**Learning to Detect APK Malware**

Yubin Yang1, Guobin Huang2, Wei Jiang3

1School of Computer Science & Engineering, South China University of Technology, 510641 Guangzhou, China

2School of Computer Science & Engineering, South China University of Technology, 510641 Guangzhou, China

3Tandon School of Engineering, New York University, 11201, Brooklyn, NY, USA

[ronald\_yang@126.com](mailto:ronald_yang@126.com), [h.guobin@mail.scut.edu.cn](mailto:h.guobin@mail.scut.edu.cn), [wei.jiang@nyu.edu](mailto:wei.jiang@nyu.edu)

**Abstract:** Mobile malwares are spreading rapidly along with the fast development of mobile phone industry, especially the Android malwares. By the end of 2017 Q1, Android OS on smartphone worldwide has accounted for 85%, while the quantity of Android malware has reached 10 million by mid 2017. As a result, manual analysis on malware is not able to handle this kind of analysis. Therefore, an effective and efficient technique for APK detection is required. Many researchers have tried to detect and classify malware by virtue of machine learning of which the validity has been proven. This paper aims at balancing effective and efficiency so as to make the model available in the case with high demand of low latency and high throughput. Based on the research by Daniel Arp et. al. [31], we have conducted a lot of analysis on Android malware and have re-extracted feature sets by applying static analysis methods and classify the feature set by Random Forest. In the experiment of 21,203 benign samples and 20,976 malicious samples, the accuracy of our classifier is up to 97% while the detection time for each sample has decreased by 50% on average.

**Key Words:** Computer Security, Machine Learning, Android Malware, Static Analysis

1. **Introduction**

Android OS has the largest installed base on mobile phone industry. However, as it is open-source, Android malware consequently increases rapidly, which makes the problem of security of the Android system critical. A lot of malwares exploit the vulnerabilities of Android, for example, some malwares may visit some malicious websites or record users’ privacy secretly [3], while some may send message secretly when running in the background, some even attain privileged control (known as root access) [5] through the vulnerabilities.

Google launched Bouncer in 2012 in order to restrain the damage of malwares to the Android platform. Bouncer scans the apps uploaded to Google Play according to the rules/patterns/signatures of known malwares, rogue programs and Trojans. Further, it installs the apps in an online Android emulator for dynamic analysis. It collects behavior features dynamically and compares them with known behaviors from malwares and Trojans etc. Though Google has made good efforts to ensure the reliability of applications, researches [35] show that the Bouncer service can still be bypassed.

Android also provides various security methods to avoid installation of malware especially the Permission System. Each application requests user permission during installation or in the runtime for task execution on the smartphone (e.g. sending SMS). However, many users give permission to unknown applications without any attention. Malwares are consequently hard to be constraint by Android Permission System.

Therefore, a lot of researches have studied the analysis and detection methods before installation of Android app, which can be roughly divided into  two methods: static and dynamic analysis. For example, TaintDroid[25], DroidScope[22], FLASKDroid[30], Droidward [29] and Andromaly[11] are the methods that can monitor application behavior in the runtime. These methods perform well in identifying malicious dynamic behavior yet it cost too much resource. Compared to that, static analysis like DroidMiner[10], DroidSIFT[19], Kirin [18], RiskRanker [9] and Drebin[31] usually has small runtime overhead.

In order to deploy the model in scenario with high demand to be low latency and high throughput like firewall, we choose static analysis. Based on the research by Daniel Arp et. al. [31], we have decompiled Android application to extract features such as permission, code and strings, and then classified the experiment samples by means of random forest algorithm.

In short, the contribution of this paper includes:

1. The samples are more effective.

The dataset is balanced. The samples are distributed evenly from 2010-2017 while the amount of benign samples is almost equal to that of malicious samples. Benign samples are filtered by VirusTotal and by other factors such as downloads, scores, developers, etc. The amount of malicious samples used in training model is over 10K which is rare in academic research (most papers use hundreds of samples ONLY). This demonstrates the efficiency of our samples (Details are provided in section 7.1.1).

2. The features extracted are filtered.

According to methods used by researcher like Daniel Arp, we have extracted more than 1 million features at the beginning. However, a lot of useless information are included in those features such as the name including activity, service, content provider and broadcast receiver, which can be randomly changed. Felt et. al. [15] show that various malwares can escape from malware detection tool by changing the packet name, type name and method name. At last, the amount of features used in current experiment is 1664.

3. In this paper, we have conducted a lot of experiments against the model, evaluated the runtime and the detection performance, and performed the malware identification as well.

4. It’s well known that static analysis will fail because of malicious programs uses obfuscation and packing technique and our classifier is no exception. In the end of this paper, we have conducted significant amount of work on manual analysis in order to mitigate this effect.

1. **Our Technique**

In order to detect Android malware effectively, we have used a widely used static analysis technique to extract features from various data sources and model them using Random Forest. This section is divided into 3 parts for clearer discussion:

1. General Machine Learning Pipeline: Describe the general process of using machine learning algorithm to detect malware (Section 6.1);

