Automated Static Code Analysis for Classifying Android Applications Using Machine Learning

Asaf Shabtai Yuval Fledel Yuval Elovici
Department of Information Systems Engineering and,
Deutsche Telekom Laboratories at Ben-Gurion University
Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
{shabtaia, fledely, elovici}@bgu.ac.il

Abstract—**In this paper we apply Machine Learning (ML) techniques on static features that are extracted from Android's application files for the classification of the files. Features are extracted from Android's Java byte-code (i.e., .dex files) and other file types such as XML-files. Our evaluation focused on classifying two types of Android applications: tools and games. Successful differentiation between games and tools is expected to provide positive indication about the ability of such methods to learn and model Android benign applications and potentially detect malware files. The results of an evaluation, performed using a test collection comprising 2,285 Android .apk files, indicate that features, extracted statically from .apk files, coupled with ML classification algorithms can provide good indication about the nature of an Android application without running the application, and may assist in detecting malicious applications. This method can be used for rapid examination of Android .apks and informing of suspicious applications.**

Keywords- Mobile Devices, Machine Learning, Malware, Security, Android, Static analysis

I. INTRODUCTION

Smart phones have evolved from simple mobile devices into sophisticated yet compact minicomputers. Designed as open, programmable, networked devices, smart phones are susceptible to various malware threats such as viruses, Trojan horses, and worms, all of which are well-known from desktop platforms. These devices enable users to browse the Internet, receive and send emails, SMSs, and MMSs, exchange information with other devices, and activate various applications, which make these devices potential attack targets [1][2].

A compromised smart-phone can inflict severe damages to both users and the cellular service provider. Malware on a smart-phone can make the phone partially or fully unusable; cause unwanted billing; steal private information; or infect every name in a user's phonebook [3]. Possible attack vectors into smart phones include: Cellular networks, Internet connections (via Wi-Fi, GPRS/EDGE or 3G network), USB and other peripherals [4].

Among the most significant smart-phone operating systems that have arisen recently is Google's Android. Android[1] is a comprehensive software framework for smart mobile devices and it includes an operating system, middleware and a set of key applications. Our security assessment of the Android framework [5] indicates that malware penetration to the device is likely to happen, not only at the Linux level but in the application level (Java) and thus exploring methods for protecting the Android framework is essential. A collection of applicable security solutions for mobile devices in general and Android in particular is presented in [6]-[8].

Mobile OS makers are now very much concerned with the security challenges that PCs have been facing down through the years [9]. The increasing number of attacks on mobile platforms along with the increasing usage has led many security vendors and researchers to propose a variety of security solutions for mobile platforms. As a case in point, Symbian and Google have designed their operating systems to run applications only in specialized sandboxes, minimizing the capability of malware to spread [10]. Additionally, common desktop-security solutions are being downsized to mobile devices. As a case in point, Botha et al. [11] analyzed common desktop security solutions and evaluated their applicability to mobile devices. When porting a security solution, such as antivirus software, the limited availability of resources (power source, CPU and memory) of such devices should be taken into account. In addition, most antivirus detection capabilities depend on the existence of an updated malware signature repository; therefore antivirus users are not protected whenever an attacker releases a previously un-encountered malware. Some malware instances may target a specific and relatively small number of mobile devices (e.g., for extracted confidential information or track owner's location) and will therefore take quite a time until they are discovered.

A robust application signing and certification mechanism, which relies mostly on manual code inspection, was integrated into Symbian's operating system and was proven highly effective in reducing malware attacks. Apple also requires all applications to pass a review process that checks amongst other things for malware. However, malware writers can still evade manual code inspection as shown in [12]. Therefore, researchers are currently looking at various alternatives that capture application semantics in order to make smarter security decisions without relying on manual code inspection. Since Android was released, several attempts to statically investigate the code of Android applications were published. Schmidt et al. [13] evaluated a

[1] http://www.android.com

978-0-7695-4297-3/10 $26.00 © 2010 IEEE
DOI 10.1109/CIS.2010.77

framework for static function call analysis and performed a statistical analysis on the function calls used by native applications. Chaudhuri [14] presented a formal language for describing Android applications and data flow among application's components. This formal language can be used for statically analyzing Android applications and data flow between applications and comparing those with security specifications defined in the application's manifest.

In this paper we propose the detection of unknown malware instances on Android-based mobile devices by applying Machine Learning techniques on static features that are extracted from Android's application files. Each application in Android is packaged in an .apk archive which is similar to standard Java .jar files and comprise of both code and resources. Android .apk files encapsulate valuable information that can help in understanding an application's behavior. The information includes requested permissions, framework methods called up by the application, framework classes used by the application, user interface widgets, etc. The primary goal of the study is to find the optimal mix of: a classification method, feature selection method and the number of extracted features that yields the best performance in accurately detecting new malware on Android. While most of previous studies extracted features that are based on byte sequence *n*-grams, in this study we evaluate the use of meaningful features from the Android application files such as the requested permissions, framework methods and classes used by the application.

Two facts support the applicability of such method for detecting malicious code on Android: 1) the proposed features can be extracted from any given Android application; and 2) the vast majority of the proposed features cannot be effectively obfuscated by Android applications. This is due to the side-effects of the installation process. In particular, information from the application manifest is only used in the installation process, leaving the application unable to alter it after the installation.

The rest of the paper is organized as follows. We start in section 2 with a survey of previous relevant studies. Section 3 describes the methods we used. Next, in sections 4 we present the evaluation and the evaluation results. Finally section 5 discusses the results and future work.

II. Related Works

Machine Learning (ML) classification algorithms were employed to automate and extend the idea of heuristic-based detection methods to enable better detection of unknown malware. In these methods, the application file is represented by static features that are extracted from the file (e.g., byte sequence *n*-grams or PE header features) and classifiers are applied to learn patterns in the code in order to classify new (unknown) files. Recent studies have shown that these methods yield very accurate classification results [15]. Applying static analysis for detecting malicious software consumes much less resources and time, compared to dynamic analysis in which features are extracted from the system while the application is running, and does not require to execute the application for the detection.

Over the past nine years, several studies have focused on the detection of unknown malware on personal computers based on its binary code content. The authors of [16] were the first to introduce the idea of applying ML methods for the detection of different malware based on their respective binary codes. Three different types of features were tested: program header (Portable Executable section), string features (meaningful plain-text strings that are encoded in programs files such as "windows", "kernel", "reloc" etc.) and byte sequence features, to which they applied four classifiers: a signature-based method (antivirus), Ripper – a rule-based learner, Naïve Bayes, and Multi-Naïve Bayes. The study found that all of the ML methods were more accurate than the signature-based algorithm. Abou-Assaleh et al. [17] introduced a framework that uses the Common N-Gram (CNG) method and the *k*-nearest neighbor (KNN) classifier for the detection of malware. For each malicious and benign class a representative profile was constructed. A new executable file was compared with the profiles of malicious and benign classes, and was assigned to the most similar. Kotler and Maloof [18] used a collection of 1971 benign and 1651 malicious executables files. N-grams were extracted and 500 features were selected using the Information Gain measure. The vector of *n*-gram features was binary, presenting the presence or absence of a feature in the file. In their experiment, they trained several classifiers: IBK (KNN), a similarity-based classifier called the TFIDF classifier, Naïve Bayes, SVM, and Decision Tree (J48). The last three of these were also boosted. In the experiments, the four best-performing classifiers were Boosted J48, SVM, Boosted SVM and IBK.

Recently, Moskovitch et al. [19] published the results of a study that used a collection of more than 30,000 files, in which the files were represented by byte sequence *n*-grams. In other studies the representation of executable files was done by OpCode sequence *n*-gram expressions, through streamlining an executable into a set of OpCodes [20].

III. Detection Method

The overall process of classifying unknown files using ML methods is divided into two subsequent phases: training and testing. In the first phase, a *labeled* set of .apk files (the training-set) is provided to the system. Each file is then parsed, and a vector representing each file is extracted based on a predetermined vocabulary. The representative vectors of the files in the training-set and real (known) labels are the input for a learning algorithm. By processing these vectors, the learning algorithm generates a classification model.

Next, during the testing phase, a test-set collection of .apks that did not appear in the training-set, are classified by that classifier. Each file in the test-set is first parsed and the representative vector is extracted (based on the same vocabulary used in the training phase). Based on this vector, the classifier will attempt to predict the class of the file. The performance of the classifier is evaluated by extracting standard accuracy measures that compare the prediction with the real label of the file. Thus, it is necessary to know the real class of the .apk files in the test-set.

Since no malicious applications are yet available for Android, we evaluated the proposed method ability to differentiate between game and tool application files. Successful differentiation between games and tools is expected to provide positive indication about the ability of such methods to learn and model an Android benign application files and potentially detect malware files.

The goal of our work was to explore methods of using data mining techniques in order to create accurate detectors for new (unseen) Android .apk files. We evaluated the following classifiers: *Decision Tree* (DT) [21], *Naïve Bayes* (NB) [22], *Bayesian Networks* (BN) [23], *PART* [24], *Boosted Bayesian Networks* (BBN) [25], *Boosted Decision Tree* (BDT) [25], *Random Forest* (RF) [26], *Voting Feature Intervals* (VFI) [27].

In ML applications a large number of extracted features present several problems such as: misleading the learning algorithm, over-fitting, and increasing model complexity. Applying feature selection in a preparatory stage enables to generate a more efficient detector, with a faster detection cycle. Nevertheless, reducing the amount of features should be performed while preserving a high level of accuracy.

We used the *filters* approach for feature selection, due to its fast execution time and its generalization ability [20]. Under a filter approach, an objective function evaluates features by their information content and estimates their expected contribution to the classification task. We evaluated three feature selection measures: *Chi-Square* (CS) [28], *Fisher Score* (FS) [29] and *Information Gain* (IG) [30]. These feature selection methods, using a specific metric, compute and return a score for each feature individually.

A problem was raised when we had to decide how many features we would choose for the classification task from the feature selection algorithms' output ranked lists. In order to avoid any bias by selecting an arbitrary number of features, we used, for each feature selection algorithm, six different configurations: 50, 100, 200, 300, 500 and 800 features that were ranked the highest out of all the featured ranked by the feature selection algorithms.

IV. Evaluation

A. Research Questions

We evaluated the capability of the proposed method to classify Android .apk files through a set of experiments, aimed at answering the following questions:

1. Is it possible to detect *unknown instances* of known Android application types?
2. Which *classifier* is most accurate in detecting malware on Android devices: DT, NB, BN, BBN, BDT, PART, RF or VFI?
3. Which *top-selection* (number of extracted features) and *feature selection method* yield the most accurate detection results: 50, 100, 200, 300, 500 or 800 top features selected using CS, FS or IG?
4. What are the specific features that yield maximal detection accuracy?

In order to perform the comparison between the various detection algorithms and feature selection schemes, we employed the following standard metrics: *the True Positive Rate* (TPR) measure, which is the proportion of positive instances (i.e., feature vectors of games) classified correctly; *False Positive Rate* (FPR), which is the proportion of negative instances (i.e., feature vectors of tools) misclassified; and the total *Accuracy*, which measures the proportion of absolutely correctly classified instances, either positive or negative. Additionally, we used the *Receiver Operating Characteristic* (ROC) curve and calculated the *Area-Under-Curve* (AUC). The ROC curve is a graphical representation of the trade-off between the TPR and FPR for every possible detection cut-off.

B. Creating the Dataset for the Experiments

Since no standard dataset was available for this study, we had to gather our own dataset. Our evaluation focuses on classifying two types of Android applications: tools and games. We collected free .apk files available on the Android Market (a Google-proprietary service which allows browsing and downloading of applications that were published by different developers). A total of 2,285 .apk files were collected. We categorized the applications according to the Market classification which results in 407 games and 1878 tools. More than 22,000 features were initially extracted from the file collection using an .apk feature extractor which we developed specifically for this task. Extracted features included:

1. *apk features*
   Each .apk is a zip archive that is similar to standard Java jar files and it holds all code and non-code resources (e.g., images, manifest) for the application. Thus, the first type of features is based on general zip file properties. Examples of such features include .apk size, number of zip entries, number of files for each file type, common folders and more.
2. *XML features*
   An .apk file contains several XML files that are used by the application during the installation and/or activation. For example, the AndroidManifest.xml is used for declaring permissions required by the application. Android uses a proprietary binary xml format which was designed to ease parsing on the device, and reduce required space. Since standard parsers could not be applied, we implemented our own Android XML parser. Examples of XML-based features are:
   - Count of xml elements, attributes, namespaces, and distinct strings
   - Features for each element name (e.g., count of distinct actions, distinct categories)
   - Features for each attribute name
   - Features for each attribute type (e.g., number, string)
   - Used permissions in the Android Manifest

3. *dex features*

   Android uses a proprietary format for Java bytecode called dex (Dalvik Executable). In order to extract meaningful features from .dex files we developed a .dex file “dissasembler”. This parser can transform contents of the .dex file into standard features (e.g., strings, types, classes, prototypes, methods, fields, static values, inheritance, opcodes). Examples for outputs of such transformations are a Boolean for each method in the framework -- method is used or not, and a Boolean for every type in framework -- type is used or not.

As a preprocessing stage, we removed useless features (e.g., features having only one value for all files) and features that can be used to identify a specific file (e.g., having only two values, but one value exists only on one file) resulting in 9,898 remaining features. Next, we applied three feature selection methods, Information Gain, Fisher Score and Chi-Square, in a preparatory stage to select the top 50, 100, 200, 300, 500 and 800 features.

C. *Experiments and Results*

In the experiment we wanted to answer the four research questions presented in section IV. According to these questions, we wanted to identify the best settings of the classification framework that is determined by a combination of: (1) the top-selection of features (50, 100, 200, 300, 500 or 800); (2) the feature selection method (CS, FS or IG); and (3) the classifier (DT, NB, BN, BBN, BDT, PART, RF or VFI) to distinguish between games and tools applications. The evaluation performed in a 10-fold cross validation format repeated 5 times for all the combinations of the optional settings. Note that the files in the test-set were not included in the training set presenting unknown files to the classifiers.

Figure 1 depicts the accuracy and FPR for each classification algorithm (averaged over all iterations, folds and feature selection algorithms). The results show that the two boosted algorithms (BDT and BBN) performed better than all other classifiers, while the Boosted Bayesian Networks outperformed all classifiers.

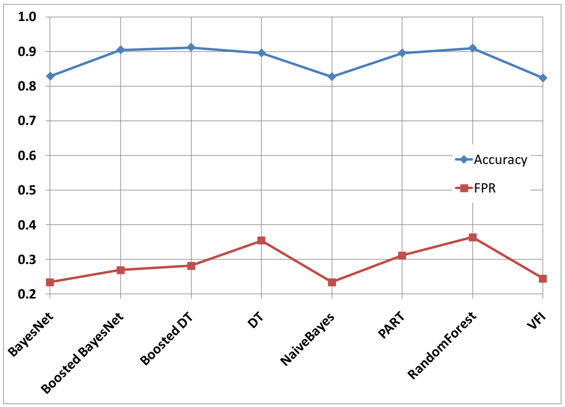

Figure 1. Mean accuracy and FPR of each classifier.

Table I depicts the accuracy, FPR, TPR and AUC for each combination of feature selection and the number of selected top features (averaged over all iterations, folds and classifiers). The results show that all three feature selection algorithms yielded similar results, where the performance was better with large number of features (i.e., 500 and 800 top features).

TABLE I. MEAN ACCURACY AND FPR FOR EACH COMBINATION OF FEATURE SELECTION AND THE NUMBER OF SELECTED TOP FEATURES

		ChiSquare	**FisherScore**	**InfoGain**
Accuracy	**50**	0.839	0.847	0.856
	100	0.861	0.868	0.871
	200	0.877	0.879	0.880
	300	0.881	0.882	0.882
	500	0.888	0.885	0.890
	800	**0.891**	0.884	**0.890**
FPR	**50**	0.336	0.349	0.308
	100	0.303	0.293	0.288
	200	0.283	0.270	0.279
	300	0.283	0.268	0.269
	500	0.283	0.269	0.279
	800	**0.254**	0.261	**0.260**

Table II depicts the averaged accuracy, FPR and AUC for the top two combinations of feature selection and the number of selected top features. Boosted BN trained on the top 800 features, selected using Chi-Square and Information Gain outperformed all other settings.

TABLE II. MEAN ACCURACY, FPR AND AUC OF THE BEST CONFIGURATIONS

Configuration	**Accuracy**	**FPR**	**AUC**
Boosted BN ChiSquare 800	0.922	0.190	0.945
Boosted BN InfoGain 800	0.918	0.172	0.945

Finally, we examine the list of ranked features. The top ranked 20 features are:

- used-method-java-lang-reflect-ArraynewInstance
- used-method-java-util-Random<init>
- used-method-java-util-RandomnextInt
- used-method-android-graphics-CanvasdrawBitmap
- used-method-android-view-SurfaceHolderlockCanvas
- used-method-android-view-SurfaceHolderunlockCanvasAndPost
- used-method-android-view-MotionEventgetAction
- used-method-android-view-MotionEventgetY
- used-method-android-graphics-CanvasdrawRect
- used-type-android-graphics-Canvas
- used-method-android-graphics-Paint<init>
- apk-size
- used-type-java-util-Random
- used-type-android-view-MotionEvent
- used-method-android-view-MotionEventgetX
- used-method-android-content-ContextgetResources
- used-type-android-graphics-Paint
- apk-folder-res-raw
- file-ext-wav
- used-method-android-graphics-CanvasdrawText

This list indicates that most important features for the classification are extracted from the .dex files of the application. In addition, these features indicate mainly usage of randomization and graphics methods/packages that are likely to be used by games.

V. Discussion and Conclusions

In this paper we used features extracted from Android application (.apk) files. The extracted data is used as features during a classification process of the applications. Since no malicious applications are yet available for Android, our evaluation focused on classifying two types of applications: tools and games. We performed an evaluation using a collection comprising 2,850 games and tools. The results show that the combination of Boosted Bayesian Networks and the top 800 features selected using Information Gain yield an accuracy level of 0.918 with a 0.172 FPR.

We suggest that the high FPR is due to the initial classification of the applications. We used the classification as appear in the Android Market; however some applications are not classified correctly. For example, many entertainment applications which are tagged as tools are more similar semantically to the games class.

Features, extracted statically from .apk files, coupled with Machine Learning classification can provide good indication about the nature of an .apk file without running it, and may assist in the detection of malicious applications. The most important features for the detection are extracted using our .dex file parser that can transform contents of the .dex file into standard features (e.g., strings, types, classes, methods, fields, static values, inheritance, opcodes). A more advanced usage of such tool is to merge two .dex files. This may be used in order to inject malicious payload into a benign .apk, thereby creating a Trojan horse. Consequently, an attacker can perform a Man-In-The-Middle attack by injecting malicious code while a user, connected to an un-trusted wireless network, is downloading an .apk file. We plan to further investigate the applicability of such attack.

References

[1] N. Leavitt, "Mobile phones: the next frontier for hackers?" Computer, vol. 38(4), 2005, pp. 20-23.

[2] D.H. Shih, B. Lin, H.S. Chiang, M.H. Shih, "Security aspects of mobile phone virus: a critical survey," Industrial Management & Data Systems, vol. 108(4), 2008, pp. 478-494.

[3] M. Piercy, "Embedded devices next on the virus target list," IEEE Electronics Systems and Software, vol. 2, 2004, pp. 42–43.

[4] J. Cheng, S.H. Wong, H. Yang, S. Lu, "SmartSiren: virus detection and alert for smartphones," Proc. International Conference on Mobile Systems, Applications and Services, 2007.

[5] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, "Google Android: A State-of-the-Art Review of Security Mechanisms," CoRR abs/0912.5101, 2009.

[6] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, C. Glezer, "Google Android: A Comprehensive Security Assessment," IEEE Security and Privacy, vol. 8(2), March/April 2010, pp. 35-44.

[7] A. Shabtai, Y. Fledel, Y. Elovici, "Securing Android-Powered Mobile Devices Using SELinux," IEEE Security and Privacy, vol. 8(3), May/June 2010, pp. 36-44.

[8] A. Shabtai, U. Kanonov, Y. Elovici, "Intrusion Detection on Mobile Devices Using the Knowledge Based Temporal-Abstraction Method," Journal of Systems and Software, vol. 83(8), 2010, pp. 1524-1537.

[9] D. Muthukumaran, et al. "Measuring integrity on mobile phone systems," Proc. ACM Symposium on Access Control Models and Technologies, 2008.

[10] G. Lawton, "Is It Finally Time to Worry about Mobile Malware?," Computer, vol. 41(5), 2008, pp. 12-14.

[11] R.A. Botha, S.M. Furnell, N.L. Clarke, "From desktop to mobile: Examining the security experience," Computer & Security, vol. 28, 2009, pp. 130-137.

[12] N. Seriot, "iPhone Privacy," Black Hat DC, Virginia, USA, 2010.

[13] A.D. Schmidt, et al. "Enhancing Security of Linux-based Android Devices," Proc. International Linux Kongress, 2008.

[14] A. Chaudhuri, "Language-Based Security on Android," Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS), 2009, pp. 1–7.

[15] A. Shabtai, R. Moskovitch, Y. Elovici, C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," Information Security Technical Report, vol. 14(1), 2009, pp.16-29.

[16] M. Schultz, E. Eskin, E. Zadok, S. Stolfo, "Data mining methods for detection of new malicious executables. Proc. IEEE Symposium on Security and Privacy, 2001.

[17] T. Abou-Assaleh, N. Cercone, V. Keselj, R. Sweidan, "N-gram Based Detection of New Malicious Code," Proc. Annual International Computer Software and Applications Conference, 2004.

[18] J.Z. Kolter, M.A. Maloof, "Learning to detect malicious executables in the wild," Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006, pp. 470–478.

[19] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, Y. Elovici, "Unknown Malcode Detection via Text Categorization and the Imbalance Problem," Proc. IEEE Intelligence and Security Informatics, Taiwan, 2008.

[20] R. Moskovitch, et al. "Unknown Malcode Detection Using OPCODE Representation," Proc. European Conference on Intelligence and Security Informatics, 2008.

[21] J.R. Quinlan, "C4.5: Programs for machine learning," Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1993.

[22] S. Russel, P. Norvig, Artificial Intelligence: A modern approach. Prentice Hall, 2002.

[23] J. Pearl, Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kauhmann, 1988.

[24] T. Mitchell, Machine Learning, McGraw-Hill, 1997.

[25] Y. Freund, R.E. Schapire, "A brief introduction to boosting," International Joint Conference on Artificial Intelligence, 1999.

[26] T.K. Ho, "Random Decision Forest," Proc. International Conference on Document Analysis and Recognition ,1995, pp. 278-282.

[27] G.D. Guvenir, "Classification by Voting Feature Intervals," Proc. European Conference on Machine Learning, 1997, pp. 85-92

[28] I.F. Imam, R.S. Michalski, L. Kerschberg, "Discovering Attribute Dependence in Databases by Integrating Symbolic Learning and Statistical Analysis Techniques," Proc. of the AAAI-93 Workshop on Knowledge Discovery in Databases, 1993.

[29] T. Golub, et al. "Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring," Science, vol. 286, 1999, pp. 531-537.

[30] C.E. Shannon, "The mathematical theory of communication," The Bell system Technical Journal, vol. 27(3), 1948, pp. 379–423.