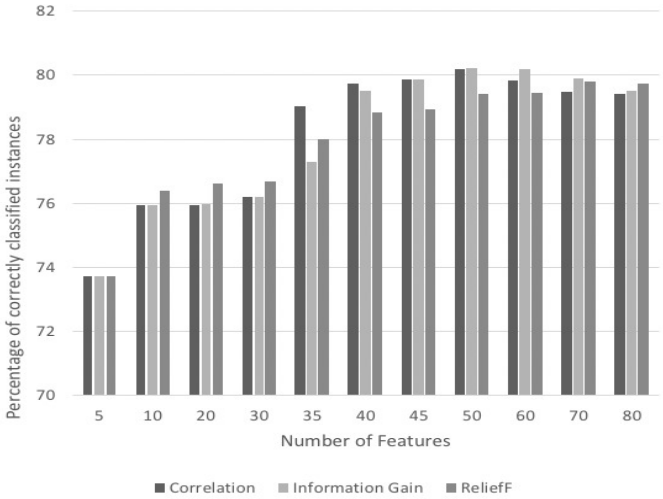

Figure 6. Percentage of correctly classified instances with changing number of attributes and attribute selection method (K-NN)

Second test case included changing of the number of features with assumptions: Number of neighbors: 1. Research on the number of features selected by 3 selection methods for K = 1 in the k nearest neighbors algorithm. With 50 features, the highest result was obtained in terms of properly classified instances - 80.1995% for information profit. Then, up to 80 traits, the value of this metric ranged from 78.3% to just over 80%. None of the selection methods was not the absolute best in this case.

2.6. SVM

First test case included Kernel Function with assumptions: Number of features: 696. Figure 7. shows that the best result by percentage of correctly classified instances was achieved for the Puk function (79.8579%), however, it is only slightly better than the Polykernel function (79.7879%), but the difference in time is large - for the Puck function this is over 60 seconds, and for Polykernel, 23 seconds. The Puk function also proved to be the best according to TP, FN, F-measure and error indicators. According to TN, at the high level of 98%, the function Normalized Polykernel won, but this should be combined with an extremely low TP rate - 55%.

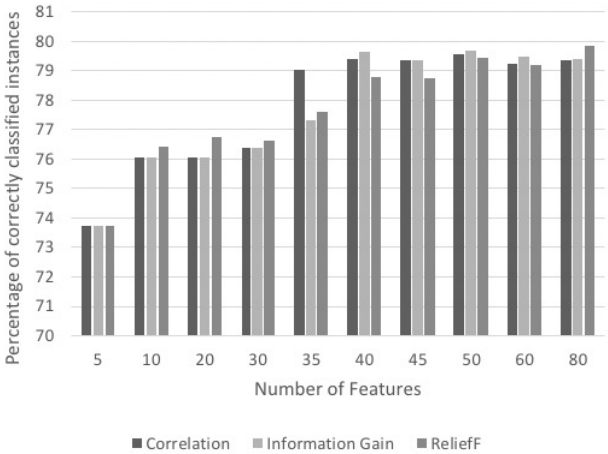

Figure 7. Percentage of correctly classified instances with varying number of attributes and attribute selection method (SVM)

Second test case included changing of the number of features. No clear favorite among the algorithms of feature selection is noticed. By reducing the number of features, a better result was not obtained according to the percentage of correctly classified instances than in sub-point a, where all 696 features were used. The highest result in this case is 79.8091% for 75 features selected by the ReliefF algorithm - it is worth noting that it is only 0.05 percentage point worse than the best result using 696 features, and the time decreased from 60 seconds to 23 seconds.

2.7. Summary of the Best Results

As part of this paper, selected aspects of the test outputs were verified by statistical tests. The tests carried out belong to the two groups: comparison of the medians of the occurrence of features in safe applications and malware and comparison of the accuracy of classifiers. Statistical methods (Shapiro-Wilk test and Lilliefors test) were checked for distribution normality for features whose size was to be compared in populations. After receiving a negative answer to the question about the normality of the distribution, the Mann-Whitney U-test was used to compare the population. It is used to answer the question whether observations in one population are greater than in the second population, which is interpreted as a comparison of medians in populations [13]. Based on [14] and [15], the McNemar test was selected to compare the accuracy of classifiers. Most of the features were dichotomous and did not have a normal distribution, which spoke for the use of the test.