2. Static Analysis: Describe the manifest and dexcode feature sets extracted from the Android APK file (Section 6.2);

3. Random Forest Algorithm: A brief introduction to random forest algorithm (Section 6.3).

**6.1 General Machine Learning Pipeline**

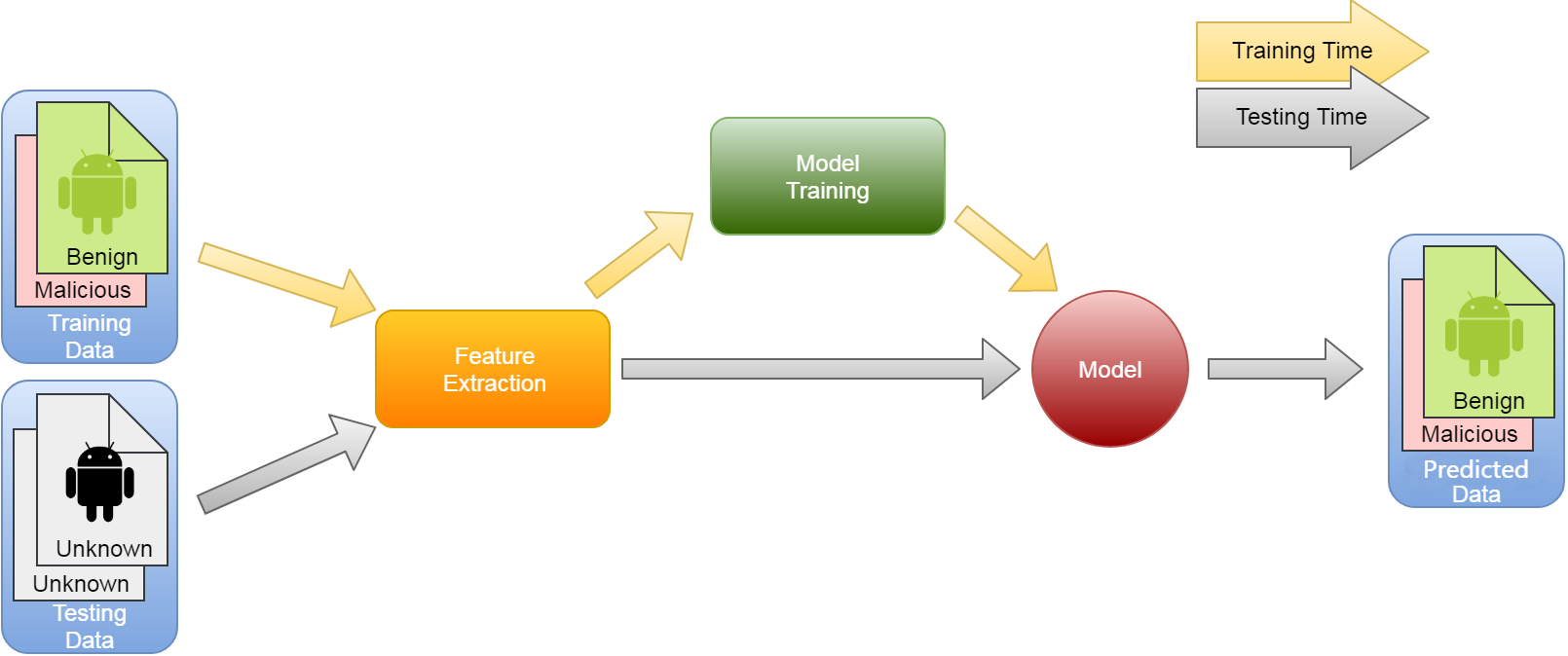
****

Figure 1. Training & Prediction Phase of the machine learning pipeline

Our objective is to train a model for Android malware detection, so we have collected a large number of malicious and benign apps (This is called the data collection phase). Secondly, we have designed and extracted various representative features for each app (We describe the related feature engineering in Section 6.2) with the hope that each feature vector will represent the APP nicely. Finally, we have trained the model so that the model can fit into the training data. At this point, our model is ready for prediction. When a new sample is presented to the model, it can return a confidence score (probability) to predict whether the sample is malicious. Figure 1 above provides a good description of model training and prediction.

**6.2 Static Analysis**

We have performed a lightweight static analysis for the Android apps. We mainly used the manifest file and the disassembled Dex code to extract the features by linear scanning. Taking into consideration generality and scalability of the analyzing process, we have represented all extracted features as strings such as permissions, intents, and API calls. We have extracted the following five groups of strings (From F1 to F5):

**6.2.1 Features from the manifest file**

Each Android app includes a manifest file named AndroidManifest.xml that developers can use for requesting resources including the hardware components, modules, and permissions for execution. Open source tool like Androguard [27] can be used to extract this information easily.

F1 - Hardware Components: If the app requests access to the camera, touch screen or the GPS module etc, then these hardware functionalities are needed to be declared in the AndroidManifest.xml file. Applications that require access to certain combinations of hardware components may potentially indicate harmful behavior. For example, an application that request access to GPS and network modules is able to collect user's location data and send it to an attacker over the network.

F2 - Requested Permissions: One of the most important security mechanisms introduced in Android is the Permission System. Prior to Android 6.0, permissions were voluntarily granted by user at installation time and then apps are allowed to access security-related resources. Previous work [7, 13] state that malware tend to request certain permissions more frequently than innocuous applications. For example, most malwares need to send SMS messages, requiring SEND\_SMS permission beforehand. Therefore, all the permissions listed in the manifest have used as features.

**6.2.2 The features from the Dex code**

Android apps are developed in Java and compiled into bytecode for Dalvik virtual machines. By using the Androguard [27] tool, we can efficiently disassemble the bytecode and get the API calls and other data info used in the app. We use this information to construct the following 3 feature sets:

F3 - Restricted API Calls: Android Permission System limits the application's access to a series of key API calls. We search the disassembled code for the occurrence of these calls, and record the function as a feature set and find out the app permissions corresponding to these APIs.

F4 - Used Permissions: Restricted API calls are extracted in F3, and F4 is a feature introduced by Felt et al. [15] - matching API calls and permissions. When a malware uses an API call that does NOT claim the requested permission, this may indicate that the malware is using a root vulnerability to bypass the Android platform restrictions.

