M. Meier, D. Reinhardt, S. Wendzel (Hrsg.): Sicherheit 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 79

Comparative Evaluation of Machine Learning-based Malware Detection on Android

Sebastian Hahn[1], Mykolai Protsenko[2], Tilo Müller[3]

Abstract: The Android platform is known as the market leader for mobile devices, but it also has gained much attention among malware authors in recent years. The widespread of malware, a consequence of its popularity and the design features of the Android ecosystem, constitutes a major security threat currently targeted by the research community. Among all counter methods proposed in previous publications, many rely on machine learning algorithms based on statically extracted attributes from an app. Machine learning, which is also inspired by the developed field of desktop malware detection, has proven to be a promising approach for fighting Android malware. Many publications, however, rely on different data sets for different application attributes, rendering the comparison of them difficult. Furthermore, there exist attribute sets known from the desktop world which have not been ported to Android yet. In this paper, we aim to step towards filling this gap by assessing the effectiveness of the total number of 11 attribute sets, including those never evaluated on Android before, using a consistent data set of 10,000 apps. Our comparative evaluation provides a ranking for the single attribute sets according the detection performance they can reach, and suggests the most effective combination of all attributes.

Keywords: Android, Malware Detection, Machine Learning

1 Introduction

Due to the widespread of Android malware, a search for reliable detection methods remains an important research topic. In general, most detection measures can be classified as either being static or dynamic approaches. Static detection is usually facilitated by machine learning algorithms based on some static attributes of an application. In the numerous publications that emerged in the last years, researchers have proposed and evaluated many attribute sets for malware detection. At the same time, many static features are known to be valuable for machine learning based detection from the x86 domain, but have never been implemented for Android. In this paper we review those approaches and assess their effectiveness by performing an evaluation on a common dataset, which can bring us closer to the selection of the most optimal combination of static attributes for Android malware detection.

This paper is organized as follows: Section 2 gives some basic information about the Android system. Section 3 provides an overview of the attribute sets participated in our evaluation and covers details of their extraction implementation. The outcome of the

[1] Friedrich-Alexander University Erlangen Nuremberg, sebastian.hahn@informatik.stud.uni-erlangen.de
[2] Friedrich-Alexander University Erlangen Nuremberg, mykola.protsekno@fau.de
[3] Friedrich-Alexander University Erlangen Nuremberg, tilo.mueller@cs.fau.de

evaluation is given in Section 4 and the results are discussed in Section 5. Finally, we conclude in Section 6.

2 Background

This section covers the basics of the Android system. Readers familiar with Android can safely skip this section.

The Android mobile platform was created by the Open Headset Alliance with the goal to cope with the resource scarcity of hand-held devices and hardware plurality provided by numerous vendors. In a nutshell, Android is based on the Linux kernel enhanced by the core libraries and runtime environment, ontop of which the applications are executed. In the early versions of Android the runtime was represented by the Dalvik VM, a JIT supporting virtual machine with register-based bytecode which can be compiled from Java bytecode. Since Android version 5.0, Dalvik was replaced with ART, the more performant ahead of time compilation approach, generating machine code for each app from its bytecode at installation time. With both Dalvik and ART, the apps can also use native code by means of the Java native interface (JNI).

Within the code of any Android app, one can highlight the four main base classes which define the application components: Activities, Services, Broadcast Receivers and Content Providers. Each Activity corresponds to a single action performed by the app with a corresponding UI screen. The Services are used to perform some long-term background tasks and do not have any interface directly assigned to them. The Content Providers facilitate delivery of data by the app in response to the requests by other apps. The Broadcast Receivers help the application to define the reaction to the global events triggered either by the Android system or by installed apps. Examples for such events are a receipt of an SMS message or a low battery notification. The broadcast of such events, as well es invocation of Activities, relies on Intent objects, which can be considered as a description of a desired action.

Despite the recent replacement of the runtime environment, the structure of the app remained unchanged. The apps are shipped as zip-archived apk files, containing metadata, GUI layout definitions, Dalvik and native code, and other resources required by the app. The crucial metadata of the app is provided within the Android Manifest file and includes the description of the application components, the default Activity to be launched at the app startup, and the permissions requested by the apps.

The permissions play a crucial role in the Android security system. Each permission corresponds to a certain security-critical action, for instance dispatch of an SMS message. Each app declares the permissions it requires in its Manifest, and they are accredited by the user before app installation. Once being granted, an app may utilize its permissions at any time after.