Table 5. Accuracy of the classifiers

Classifier	Options	Correctness	Learning and testing time
Random Forest	max depth = 80 Iterations = 55 Features = 696	80.6662	74552.3246
Naive Bayes	Features = 10 (RelieF)	76.1217	9.5848
Logistic Regression	Iterations = 20 Features = 696	79.1152	2441.0845
k-NN	neighbors = 1 features = 696	80.3301	20979.0845
SVM	Kernel Function = PUK features = 696	79.8579	60714.3058

All statistical tests were carried out in the MATLAB environment. The best result in terms of percentage of correctly classified instances, equal to 80.6662% was obtained by random forest, with an iteration number equal to 55, maximum depth equal to 80 and 696 features. At the same time, the learning and testing time was the highest, at 75 seconds. It is worth comparing this result with the algorithm k nearest neighbors, which acted three times shorter, and correctly classified instances are lower by only 0.4 percentage points. Unfortunately, none of the algorithms exceeded 81 according to the discussed indicator. A comparison of the classification results times is shown in Table 5.

Table 6. Results of statistical surveys

	RF	NBC	KNN	LR	SVM
RF		←	=	←	←
NBC			↑	↑	↑
KNN				←	←
LR					↑
SVM					

Then the classifiers were statistically compared to the accuracy of the classifiers with the McNemar test. Table 6 shows the results of statistical surveys. The equality sign says that there were no grounds for rejecting the null hypothesis about the equality of the classifiers accuracy. The arrow indicates the classifier for which an alternative hypothesis has been adopted with greater accuracy than for the second classifier. According to statistical surveys, there are the following relationships between the accuracy of classifiers: Random forest algorithm and k nearest neighbors have the same accuracy, and both are more accurate than logistic regression, SVM and the naive Bayesian classifier. The naive Bayesian classifier has an accuracy lower than all other algorithms.

3. CONCLUSION AND FUTURE WORK

Paper is focused on the issue of malware detection for currently the most popular mobile system Android, using static analysis. In this thesis, an overview of Android malware analysis was presented, and a unique set of features was chosen that was later used in the study of malware classification. Five classification algorithms (Random Forest, SVM, K-NN, Nave Bayes, Logistic Regression) and three attribute selection algorithms were examined in order to choose those that would provide the most effective malware detection. The characteristics of malicious software were identified based on a collected set of applications. This analysis was conducted for features extracted from Java class code. It was determined which source of features provides higher quality of classification.

Research has been carried out to select the best classification algorithms for application, which is detection of malware on the Android platform, indication of the applications features of the highest usefulness in the classification of malware. Among the classification algorithms, the best proved to be: random forest and k nearest neighbors. They obtained the highest scores on the percentage of correctly classified instances (at the level of 80.3% - 80.7% for Java code). The accuracy of these classifiers was examined statistically and turned out to be the same. With the use of the naive Bayesian classifier and logistic regression, the classification accuracy was lower. It was noticed, to a small extent, the advantage of the existence of patterns of implementation of the onReceive method in malware, namely calling the function of starting or scheduling a new service.

The research on Java code has shown a strong presence of methods for manipulation on strings, as well as for downloading them outside of Java code. Such actions are manifestations of attempts to hide the real purpose of the application, i.e. obfuscation of the code. In addition, there has been a high use of methods that give access to and launch services (including system services). There is an increased presence of the method for data transfer over the HTTP protocol compared to secure applications, as well as methods for handling intentions, especially secret ones [20-23]. However, the quality of malware detection based on Java code proved to be low. None of the algorithms did exceed 81% of correctly classified instances. There are many reasons for this: the transformation and obfuscation of the code, the mechanism of reflection, manipulation on the chains of characters make the extraction of features a difficult task. Calling the API method can be implemented in several ways, and code transformation additionally increases the difficulty.