F5 - Suspicious API Calls: When an application accesses sensitive data or resources on a smartphone, it may call specific APIs that can cause malicious behavior, so we extracted these APIs as one feature set. Wherein, the extracted feature set does not have any intersection with F3.

**6.3 The Random Forest Alg.**

We have applied bootstrap method to Random Forest. The decision trees inside the ‘Forest’ is independent between each other. The result of the classification is based on the majority of votes the classification trees decides.

We have tried different algs. but finally settle down to Random Forest. Not only because the Random Forest alg. enjoys good anti-noise ability, good stability but faster computational speed compared to SVM. It has been shown [36] that there is no significant difference between the Random Forest and SVM for generalization, the binary classification capability, but RF has the advantage being ‘preprocessing-free’.

1. **Evaluation**

After introducing our technique, we implement and evaluate its performance in this section. We conduct 4 steps as follows:

1. Preprocessing: This includes some description of our datasets, data cleaning, feature engineering and selection, feature sets visualization and analysis.

2. Detection Performance: We evaluated the detection accuracy, ROC curve, precision and recall etc of our model on a dataset comprised of 21,203 benign and 20,976 malicious samples. Further, we compared the results with Drebin and the commercial antivirus scanners as well.

3. Model Robustness: Model robustness refers to the ability to detect unknown malwares, survive from new evasion technique etc. We have classified the experimental datasets into malware families and evaluated the models by malware families.

4. Runtime Performance: Runtime related efficiency metrics are reported here.

**7.1 Data sets and feature sets**

First, we describe the datasets and feature sets used in our experiment. Then we compute the feature weights and give an explanation to the Top 10 features from our model. Finally, we visualize the classification ability of feature sets and make the results more intuitive for data analysis. At this point, a key question has been answered: Why these selected features have strong differentiating power for APK malware detection.

**7.1.1 Description**

**Data sets**: In this experiment, we have collected 154,113 apps, both benign and malicious, with 112,424 samples from different app markets and 41,689 malicious samples from the github project, security forums, blogs and the VirusTotal.

In order to identify the benign ones, we obtain the information such as downloads, rankings, ratings, and developer info on the application market. We have selected those 21,203 real benign apps based on these information. In order to identify the malwares, we have sent each sample to the VirusTotal service and marked them as malicious if it can not pass the scanning of the AV engines.

We believe that we have correctly labelled the datasets into benign and malicious ones. The final dataset is consisted of 21,203 benign samples and 20,976 malicious samples.

Meanwhile, we have also collected 5560 malicious samples used by Drebin. However, we just used the Drebin samples for model evaluation rather for model training. (See 7.2.3 for details)

**Feature sets**: According to Section 6, the total number of features extracted via static analysis from samples is shown in Table 1. F-ALL is the sum of all features in a total of 1,664. From the table, we find that there are 1,039 F2 features related to request permissions, while the Android system has only 150 system permissions [34] according to official document. We found that this is mainly due to some apps that (1) Use custom permissions and (2) Use irregular permission such as the letter case, spelling mistakes etc.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Table 1 The number of features | | | TYPE | NUMBER | | F1 | 147 | | F2 | 1039 | | F3 | 346 | | F4 | 59 | | F5 | 73 | | F-ALL | 1664 | | |  |  | | --- | --- | | Table 2 The top10 of features | | | 1 | F5\_Landroid/util/Base64;decode | | 2 | F2\_android.permission.READ\_EXTERNAL\_STORAGE | | 3 | F2\_android.permission.CAMERA | | 4 | F3\_Landroid/provider/Settings$System;putString | | 5 | F5\_Landroid/view/KeyEvent;getDeviceId | | 6 | F4\_android.permission.BLUETOOTH | | 7 | F5\_Ljavax/crypto/Cipher;getBlockSize | | 8 | F2\_android.permission.RECORD\_AUDIO | | 9 | F2\_android.permission.SEND\_SMS | | 10 | F2\_android.permission.RECEIVE\_USER\_PRESENT | |

**7.1.2 The weight of feature sets**

First, in order to better understand the weight of the extracted features in the whole feature set, we use the chi-square [32] test to characterize the F-ALL features, extract the weights of the features. From Table 3 we know that the F-ALL has a total of 1664 features within which the top 430 features occupy the feature weight of 99.00%. More, the largest number of features is selected from F3 while F5 enjoys the highest weight on average for each selected feature.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table 3 The weight of features | | | | | |
| Type | NUM-1 | WEIGHT | AVG(WEIGHT/ NUM1) | NUM-2 | NUM1/NUM2 |
| F1 | 33 | 3.81% | 0.12% | 147 | 22.45% |
| F2 | 141 | 32.99% | 0.23% | 1039 | 13.57% |
| F3 | 171 | 31.36% | 0.18% | 346 | 49.42% |
| F4 | 32 | 10.24% | 0.32% | 59 | 54.24% |
| F5 | 53 | 20.61% | 0.39% | 73 | 72.60% |
| F-ALL | 430 | 99.01% | - | 1664 | 25.84% |

Table 2 lists the top 10 features with highest weights, of which the most important feature is *F5\_Landroid/util/Base64*. One possible reason is that malwares seldom use system's own base64 algorithm class while the good ones do. Further, the malwares prefer to use some third party lib to escape the string decryption from antivirus scanners. Feature *F2\_READ\_EXTERNAL\_STORAGE* is the permission that can read external storage resources. This permission has been request by lots of malwares because users’ privacy info such as pics, chatting message from their phone can be stolen via this kind of permission. Feature *F2\_SEND\_SMS* is the permission used to apply for sending and receiving SMS. This permission is necessary for those malwares which need to send and receive SMS messages. Their typical malicious behavior includes sending SMS message for high fee reduction, performing C&C connection hidden from the backend via SMS message etc. Feature *F5\_Landroid/view/KeyEvent* is used to monitor key events. One possible explanation of why this feature gets into top 10 is that malwares seldom need interaction capabilities ( They usually perform malfunction and hide in background ) while one of the most important functionalities for good android apps are human-computer interaction.This is where this feature *F5\_Landroid/view/KeyEvent* come into play. Overall speaking, the above top10  are mostly sensitive operations but also featured in strong differentiating power.

**7.1.3 The visualization of the feature sets**

In order to better observe the classification ability of different types of features, we have used principal component analysis (PCA [33]) to map each feature into two-dimensional data, and plot it as shown in Figure 2. Here, we randomly have selected 200 benign and malicious samples respectively from the whole sample set. We can see from Figure 2 that the classification ability of F1 is not obvious (There is no clear classification curve for the plane),which is mainly because malwares seldom use hardware related info, while other types of features can find the classification curves clearly after using PCA.

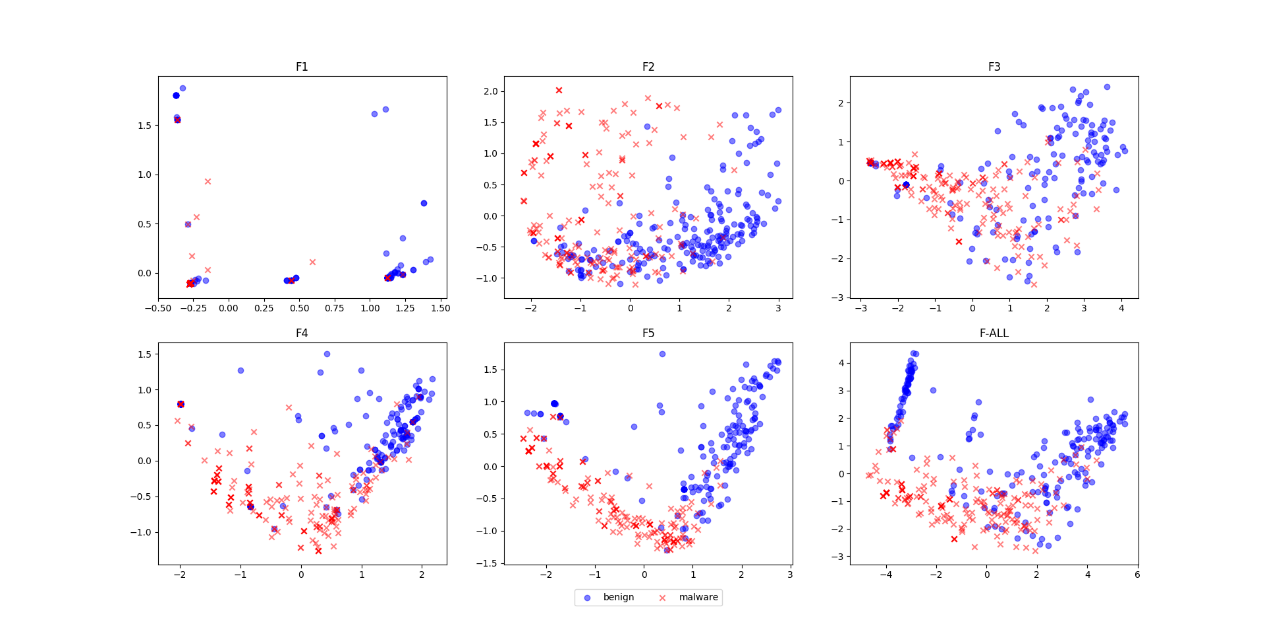


Figure 2 the 2D plots of the feature sets for both benign and malignant samples when using PCA dimensionality reduction

**7.2 Detection Performance**

In this section, we have evaluated the performance of our model. We randomly divided the dataset into training (70%) and testing (30%). The parameters of the trained model is fixed after model training and 10-fold cross validation has applied. Then we list the detection performance of 5 types of feature sets by 3 common machine learning algs. and do the comparison. Finally, we have compared the model with the common antivirus engines from 10 manufacturers.

**7.2.1 The accuracy**

This section uses the SVM, KNN and Random Forest algorithm for performance comparison, the results are shown in Table 4:

|  |  |  |  |
| --- | --- | --- | --- |
| Table 4 Detection rate of difference algorithm | | | |
|  | ACCURACY | | |
| TYPE | SVC | KNN | RF |
| F1 | 69.74% | 70.30% | 70.67% |
| F2 | 91.46% | 91.54% | 90.56% |
| F3 | 92.23% | 93.77% | 93.33% |
| F4 | 85.45% | 90.39% | 90.78% |
| F5 | 92.08% | 94.12% | 94.20% |
| F-ALL | 97.28% | 96.97% | 97.40% |

The classification effectiveness of F1 is the worst, which is in alignment with the conclusion we draw from Figure 3 in Section 7.2.2. Malicious apps rarely use hardware component, this also helps to explain why the effect of F1 is the worst. Dramatically compared, the other proposed feature sets in our work are all basically over 90% of detection accuracy. When using all the effective features, the detection rate is over 97% when the compound effect comes into play.

Last but not the least, Figure 3 shows the ROC curve of the learned model. The area under the ROC curve is near to 1, indicating a good ML model have been trained.

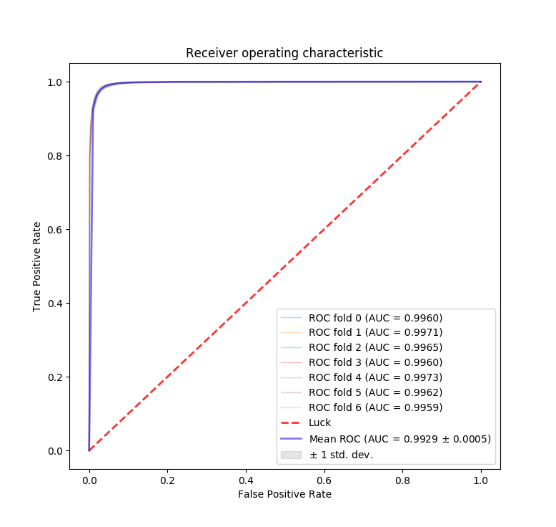
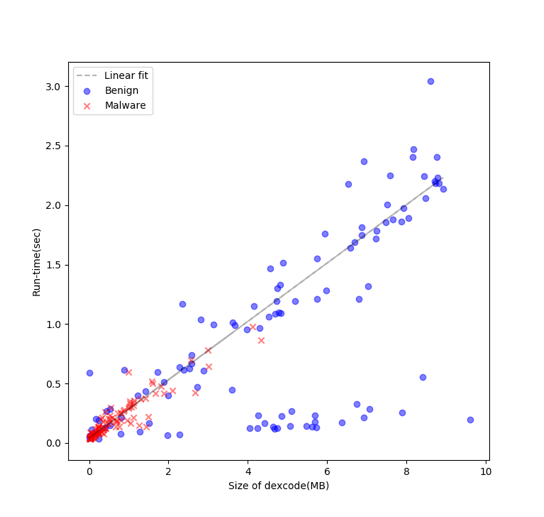
 ****

Figure 3. The ROC curve of the classifier Figure 4. Run-time performance

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table 5 Detection rates of our model, Drebin and anti-virus scanners. | | | | | | | | | | | | |
|  | Model | Drebin | AV1 | AV2 | AV3 | AV4 | AV5 | AV6 | AV7 | AV8 | AV9 | AV10 |
| Our sets | 97.32% | - | 94.57% | 93.82% | 93.16% | 88.68% | 88.01% | 83.97% | 79.68% | 62.07% | 54.92% | 24.75% |
| Drebin sets | 99.81% | 94.00% | 83.80% | 99.38% | 98.67% | 94.67% | 99.03% | 96.02% | 97.87% | 96.18% | 98.09% | 79.85% |

7.2.2 Comparison with Drebin

In this section, we compare the accuracy of different models under the same dataset. To do this, we obtained 5,560 malicious samples used by Drebin, leaving 4,766 samples after cleaning up 794 samples duplicate in our dataset. As shown in Table 5, the accuracy of our model is 99.81%, and only 9 malicious samples are falsely detected. Compared to 94% accuracy of Drebin, our model increased the overall accuracy by about 6%.

7.2.3 Comparison with AV scanners

Although we achieved good results compared to Drebin, more studies are needed in order to tackle the practical side of our model, like comparing the performance with the 10 best common antivirus engines in the market. The test results of each antivirus engine can be obtained through VirusTotal.

Table 5 shows our experimental results. As shown, the detection rate of AVs are significantly different. Among the ten antivirus engines, the highest accuracy was 94.6% while the lowest can only be 24.8%. We suspect that the lowest one is insensitive to Android malware. Compared to those, the accuracy of our model is 97.3%, better than all antivirus engines at a large margin.

For the Drebin dataset, all the antivirus engines perform well. The reasons behind is that the Drebin dataset is relatively old (from 2010 to 2012) and most of the malicious signatures have been found. According to the numbers we got, 8 out of 10 antivirus engines achieve more than 95% of accuracy while the lowest among the 10 can be 79.85%. In this setup, our model achieves 99.81% accuracy, outperforming all antivirus engines again.

**7.3 Detection of malware family**

In order to test the ability of model to detect unknown samples, we have firstly classified the sample into families. In this paper, we use AVClass [16] , an open source tool to classify the family for malware. By putting a VirusTotal report of sample into AVClass, it will output the most likely malware family label. The accuracy of the family classification for AVClass is around 88%.

Then, we use a standard Random Forest model for testing. Here, 4 kinds of situations are listed (Note: T1 is the reference group):

T1 -- Normal classification test

T2 -- Top10 families are excluded in training samples

T3 -- Top20 families are excluded in training samples

T4 -- Top20 families with random 10% in training samples

**7.3.1 Malware family classification**

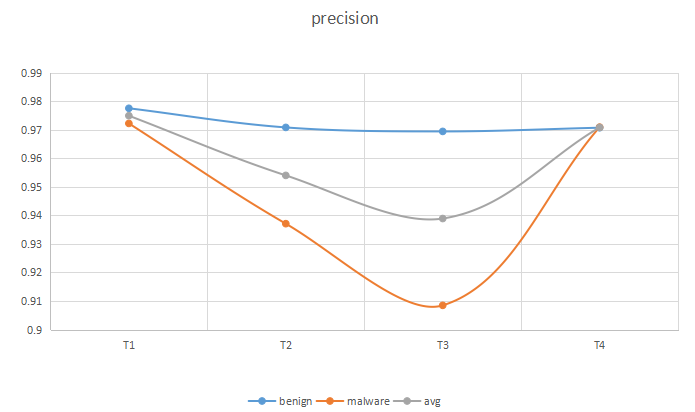
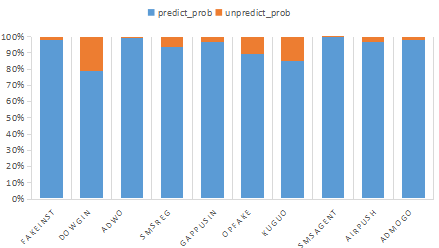
After using AVClass for classification, 20,976 malicious samples were divided into 454 families, among which 869 malicious samples don’t belong to any recognized families. Table 7 collects and sorts by decreasing number of malware family. The total number of samples here is 15,997, accounting for 76.26% of the total size.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table 7 ranks the top 20 malicious families | | | | | |
| index | family | count | index | family | count |
| 1 | fakeinst | 4444 | 11 | plankton | 369 |
| 2 | dowgin | 1878 | 12 | smspay | 355 |
| 3 | adwo | 1491 | 13 | youmi | 326 |
| 4 | smsreg | 1083 | 14 | mobwin | 306 |
| 5 | gappusin | 1044 | 15 | utchi | 305 |
| 6 | opfake | 991 | 16 | umpay | 266 |
| 7 | kuguo | 988 | 17 | lotoor | 250 |
| 8 | smsagent | 471 | 18 | wooboo | 222 |
| 9 | airpush | 441 | 19 | mseg | 193 |
| 10 | admogo | 384 | 20 | secapk | 190 |

In our dataset, the Fakeinst family contains the largest number of samples. Fakeinst is the largest family between 2012 and 2015, they (1) Pretend to be mainstream apps (2) Handshake from back-end (3) Send deduction message and (4) Hide its icon for evasion. According to Zhou et al. [17], 86% of their samples belong to Fakeinst. The airpush, sms agent and gappusin are famous families as well, of which the main behaviors are (1) Stealing personal or account information (2) Gaining access to device functions via backdoors (3) Downloading all kinds of malicious applications and (4) Sending text messages or calling premium numbers.

**7.3.2 Testing by types**

After classifying the samples, we then test each of the family using Random Forest. Figure 5 shows the accuracy of the output from the four tests. Blue indicates the accuracy of the benign sample, orange indicates the accuracy of the malicious sample, and gray indicates the average accuracy. In the process of using T1-T3, we gradually reduced the test set of malicious families, resulting in various degrees of decrease in the accuracy of each experimental result. At T4, we increased the testing set by adding 10% of the top 20 malware family, the accuracy of which is back to T1.

|  |  |
| --- | --- |
| Figure 5 the accuracy of T1-T4 | Figure 6 the classification of the TOP 10 families in T2 |

**T1——Common classification test**

The samples were divided into 70% training and 30% testing. The results of the test can be seen from T1 of Figure 5 that the classification of the samples are over 97%.

The Figure 7-A shows the top 20 malicious family classification results in T1. Most malware families achieve more than 95% accuracy when providing 70% of the samples as a training set. However, the detection rate of the secapk family was only 71.19%. The possible reason is: the number of training samples are NOT enough, resulting in low detection rate (We will return back to this family in Section 8).

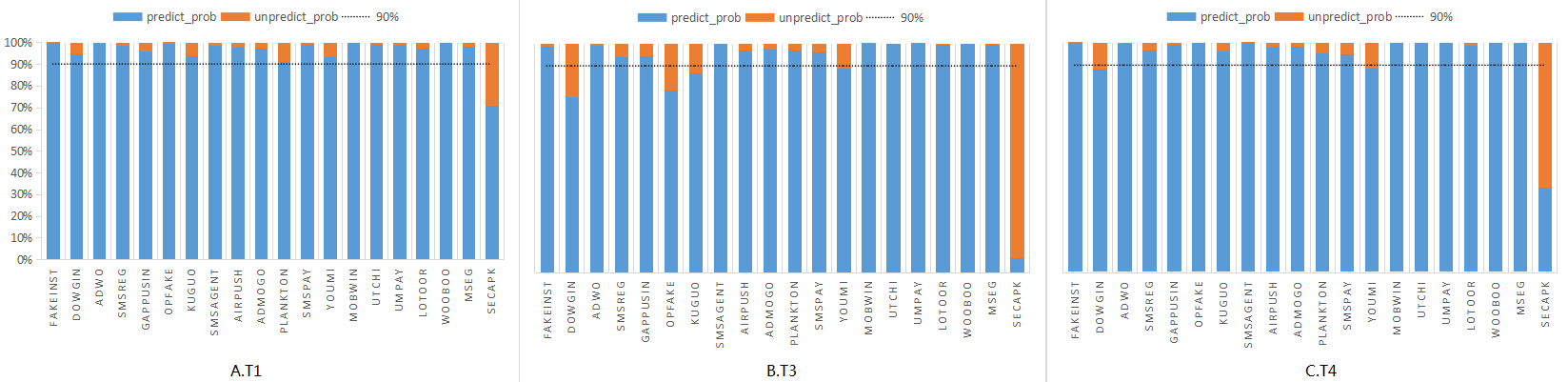


Figure 7 the top 20 families of classification results in T1, T3, T4

**T2——Top10 families are excluded in training samples**

The top 10 families do not provide training samples. We randomly select the same number of benign samples from other families as the training set, the remaining benign samples as the testing set. Figure 5 shows the accuracy of T2 which decreases to 95 %.

The results are shown in Figure 6. These samples are not for training, but for testing the ability of our model to detect unknown samples. Compared with T1, each family's recognition rate has decreased, among which that of dowgin family has decreased by 15.9%.

**T3——The top 20 families do not provide training samples**

The same as T2, this section increased the number of families from 10 to 20, which means that none of the top 20 families provide training samples., As shown, the accuracy was reduced by 1.5% compared to T2. With more and more new samples / families are added, if the model haven’t been updated periodically, we can expect significant decrease of accuracy.

The top 20 malicious family classification test results are shown in Figure 7-B. The detection rate of five malicious families are below 90%, and the secapk family recognition rate is less than 7%.

**T4——The top 20 families provided 10% of the samples**

Unlike T3, T4 provided 10% of the random samples from the top 20 families. The results showed that the results have been increased from 94% to 97% compared to T3 for the classification accuracy.

The top 20 malicious family classification results are shown in Figure 7-C. Compared with T3, the detection rate of most families are above 90%, almost the same as as T1, and the secapk family recognition rate has increased to 37%.

**7.3.3 Summary**

From above, the results show that (1) The detection rate of these families will be greatly reduced without the samples being in the corresponding families before the prediction time and (2) ONLY 10% of the samples of each family for training is able to ‘activate’ our classifier and obtain the high detection rate as T1.

Although there was no significant reduction in the detection rates of these families when exclude the top 20 families from training, about 90% of the overall detection rate is observed, indicating that the effectiveness of the extracted features and the robustness of the model.

In the future, if the model is not updated, the detection rate will inevitably decrease due to the continue appearance of new families / samples. Because of this, we need to update the model in time while only a handful of new samples are needed.

**7.4 Run-time Performance**

In order to show the efficiency of our model and collect info for better model optimization, we statistically analyze the prediction time. Here, we first randomly select 100 benign samples and 100 malicious samples, and then plot the running time and the size of dexcode for these 200 samples. Our hardware configuration is: i5 CPU with 8G memory.

Among them, Figure 4 is a scatter plot for dexcode size (X-Axis) and the running time (Y-Axis), the blue dots represent the benign samples while the red cross refers to the malicious ones.

Firstly, the dexcode size is roughly proportional to the model prediction time. That is, as the dexcode of a sample increases, it takes more time for prediction. Secondly, we can see that some of the benign samples have a large dexcode size, but their prediction time is small. Through manual analysis, we find that these samples are reinforced by some manufacturers' security services, making the pre-dexcode fail to be reversed by static analysis method (In this case, only a fraction of time is needed). Finally, we find that the majority of malwares are located in the lower left corner of the plot, while the benign samples are more evenly distributed. This shows that the size of the malwares are generally small. This is because the function of the malwares is usually less than that of the benign samples and also, designed for spreading to larger population.

Table 6 Detailed run-time analysis of model

|  |  |  |
| --- | --- | --- |
|  | AVG Run-time(sec) | The number of samples detected daily |
| Benign | 0.936 | 92307 |
| Malware | 0.194 | 445360 |
| AVG | 0.565 | 152920 |

In Table 6, we compute the latency and throughput of our system. From the table, we can see that the average prediction time of benign samples is much longer than that of malwares. It also shows that the dexcode size of benign samples is much larger than that of malwares. The Drebin model takes an average of 0.75 second per sample for prediction, with an estimated 100,000 samples a day. On average, our model takes 0.57 second for each sample and can detect about 150,000 samples a day. It’s clear that our model improves the system throughput by 50%.

1. **Discussion**

This paper, as a whole, validates the feasibility of using machine learning to statically detect Android malwares. In this section, we will focus on one difficulty: the detection capability of static analysis technique in the face of app obfuscation, hardening and packers.

**The secapk family** - Because the detection rate of the secapk family is low, we randomly sampled a few samples of this family and analyze the behavior and characteristics of these samples manually. Since the dex code has been encrypted, features extracted by static analysis for these samples are limited, the secapk family only has low detection rate in our experiment.

**New samples** - Apart from the dataset we used for model training, we collected 8,700 new malicious samples from other sources that caused our model accuracy drop by 3% from 97% to 94%. Preliminary analysis shows that the decrease of the accuracy is mainly due to a large number of packers. Although this is a well-known problem existed to all static analysis technique including ours, how to solve this problem remains for our future research directions.

Based on the above two aspects, further work to analyze the packers app is needed, as well as the extraction of effective static features. As an attempt, we perform the following steps:

First, we used a technique similar to Section 7.4 to take the effects of packers on our model into consideration. Secondly, we tried to manually analyze some of the packers' apps. Thirdly, we have managed to add N-gram algorithm to extract the character of bytecode of the Dalvik virtual machine for classification.

The above steps works and the accuracy of our model increases from 94% to 98% even we add a new significant amount of packed apps.

Due to the limited space, in this section, we only discuss the effectiveness of using N-gram for classification of packed malwares. Most security vendors scan the apps while providing packer services to the normal APKs but not malwares.Because of this, in our sample, only 5% of malware has been equipped with packers, compared to 35% for the case of benign samples. Packers typically use the Android dynamic loading method, specifically, using DexClassLoader class to load the original dex code. Of course, not only malware but also benign programs use dynamic loading.

However, it is clear that their purposes of using dynamic loading vary dramatically. Since benign software is developed for (1) Plug-in development or (2) Due to huge project size, multiple classes.dex files are required for development, while Malware uses Dynamic Loading mostly to evade detection by antivirus engines. Therefore, malware decrypts malicious dex files before dynamically loading them. The same family of malware, will use the alike encryption method. Therefore, we think that the use of N-gram algorithm to analyze bytecode can extract the behavior of malicious samples to decrypt malicious dex files. Combined with the static analysis we used at the beginning of the paper, we get better results and the accuracy of the model has improved to 98%.