3 Attributes and Implementation

All previously proposed solutions to the malware detection problem can roughly be classified in static, dynamic and hybrid approaches. The static ones analyze an app without its execution, the dynamic ones, on the opposite, aim to classify the app by monitoring its behavior at run time, and the hybrid approaches combine static and dynamic detection methods. In this paper, we focus on the static detection based on machine learning. A review of static attributes utilized in previously published classification approaches is provided in the following. An overview of the well-known dynamic and hybrid detection tools was performed by Neuner et al. [Ne14].

3.1 Manifest attributes

This subcategory of attributes is extracted from the Manifest which every apk has to contain. The Manifest is located in the main folder of the apk and contains various information regarding the application.

- **Permissions**: As mentioned in Section 2, Android permissions reflect the security-sensitive actions an app intends to make use of. Due to the fact that such actions are only accessible if the corresponding permissions have been granted, and the mandatory nature of their declaration, intuitively they can be very helpful in recognizing potentially malicious applications. Indeed, in many detection approaches known from the literature permissions play a crucial role [Sa13b, AZ13, Sa13a].
- **Features**: Features declare software and hardware properties on which an application depends. Unlike the permissions, the declaration of features is not necessary for an application to function properly and serves solely informational purposes, allowing to identify compatibility of an app with certain Android devices and system versions. The use of this attribute set was proposed by Santos et al. [Sa13a].
- **Intents**: This attribute is defined by the intents an app is using. Intents are used to either start an Activity, a Service, or to broadcast an event to all subscribed receivers. The use of Intents as attributes for malware detection was proposed by Arp et al. [Ar14]. An app components can only react to certain Intents if the corresponding `intent-filters` are declared in the Manifest. The filter declaration contains an action element, which describes what has to be performed in response to the received Intent or what global event has occurred. In this paper, we use the declared action as a representation of the Intent filters.
- **Application Components**: This feature set, utilized among other attributes by Arp et al. [Ar14], is defined by the application components, namely Activities, Services, and Broadcast Receivers, declared in the Android Manifest file. The Content Providers were not included in this set.

3.2 Code attributes

This set of attributes is extracted from the code contained in an app. If not stated otherwise, the following attributes refer to the bytecode of an app. The features derived from the native code are marked as such.

- **Used URLs**: This attribute composed from the URLs that are found inside the code of an app. The presence of certain URLs may help to identify malware since they can reveal command and control servers of botnets or resources to update and enhance malicious functionality. Since not every URL leads to a malicious website, we follow the example by Apvrille and Strazzere [AS12] and remove extremely frequently used URLs like the Android Market's URL, Google services, XML schemes and advertisement companies. Furthermore, the presence of URLs is taken into consideration regardless of the number of their occurrences.
- **Codesize**: The codesize feature is defined straight forward, it is the sum of the sizes of all codefiles in an apk. This feature was a part of the attribute set proposed by Apvrille and Strazzere [AS12].
- **Times of appearance of opcodes**:
  This attribute is based on the statistical properties of an app's code that was proposed by Santos et al. [Sa13a] to detect unknown x86 malware (PE). According to the authors, opcodes reveal significant statistical differences between malware and legitimate software. Following their example, for this attribute we compute the number of appearances of each opcode in the app's code.
- **Opcode sequences**: Similarly to the previous one, this attribute set is based on the occurrences of the bytecode instructions in an app's code, and was proposed for detection of Windows malware. Here, we apply the similar approach on the Android bytecode. In particular, we define attributes corresponding to the number of occurrences of the bytecode sequences of length two. Note that here only the instruction opcodes are considered, whereas the arguments of each instruction are discarded. Examples of the usage of such attribute for Android malware detection can be found in the works by Jerome et al. [Je14], Canfora et al. [CMV15], or Kang et al. [Ka13].
- **Presence of the native code**: Since the native code might be harder to analyze and is not supported by many analysis tools to the same extent as Dalvik bytecode, malicious apps may try to hide parts of their functionality in the native code. Furthermore, native code may be used to exploit vulnerabilities of the Android system, for instance to gain the root access to the device. Therefore, we add this attribute which reflects the presence of native libraries in the app. The use of the native code was also utilized as a feature for malware detection by Apvrille and Strazzere [AS12].

3.3 Other known attributes

This section provides an overview of the other attributes known form the literature, which do not fit in any of the categories above.

- **Presence of executables and zip-files in assets**: The use of this attribute set was suggested by Apvrille and Strazzere [AS12], motivated by the rational assumption that an attempt to hide data, e.g. an executable exploit, in the assets is a clear evidence for the app's malignity.
- **Statistical analysis attributes**: This statistical feature set was proposed by Tabish et al. [TSF09] for detection of x86 malicious files, not limited to executables only. In this paper we adapt this approach for Android malware detection, extracting the raw data from the files of the apk.
  Following the methodology described in the original paper, the content of each file is decomposed into blocks, which are used to form n-grams with various values of n. According to the authors [TSF09], the block size "plays a critical role in defining the accuracy", in this work it was set to 2000. The approach was evaluated for the same values of n as used by by Tabish et al., namely one, two, three and four.
  The feature set contains 13 items, which are computed exactly as described in the original paper, although other sources provide slightly different definitions of some values. The computation is performed for n-grams with n equals one, two, three, and four, resulting in a total number of 52 integer attributes in this set.

3.4 New attributes

Next we present the new features we propose in this paper, which to our best knowledge have never been used as attributes for Android malware detection.

- **Entropy of files in the resource-folder**: Previously we have introduced the attribute examining the presence of the executable or archived files in the assets of the app, originally proposed by Apvrille and Strazzere [AS12]. However, malicious payload can be hidden by more advanced means, e.g., using steganography. This attribute goes one step further in the search for suspicious contents of the apk. For this purpose we utilize entropy of the files in the resource folder as a measure of their 'structural order', with the expectation that unusual values will spot encrypted or otherwise alien content. To detect possible anomalies, we also calculate such statistical features of the entropy over all resource files as the arithmetic mean, the standard deviation and the empirical variance. On top of that we also take the maximum and minimum entropy values for each apk's resource file.
- **GUI Layout**: With this feature set we aim to recognize applications, similar by their visual appearance to the ones already known to the classifier. The attributes are extracted from the layout xml-files that define the structure of the app's GUI. These xml-files contain a hierarchy of layout items, for example LinearLayouts, Textboxes,

Buttons or Views. The features are defined as the total and the average number of each kind of Layout, Textbox, Button, and View in all xml-files of the apk.

- **Number of methods and number instructions per method**: This attribute should help the classifier to identify applications with similar code-structure as the known ones. For this purpose, from each apk we extract the following values: the number of methods in the code, the average number of opcodes in each method, and the standard deviation and empirical variance of the number of opcodes in each method. Note that the methods containing 3 or less opcodes are ignored, since we assume that those are getters or setters, or other short methods without relevant functionality.

3.5 Implementation

The extraction of the attributes described previously in this section was implemented ontop of the Androguard framework [DG11], which provided us with the necessary capabilities of processing the bytecode, as well as the Manifest and other xml files of the Android application package. For the classification of the samples we have utilized the WEKA tool [Ha09], which includes the classifiers that have demonstrated top performance in detecting Android Malware in previous studies: e.g., Random Forest, Bayesian Network, and K Nearest Neighbors.

4 Evaluation

This section is devoted to the practical evaluation of the attributes presented in Section 3. The evaluation contains the performance assessment of both single feature sets and combinations of the most promising attributes.

4.1 Dataset

The dataset we used for the evaluation was provided by Spreitzenbarth et al. and was collected within the MobileSandbox project [Sp13]. Out of the total number of obtained apps, we have randomly formed a dataset which contains 10,000 apps: 5,000 malicious samples and 5,000 benign ones.

For the purpose of the evaluation the dataset was randomly divided into a training- and a testset, in the proportion 80 to 20. Note that the testset does not contain any malware samples from the families included in the trainingset. This allows evaluation of the detection for previously unknown malware.

4.2 Performance Metrics

Next we describe the metrics of the detection quality which we utilized in our work to compare performance achieved by various classifiers with various attribute sets. The true

positive and true negative rates (TPR and TNR) correspond to the fraction of the correctly identified benign and malicious samples respectively. As an opposite, the values of false positives and false negatives (FPR and FNR) indicate the percentages of benign samples flagged as malicious and malicious samples assumed benign, respectively.

The true positive rate is sometimes called precision, meaning the ratio of the malware the classifier detected correctly amongst the total number of samples the classifier flagged malicious.

The accuracy can be considered 'the ratio of correct hits' and is defined as the sum of true positives and true negatives divided by the overall sample count in the test set. Sometimes instead of the false positive ratio its opposite is used, namely the specificity, as the ratio of true negatives to the total number of benign samples. Additionally, in our evaluation we include the time it took to build the classifier and the test time.

4.3 Single Feature Set Evaluation

This part of the evaluation aims to assess the detection performance of each single attribute set described previously in Section 3. The outcome is summarized in Table 1. According to the results, Android permissions are the best single predictor of the app's malignity, reaching the accuracy of about 96%. Very good classification quality was also achieved by opcode frequency, opcode sequences, and app components, with accuracy above 90%. For these and the most other attribute sets the best performance was shown by the Random Forest.