REFERENCES

[1] Kabakus, Abdullah Talha, & Ibrahim Alper Dogru, (2018), “An in-depth analysis of Android malware using hybrid techniques”, Digital Investigation.

[2] Verma, Prashant & Akshay Dixit, (2016), “Mobile Device Exploitation Cookbook”, Packt Publishing Ltd.

[3] Mohsen, Fadi, (2017), "Detecting Android Malwares by Mining Statically Registered Broadcast Receivers", Collaboration and Internet Computing (CIC), IEEE 3rd International Conference.

[4] Yerima, Suleiman Y, (2013), "A new android malware detection approach using bayesian classification", Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on. IEEE.

[5] Aafer, Yousra, Wenliang Du, & Heng Yin, (2013), "Droid apiminer: Mining api level features for robust malware detection in android", International conference on security and privacy in communication systems, Springer.

[6] Seo, Seung-Hyun, (2014), "Detecting mobile malware threats to homeland security through static analysis", Journal of Network and Computer Applications 38, pp: 43-53.

[7] Aung, Zarni, & Win, Zaw, (2013), "Permission-based android malware detection", International Journal of Scientific & Technology Research 2.3, pp: 228-234.

[8] Feizollah, Ali, (2017), "Androdialysis: Analysis of android intent effectiveness in malware detection", Computers & Security 65, pp: 121-134

[9] Mas' ud, Mohd Zaki, (2014), "Analysis of features selection and machine learning classifier in android malware detection", Information Science and Applications (ICISA), 2014 International Conference on. IEEE.

[10] Al Ali, Mariam, (2017), "Malware detection in android mobile platform using machine learning algorithms", Infocom Technologies and Unmanned Systems (Trends and Future Directions)", (ICTUS), 2017 International Conference on. IEEE.

[11] Li, Yiran, & Zhengping Jin, (2015), "An Android Malware Detection Method Based on Feature Codes.", Proceedings of the 4th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering.

[12] Nezhad Kamali, Maryam, Somayeh Soltani, & Seyed Amin Hosseini Seno, (2017), "Android malware detection based on overlapping of static features", 7th International Conference on Computer and Knowledge Engineering (ICCKE 2017), October 26-27, 2017, Ferdowsi University of Mashhad.

[13] B.H. Robbins, (2010), "Non Parametric Tests", B.H. Robbins Scholars Series, Dept. of Biostatistics, Vanderbilt University.

[14] Bostanci, Betul, & Erkan Bostanci, (2013), "An evaluation of classification algorithms using McNemars test", Proceedings of Seventh International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2012). Springer, India.

[15] Dietterich, Thomas G, (1998), "Approximate statistical tests for comparing supervised classification learning algorithms." Neural computation 10.7, pp: 1895-1923.

[16] La Polla, Mariantonietta, Fabio Martinelli, & Daniele Sgandurra, (2013), "A survey on security for mobile devices", IEEE communications surveys & tutorials 15.1, pp: 446-471.

[17] Tam, Kimberly, (2017), The evolution of android malware and android analysis techniques", ACM Computing Surveys (CSUR) 49.4, pp: 76

[18] Liang, Shuang, & Xiaojiang Du, (2014), "Permission-combination-based scheme for android mobile malware detection", Communications (ICC), 2014 IEEE International Conference on. IEEE.

[19] Saracino, Andrea, (2016), "Madam: Effective and efficient behavior-based android malware detection and prevention", IEEE Transactions on Dependable and Secure Computing.

[20] Linn, Cullen, & Saumya Debray, (2003), "Obfuscation of executable code to improve resistance to static disassembly", Proceedings of the 10th ACM conference on Computer and communications security, ACM.

[21] Enck, William, Machigar Ongtang, & Patrick McDaniel, (2009), "On lightweight mobile phone application certification", Proceedings of the 16th ACM conference on Computer and communications security. ACM.

[22] Vidas, Timothy, & Nicolas Christin, (2014), "Evading android runtime analysis via sandbox detection." Proceedings of the 9th ACM symposium on Information, computer and communications security. ACM.

[23] Burguera, Iker, Urko Zurutuza, & Simin Nadjm-Tehrani, (2011), "Crowdroid: behavior-based malware detection system for android", Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM.