Journal of Computers Vol. 30 No. 4, 2019, pp. 151-158
doi:10.3966/199115992019083004014

A Detection System of Android Malware Based on SVM Algorithm

Lian-Fen Huang[1], Chao-Lin Ye[1], Chao Feng[1*], Han-Bo Li[2], Ying-Min Zhang[1]

[1] Department of Communication Engineering, Xiamen University, Xiamen, China

[2] Department of Electronic Engineering, Xiamen University, Xiamen, China
chaof@xmu.edu.cn

Received 13 January 2019; Revised 14 January 2019; Accepted 12 March 2019

Abstract. In this paper, we propose a new detection system of android malware, a lightweight system combining static detection and Support Vector Machine algorithm (SVM). This system adopts static analysis to gather features of an application directly. These features are mapped to a vector space. An app corresponding a vector in the vector space, the SVM algorithm use these vectors to train a model which can indicate whether an android app has suspicious behaviors. To evaluate this system, we use a train set of 4500 benign applications and 6500 malicious applications to train model and acquire a *True Positive Rate* (TPR) of 97.96% and *False Positive Rate* (FPR) of 0.84% with a test set of 1371 benign applications and 829 malicious applications. Compared to traditional method for detecting android malware, this system can obviously improve the detection accuracy of malicious app and reduce the analysis time, while avoiding the high complexity of dynamic analysis. Moreover, this system provides a friendly web interface, allowing users to upload the app file and receive the analysis report.

Keywords: Android, Malware, SVM, web interface

1 Introduction

As one of the most popular mobile platform for smartphone, Android system is facing the increasing threat from attackers and malware. According to statistics, in the first half of 2018, 360 Internet Security Center has intercepted a total of 2.831 million new malicious programs on the Android platform. Among them, 127,455 new mobile phone ransomware were intercepted, and 4,806 new mobile mining trojans were intercepted from January to July. The addition of mining trojans is nearly 20 times the total interception in 2017 [1].

Obviously, if malware can be detected and predicted in advance, the harm of malware to Android can be avoided, which will greatly improve the security situation of the current Android platform.

1.1 Detection Method for Android Malicious APP

Currently, detection techniques for malicious applications (malicious code) can be divided into two categories: static detection and dynamic detection.

Static detection. Static detection, such as *Kirin* [2], *Stowaway* [3], *RiskRanker* [4] and *Drebin* [5], refers to the use of decompilation techniques, control flow analysis, data flow analysis and semantic analysis to identify the behavior characteristics of the application without running the code. At present, the major security vendors widely use the feature code scanning method based on static analysis. Its main principle is to analyze the samples of malicious applications and store the extracted malicious sample signatures into the database. When scanning the application to be detected, the feature code of the application to be detected is extracted and compared with the database. If there is a feature code of the application to be detected in the database, it is determined to be malicious. This method has the advantages of being fast

* Corresponding Author

and efficient, but it is difficult to combat protection techniques such as code obfuscation.

Dynamic detection. Dynamic detection, such as *TaintDroid* by Enck et al. [6], *Droidbox* by Lantz [7], *pBMDS* by Xie [8] and Mobile-Sandbox by [9] Spreitzenbarth [9], is concerned with the actual behavior of the application. By running the application to be detected in an executable environment and monitoring its system calls, network access, file operations and memory modifications in real time, it is determined whether the application has malicious behavior. Compared to static detection techniques, dynamic detection techniques do not effectively audit all code because some code needs to be triggered under special conditions.

The static detection method of traditional Android malware is to obtain its static features by decompiling the unknown application software, and then use the acquired static features to calculate the similarity between them and the known malicious sample model, and finally According to the obtained similarity value and the previously set threshold value, when it is greater than the threshold value, it is considered as malware, and such a process is used to determine the category of an unknown application software. The key point in this detection process is the threshold setting. In the traditional static detection method, it is based on past experience and personal judgment. This setting has too many personal subjectivity. The uncertainty caused by such subjectivity largely affects the accuracy of the final prediction.

Although the dynamic detection method can obtain additional information that cannot be obtained by the static detection method, the dynamic monitoring method needs the sandbox environment support, result in the long-time detection and the implementation complexity.

1.2 Our Method

Considering the advantages and disadvantages of static detection and dynamic monitoring, we choose static analysis detection method to build this lightweight detection system. At the same time, the detection of maliciousness of unknown APP is essentially a classification problem. For this reason, the system uses SVM-based machine learning algorithm to implement the classification. A classifier model can be obtained by classifying the training samples. When the unknown app samples are detected, the classification model can be used to directly predict. The advantage of machine learning is that it does not need to rely on very large sample libraries, reduce the error of human intervention, and has good predictability for unknown app samples.

1.3 Roadmap

The remainder of this paper is organized as follows: Section 2 illustrates the classification model training process of the detection system. Section 3 evaluates the performance of this system. Section 4 introduces the web interface. A conclusion is drawn in Section 5.

2 Implement of Malicious APP Detection

2.1 APP Sample Library Introduction

The Android application sample dataset of this system is divided into benign application samples set and malicious application samples set. Among them, the sample dataset of the benign application has up to 5000 samples, which come from the xiaomi application market [10] and the 360 application market [11]. The malicious application samples come from the *Drebin Dataset* [5] and *Android Malware Dataset* (AMD) [12], which are two well-known malicious application sample libraries. Among them, Drebin's sample library collected 5,560 Android malicious application APK sample files from August 2010 to October 2012, a total of 179 malicious code families. AMD's sample library collected 24,553 malicious applications from 2010 to 2016, a total of 71 malicious code family categories, 135 variants.

These samples provide strong data support for subsequent machine learning model training.

2.2 Static Feature Extraction

The file format of the Android application is APK, which is essentially a compressed package. After decompression, multiple files containing feature information can be obtained. The decompressed internal file structure is shown in the following Figure.

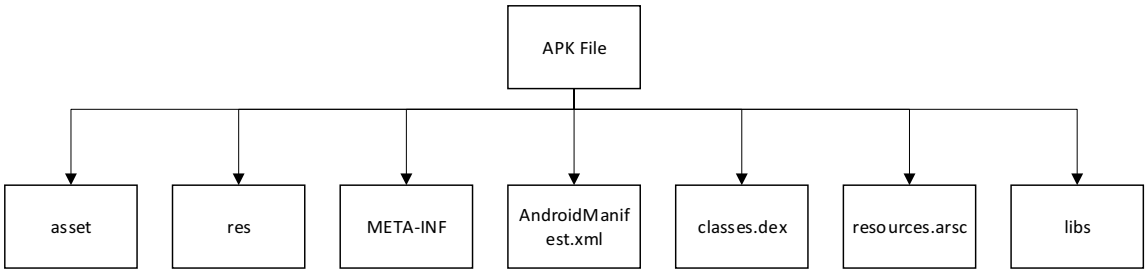


Fig. 1. File structure of an APK file

The *AndroidManifest.xml* file is an application configuration file used to configure the name, version, permissions, referenced library files, and all used Android component names and attributes. The classes.dex file is a Dalvik virtual machine executable file that is a bytecode file generated by source code compilation. It can be disassembled into a Smali file by decompiling tools.

This system can extract four aspects of features from the *AndroidManifest.xml* file:

- S_1 : *APP four major components*: Activity, Service, Content Provider and Broadcast Receiver
- S_2 : Application permissions
- S_3 : *External hardware components*: on which the App runtime depends
- S_4 : Filtered intents

By analyzing the Smali file obtained by disassembling the *classes.dex* file, the system can extract the following four features [5]:

- S_5 : *Restricted API function call*. The Android system provides a number of API function interfaces for the application to invoke, but some of the key functions are to restrict the application from making calls. Some malicious applications call these key functions in order to implement a root privilege attack. Therefore, in order to better understand the function of an Android application, the system will extract all the API functions called by the application.
- S_6 : *Suspicious function call*. After consulting the research results of other malicious application analysis organizations, it is found that some API functions are often called by malicious applications. For example, *system/bin/su*, *getExternalStorageDirectory*, *HttpPost*, *printStackTrace*, etc. Therefore, this system made a list of suspicious functions and checked whether they called the functions in this list when statically parsing the Smali file.
- S_7 : *The permissions actually used*. Most malicious applications will apply for a large number of permissions in the manifest file, and some permissions are not used when the application is running. Therefore, it is necessary to find out which system permissions are actually being used when the application is running.
- S_8 : *Network address*. Some malicious applications will establish a network connection with the attacker's server after stealing the user profile information, and then send the data to the attacker. Therefore, the disassembly file may contain an IP address, host name, and URL. Some addresses even contain botnet addresses. Therefore, the system will also extract the network address appearing in the disassembly file as one of the features.

In summary, we extracted eight feature sets from the *AndroidManifest.xml* file and the disassembly *classes.dex* file respectively, from which we can analyze the functional implementation and behavior of the application. The flow chart of the APK static feature extraction is as follows:

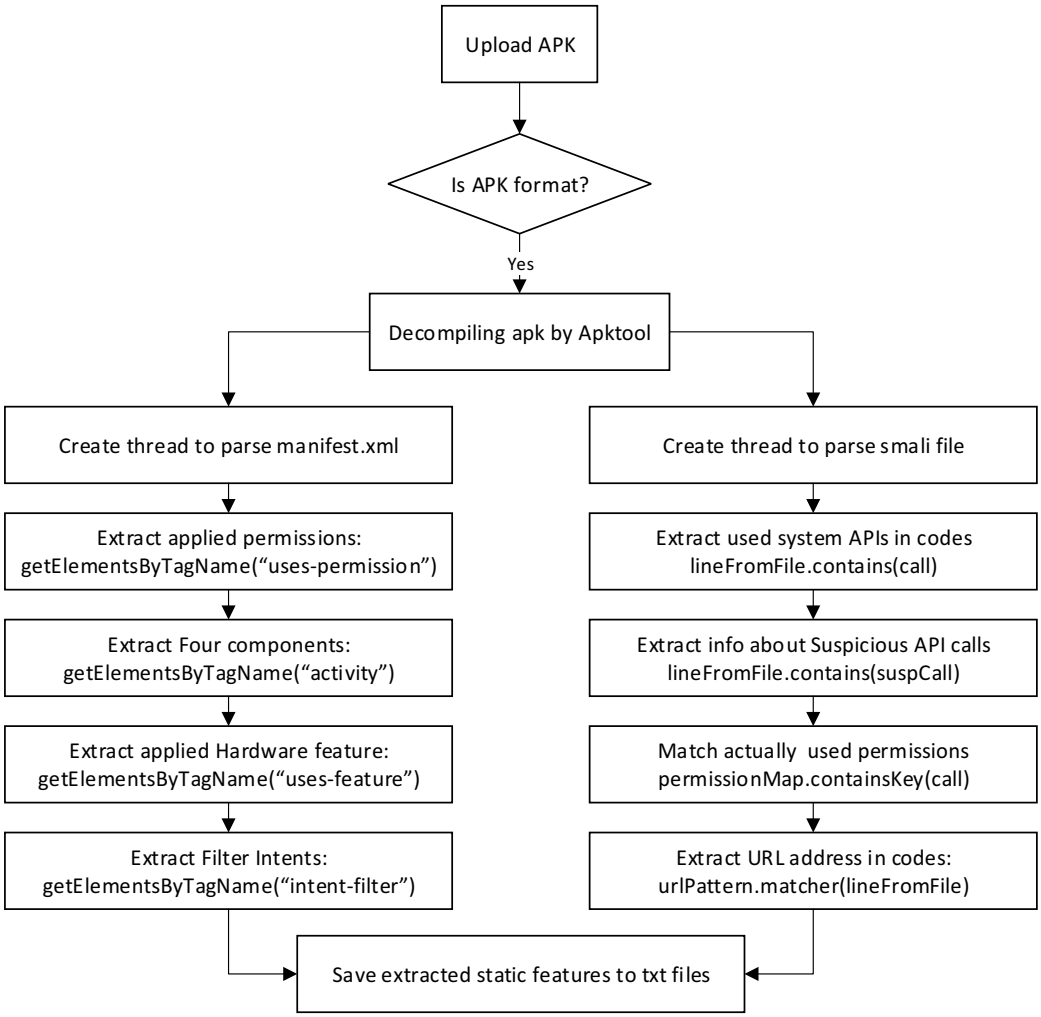


Fig. 2. APK static feature extraction flow chart

The following figure shows the feature information in a txt file obtained after static analysis of an application:

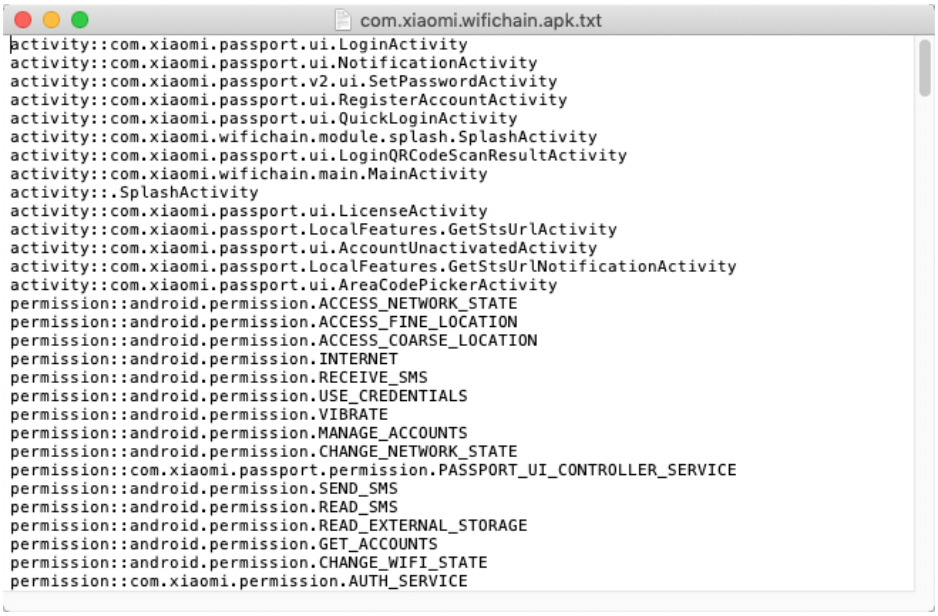


Fig. 3. Screenshot of extracted static feature info

2.3 Feature Vectorization

After the above steps, the static features of an APP are saved as strings in a txt file. In order for the machine learning algorithm to process, the string needs to be converted to a numeric type. Therefore, we need to map the features to the numerical vector space. Here we de-emphasize the eight feature sets extracted from all the applications to form a larger feature set:

$$S = S_1 \bigcup S_2 \bigcup S_3 \cdots \bigcup S_8 \# \tag{1}$$

For each application, we define a mapping function $I(x, s)$. If the application x contains the feature s on feature set S, then the value of corresponding position is set to 1, otherwise 0. The function definition is shown below:

$$I(x, s) = \begin{cases} 1, & application\ x\ contain\ features \\ 0, & otherwise \end{cases} \# \tag{2}$$

After mapping, each application x will have a feature vector $\varphi(x)$. Since we are using supervised learning, we use a set of samples of known categories to adjust the parameters of the classifier to achieve the required performance. So we also need to label each sample and tell the algorithm the category of the training sample during the training. We use y to indicate the category of each sample. If the sample x_i is a malicious sample, the value of y_i is 1, otherwise -1. Therefore, for each application x_i in the training sample, $(\varphi(x_i), y_i)$ gives out the information of features and category.

2.4 Model Training

The machine learning algorithm this system uses is the linear *Support Vector Machines* (SVM) [13]. Given two classes, i.e. malicious and benign application, a linear SVM model can determine a hyperplane which can separate the two classes with maximal margin (see Fig. 4).

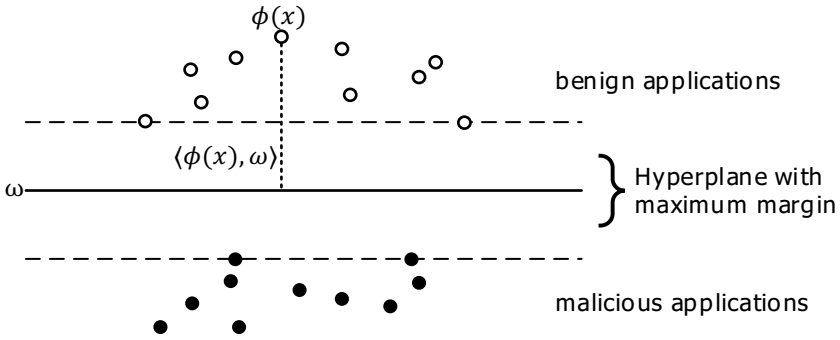


Fig. 4. Schematic diagram of SVM [5]

The hyperplane can be defined as:

$$\omega^T x + b = 0 \# \tag{3}$$

The optimal hyperplane of the required solution needs to meet the following constraints:

$$\min \frac{1}{2} \| \omega \|^2 \# \tag{4}$$

$$\text{s.t. } y_i(\omega^T \varphi(x_i) + b) \geq 1, i = 1, 2, ..., m \# \tag{5}$$

where m is the size of train set.

The goal of model training is to acquire the best parameter ω, b of the optimal hyperplane. To achieve the goal, a train set of 4500 normal applications and 6500 malicious applications is used to train model. The total vector space dimension, i.e. the length of feature vector $\varphi(x)$, adds up to 324465 in this train

set.

After finishing training, the best parameter ω, b of the optimal hyperplane have been determined. To detect a given unknown application z whether malicious or not, the first step is to decompile the application z for its features, and then the vector $\varphi(z)$ is acquired by mapping the application's features into feature vector space. Finally, a detection function is given:

$$f(z) = \omega^T \varphi(x) + b\# \quad \textbf{(6)}$$

Given a threshold t, $\varphi(z) > t$ declares application z as malicious application, otherwise as benign one. The threshold t is chosen as 0 most of the time.

3 Performance

To evaluate the detection performance of this system, a test set of 829 benign applications and 1371 malicious applications is applied. The detection result is shown in Table 1.

Table 1. Evaluation result of this system with a test set

Real category	Classification result	
	Benign	Malicious
Benign	1343	28
Malicious	7	822

According to Table 1, the *True Positive* (TP) is 1343, the *False Positive* (FP) is 7, the *False Negative* (FN) is 28 and the *True Negative* (FP) is 822. Therefore, this system can acquire good classification effect with *True Positive Rate* (TPR) of 97.96% and *False Positive Rate* (FPR) of 0.84%.

4 Web Interface

In order to facilitate the user to submit the APK file to be tested and view the detection report, the system provides a friendly web interaction interface based on the Python Flask framework [14]. The entire web implementation framework is shown below:

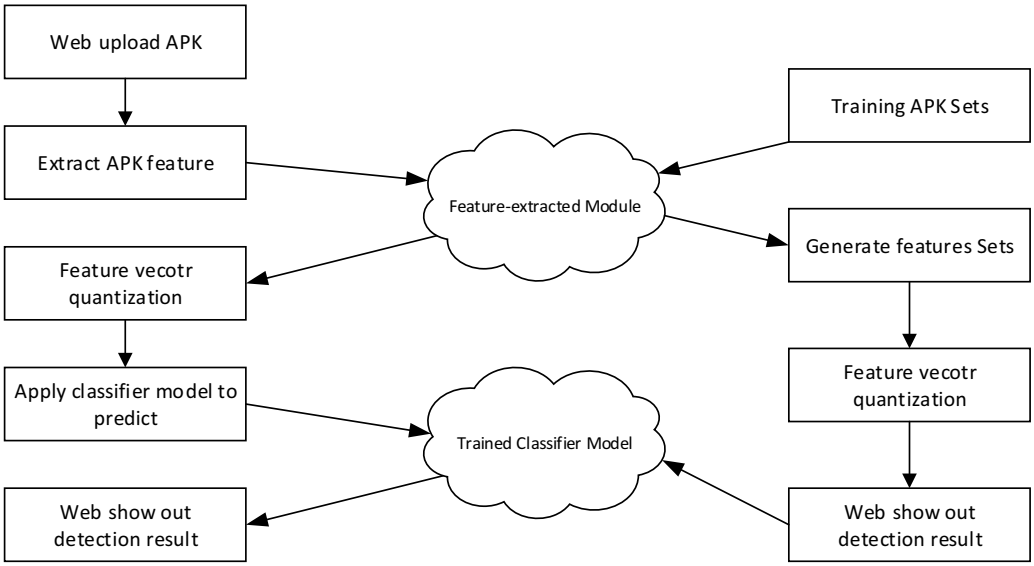
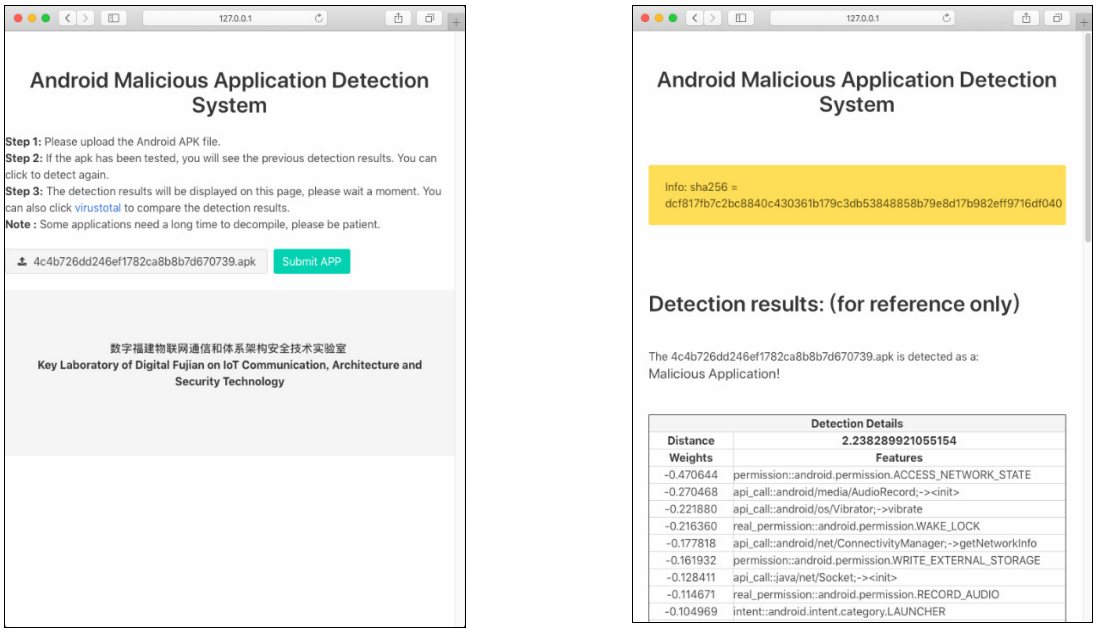


Fig. 5. The framework of web interaction interface

After submitting the APK file, the system will call the feature extraction module to perform static feature scanning on the APK file to extract static features. These static features are then mapped to a high-dimensional vector space to obtain a feature vector. By calculating the margin between the feature

vector and the system's pre-trained classification model (i.e. hyperplane), it is determined whether the uploaded APP is malicious or not. Finally, the results of the detection are returned to the user in the form of a report.

Fig. 6(a) illustrates the interface for users to submit APK files. Fig. 6(b) illustrates the interface of the returned detection report. The report gives the judgment of whether the uploaded APK file is a malicious APP, and shows the most relevant APP features and their corresponding contribution weights in the form of a table.



(a) The framework of web interaction interface (b) The framework of web interaction interface

Fig. 6. The web interaction interface

5 Conclusion

In summary, this system makes a deep research on Android malware detection technology, and extracts useful feature information from Android APK files through static analysis such as reverse engineering. Based on the theory of machine learning, a static detection system for Android malware was designed and implemented. In the actual test, the system has shown a good classification effect. However, the system still has certain limitations and room for improvement.

First of all, the system is based only on static analysis technology. It is mentioned that static technology is difficult to combat code obfuscation technology. Therefore, if the malicious application uses code obfuscation technology, the difficulty of decompilation of the application will increase, which will affect the extraction of static features and thus affect the impact the detection results. Moreover, at present, the system simply extracts the static features of the application, does not involve semantic analysis, and does not dig deeper into the association between these static features.

The system still needs to be optimized from three aspects:

- *Static analysis speed*. At present, the decompilation speed of the APK Dex file is slow. Therefore, when analyzing the APK file with confusion for the code, the application detection time is relatively long, for which we will try to use other better Android application static analysis tool for static analysis.
- *Explanation to malicious behavior*. Now the system can only diagnose whether the application is a malicious application and provide relevant feature weight information. However, if the application is malicious, there may be no malicious description of the malicious behavior. Therefore, we will also optimize in this respect.

- *Improve the classification accuracy*. We will also try to mine more feature information (such as combined with semantic analysis), improve existing algorithms, and further improve the classification effect of the classifier.

Acknowledgements

The work was supported in part by the Key Laboratory of Digital Fujian on IOT Communication, Architecture and Security Technology under Grant 2010499.

We would like to thank Drebin and AMD teams for their malicious Android applications dataset. Also, we would like to thank the anonymous reviewers for proofreading our paper and for giving us helpful comments to improve this work.

References

[1] 360 Internet Security Center, 2018 China Mobile Security Ecosystem Report. <http://zt.360.cn/1101061855.php?dtid=1101061451&did=491398428/>, 2018.

[2] W. Enck, M. Ongtang, P.D. McDaniel, On lightweight mobile phone application certification, in: Proc. ACM Conference on Computer and Communications Security, 2009.

[3] A.P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proc. ACM Conference on Computer and Communications Security, 2011.

[4] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang. Riskranker, scalable and accurate zero-day android malware detection, in: Proc. International Conference on Mobile Systems, Applications, and Services, 2012.

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon., Konrad Rieck, DREBIN, Effective and explainable detection of Android malware in your pocket, in: Ndss, 2014.

[6] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, A.N. Sheth. TaintDroid, An information-flow tracking system for realtime privacy monitoring on smartphones, in: Proc. the USENIX Symposium on Operating Systems Design and Implementation, 2010.

[7] P. Lantz, Droidbox - android application sandbox. <https://www.honeynet.org/gsoc2011/slot5>, 2011.

[8] L. Xie, X. Zhang, J.-P. Seifert, S. Zhu, A behavior-based malware detection system for cellphone devices, in: Proc. WiSec'10 - Proceedings of the 3rd ACM Conference on Wireless Network Security, 2010.

[9] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, J. Hoffmann, Mobile-sandbox: having a deeper look into android applications, in: Proc. the 28th Annual ACM Symposium on Applied Computing, 2013.

[10] Xiaomi Inc., Xiaomi App Store, 2018.

[11] 360 Inc., 360 Mobilephone Assistant, 2018.

[12] Argus Cyber Security Lab, Android Malware Dataset, 2018.

[13] M.A. Hearst, S.T. Dumais, E. Osuna, J. Platt, B. Scholkopf, Support vector machines, IEEE Intelligent Systems and Their Applications 13(4)(1998) 18-28.

[14] M. Grinberg, Flask Web Development: Developing Web Applications with Python, O'Reilly Media, 2018.