Such attributes as Intents, the presence of the native code or executable and archived files in the assets, as one could have expected, did not prove themselves as useful attributes on their own.

	Acc., %	TPR, %	FPR, %	Build Time, ms	Test Time, ms	Classifier
Permissions	96.02	96.00	3.95	6669	414	Random Forest
Opcode frequency	94.82	90.89	1.25	2196	321	Random Forest
Opcode sequences	94.82	90.29	0.65	10611	539	Random Forest
App components	94.27	91.25	2.7	199721	3628	Random Forest
Res. folder entropy	86.10	82.69	10.5	1606	95	Random Forest
Layout files	85.75	77.79	6.3	1359	233	Random Forest
Instr. per method	85.64	86.59	15.3	1337	108	Random Forest
Statistical analysis	79.92	75.44	15.6	2936	203	Random Forest
Codesize	76.12	79.34	27.1	3	394	KNN-5
Intents	74.24	50.82	3.26	3579792	3108	Random Forest
Features	52.63	98.75	92.92	258	169	Bayesian Network
Native code	51.74	08.70	5.25	8	7	Naive Bayes
Exec. and zip-files	50.26	98.40	97.85	8	11	Naive Bayes

Tab. 1: Ranking of the single attribute sets

4.4 Feature Combination Evaluation

As a second part of our evaluation, we have examined the performance of various combinations of the attribute sets. The results for the best of them are summarized in Table 2. The best performance with the accuracy over 97% was achieved by the combination of all attributes extracted from the Android Manifest, namely permissions, features, intents, and application components. Notably, we witnessed a slight accuracy increase if the features were excluded from this set.

In general, the detection performance based on permissions was improved if such attributes as the resource folder entropy, the opcode frequency or the layouts were added. On the other hand, the permissions performed better on their own than in combination with opcode sequences, application components, or statistical analysis attributes.

For the attribute set combinations the best classifier was also the Random Forest algorithm, showing the best performance in vast majority of cases including the top combinations.

	Acc., %	TPR, %	FPR, %	Build Time, ms	Test Time, ms	Classifier
Manifest attr. without feat.	97.35	96.95	2.25	108310	1793	Random Forest
Manifest attr.	97.30	96.65	2.05	105550	1755	Random Forest
Perm. and entropy	96.80	96.35	2.75	4744	477	Random Forest
Perm. and oc freq.	96.40	93.69	0.9	2600	348	Random Forest
Perm. and layout	96.30	95.15	2.55	3979	350	Random Forest
Perm., layout, oc freq.	96.30	93.54	0.95	2682	350	Random Forest
Perm. and oc seq.	95.57	96.45	5.3	172122	9	J48
Perm., app comp., oc freq.	95.35	91.64	0.95	17962	856	Random Forest
Perm. and stat. analysis	95.22	94.05	3.6	3186	393	Random Forest

Tab. 2: Ranking of the attribute set combinations by accuracy

5 Discussion

The results presented in the previous section indicate that the best performing attributes can be derived rather from an app's metadata stored in the Android Manifest than from the actual code of an app, although the code based attributes like opcode frequency and sequences also performed quite well. The reason is the nature of Android permissions which more or less precisely declare all the security-critical actions an app can possibly perform.

One the one hand, the Manifest attributes can provide a reliable and easy way to detect malicious apps, which does require neither static nor dynamic analysis of the code. On the other hand, the actual behavior of an app can be theoretically only derived from its code, a fact that inevitably leads to the possible drawbacks of the metadata-based detection: either a high false-positive rate, or high risk of missing truly malicious apps, i.e., low true positive

rate. Indeed, Table 1 shows, that the permissions on their own tend to have a comparatively high false positive rate.

The code attributes such as opcode frequencies and sequences, on the contrary, seem to be tightly bound with the actual behavior of an app defined by its code base, and therefore show much lower false positive rate as single attribute sets. Furthermore, in combination with permissions, opcode frequencies prove themselves as the most conservative attributes, with the least false positive rate.

The practical implementation of the malware detection tool based on our results has two options. First, it can be performed on device, which means an autonomous detection without Internet connection is possible, at cost of higher runtime overhead and the need of having the up-to-date trained classifier stored locally. Second, one could submit either the apk file itself, or the extracted attribute vector to a cloud for the analysis. Since in general it is associated with less overhead and it does not expose the classifier to the public, the second option seems to be more reasonable.

From the practical point of view, the main advantage of the strictly Manifest-based attributes would be the absence of the need to analyze code or other parts of the app, which would result in a much lower overhead. However, in our opinion, this advantage is outweighted by the disadvantage of having higher false positive rate, which is a quite important feature, as its high value would restrict users from installing benign applications.

Furthermore, for a practical anti-malware tool a test time would become a parameter of a higher importance. In our case, the best performing classifier Random Forest, despite having a high time consumption for training, can perform classification in relatively short amount of time.

6 Conclusion

In this paper we have examined different attributes according to their performance for machine learning based detection of Android Malware. In total, 11 previously known and new feature sets were evaluated as stand-alone attributes and in various combinations. Summarizing the results, we can point out that the leadership among the single feature sets belongs to Android permissions, which achieved accuracy of about 96%. For the combination of the attributes, the best performance was achieved by the following Manifest attributes: permissions, intents, and application components with over 97% accuracy. In both cases, the best accuracy was reached by the Random Forest classifier. Therefore, our results suggest the use of this combination either for a purely static detection approach, or as a 'reinforcement' of dynamic or hybrid tools like MobileSandbox [Sp13].

References

[Ar14] Arp, Daniel; Spreitzenbarth, Michael; Hübner, Malte; Gascon, Hugo; Rieck, Konrad; Siemens, CERT: Drebin: Effective and explainable detection of android malware in your pocket. In: Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS). 2014.

[AS12] Apvrille, Axelle; Strazzere, Tim: Reducing the Window of Opportunity for Android Malware Gotta catch'em all. Journal in Computer Virology, 8(1-2):61–71, 2012.

[AZ13] Aung, Zarni; Zaw, Win: Permission-based Android malware detection. International Journal of Scientific and Technology Research, 2(3):228–234, 2013.

[CMV15] Canfora, Gerardo; Mercaldo, Francesco; Visaggio, Corrado Aaron: Mobile Malware Detection using Op-code Frequency Histograms. In: SECRYPT 2015 - Proceedings of the 12th International Conference on Security and Cryptography, Colmar, Alsace, France, 20-22 July, 2015. pp. 27–38, 2015.

[DG11] Desnos, Anthony; Gueguen, Geoffroy: Android: From Reversing to Decompilation. In: Black Hat Abu Dhabi. 2011.

[Ha09] Hall, Mark; Frank, Eibe; Holmes, Geoffrey; Pfahringer, Bernhard; Reutemann, Peter; Witten, Ian H.: The WEKA Data Mining Software: An Update. ACM SIGKDD Explorations Newsletter, 11(1):10–18, November 2009.

[Je14] Jerome, Q.; Allix, K.; State, R.; Engel, T.: Using opcode-sequences to detect malicious Android applications. In: Communications (ICC), 2014 IEEE International Conference on. pp. 914–919, June 2014.

[Ka13] Kang, Byeongho; Kang, BooJoong; Kim, Jungtae; Im, Eul Gyu: Android Malware Classification Method: Dalvik Bytecode Frequency Analysis. In: Proceedings of the 2013 Research in Adaptive and Convergent Systems. RACS '13, ACM, New York, NY, USA, pp. 349–350, 2013.

[Ne14] Neuner, Sebastian; der Veen, Victor Van; Lindorfer, Martina; Huber, Markus; Merzdovnik, Georg; Mulazzani, Martin; Weippl, Edgar R.: Enter Sandbox: Android Sandbox Comparison. In: Proceedings of the IEEE Mobile Security Technologies workshop (MoST). IEEE, IEEE, 05/2014 2014.

[Sa13a] Santos, Igor; Brezo, Felix; Ugarte-Pedrero, Xabier; Bringas, Pablo G: Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences, 231:64–82, 2013.

[Sa13b] Sanz, Borja; Santos, Igor; Laorden, Carlos; Ugarte-Pedrero, Xabier; Bringas, Pablo Garcia; Álvarez, Gonzalo: Puma: Permission usage to detect malware in android. In: International Joint Conference CISISÂ12-ICEUTE´ 12-SOCO´ 12 Special Sessions. Springer, pp. 289–298, 2013.

[Sp13] Spreitzenbarth, Michael; Freiling, Felix; Echtler, Florian; Schreck, Thomas; Hoffmann, Johannes: Mobile-sandbox: Having a Deeper Look into Android Applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. SAC '13, ACM, New York, NY, USA, pp. 1808–1815, 2013.

[TSF09] Tabish, S Momina; Shafiq, M Zubair; Farooq, Muddassar: Malware detection using statistical analysis of byte-level file content. In: Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics. ACM, pp. 23–31, 2009.