1. **Related work**

Android Malware Detection has been a hot research topic in the past few years. Several different techniques have been proposed by the research community. We referred to Felt [15] and Zhou [17] for the overall android malware landscape and evolution.

**9.1 Static Analysis**

Static analysis of Android APK mainly refers to package de-compilation in order to extract features without running the program. The features which usually been extracted are permissions, API, strings, etc. Further, static analysis can also be used for abnormal control and data flow analysis.

Permission-based detection tool like Kirin [18] is representative in early days. Essentially, this is a static pattern matching based analysis technique that can identifies dangerous permission combinations. In their paper, nine security rules are carefully designed for detection. Although they enjoy high precision, their coverage rate is low due to the small number of rules.

RiskRanker [9] is designed to identify 0day Android malware by evaluating the potential security risks introduced by untrusted apps. It screens a large number of apps from the Android market and studies certain behaviors from them for detection, such as encryption and dynamic code loading etc, which create malicious patterns that can be used to detect invisible malware. Wu et. al. [14] proposed DroidMat, a tool based on static analysis that can get the permission info, component info and API info from the Android App as features. After that, a classification model is trained in order to detect unknown APK.

Static analysis is also used to analyze data leaks and malicious data flow. Hoffmann et al. [23] proposed SAAF, a static Android analysis framework that analyzes the smali code and creates program slices to perform dataflow analysis and to backtrack the parameters used by a given method. The framework enables (1) Automatically identify suspicious code snippets (2) Control flow diagram visualization and (3) Ad-related code identification.

Arzt et al. [24] proposed an analysis tool called FlowDroid for sensitive information leakage detection on Android by means of taint analysis. Unlike dynamic analysis, this static analysis approach does not introduce runtime overhead into the normal execution of the target application and can detect information leaks before the app is distributed to users. Once a malicious app has been found, it will be removed from the APP market.

DroidMiner [10], DroidSIFT [19] are FlowDroid-based tools that enable the detection of malicious programs by API dependency and similarity. Taint tracking technique which based on data analysis have been studied by many people in recent years. Although it can analyze the Android application in a much greater detail, its practical usage has been greatly thawted by its heavy overhead. For example, DroidMiner takes an average of 19.83s to detect one Android app and DroidSIFT takes 175.8s to detect one Android app. The analysis time of the above 2 is too long.

**9.2 Dynamic Analysis**

Dynamic analysis usually involves the execution of programs in a virtual machine or protected environment. Apparently, dynamic analysis introduces more runtime overhead than the static analysis counterpart.

Yang et al. [29] proposed DroidWard, a dynamic analysis system that can generalize the app behavior into six novel dynamic features. Decisions are made by combining the static and the new dynamic features after feeding the sample feature vector to the learned model. It takes around 120 seconds on average for examining an app.

In order to solve the problem that the Android itself can not provide visibility of private data, Enck et al. [25] proposed an system-level dynamic taint analysis system called TaintDroid. By using Android's virtual environment for real-time analysis, TaintDroid is able to track multiple sensitive data sources. The paper shows that (1) TaintDroid ONLY results in a 32% runtime overhead when using the CPU-binding micro benchmarks; (2) The overhead is negligible on the third party interaction app.

Dynamic analysis can conquer the problem of obfuscation and packer nicely. But it's overhead is not acceptable to mobile devices, it seems to be a feasible way to put the dynamic analysis into cloud. For example, ParanoidAndroid [26] uses virtual clones of smartphones to run and replay in parallel both in the cloud and on device. However, even the smallest execution trace sent to cloud will result in a significant decrease of battery life.

**9.3 Machine Learning**

In order to compensate the disadvantages of static and dynamic analysis, machine learning technique has also been applied to the Android malware detection problem.

The biggest advantage we prefer machine learning over static pattern matching analysis is that static machine learning can see the unseen malwares.

The biggest advantage we prefer machine learning over dynamic behavior analysis is (1) Shorter decision time (2) Pre-execution and (3) Low overhead.

Moghaddam et al. [20] propose a static analysis method, which improves the existing malware classification problem. This technique will first analyze the bytecode files, extract manifest and class features and then classify each class of features by group so that the best feature set can be found.

Droidmat [14] uses API calls to track and files to display for malware detection. R.Dhaya et al. [21] propose a malware detection system based on N-gram. This system obtains the program behavior through traditional static analysis, and uses N-gram to differentiate the two classes. Daniel Arp et al. [31] propose another static analysis technique, this technique extracts the app's permission information, component information and API information as features, and then applies SVM to model training. The resulting Drebin model can also run on mobile device.

Among all the related work, Drebin is the most similar to our work. We compare our work to theirs in Section 7. In summary, we (1) Collect a much larger dataset and (2) Has a more extensive coverage for various kinds of malwares and (3) Optimize the feature set significantly. Last but not the least, we prefer Random Forest to SVM due to better detection accuracy and smaller runtime overhead.

1. **Summary**

The android malwares are growing rapidly and an effective and efficient APK malware detection technique is needed. In this paper, we have proved the validity of applying machine learning to the problem of APK malware detection; We also have improved a well-known malware detector both in effectiveness and efficiency. Our random forest based model can achieve accuracy over 97% while the detection time being 50% faster.

1. **References**

[1] Smartphone OS Market Share, 2017 Q1

<https://www.idc.com/promo/smartphone-market-share/os>

[2] First Half of 2017 Mobile Security Report - Cheetah Mobile and ANTIY Mobile Security jointly released, <http://www.cmcm.com/blog/cn/security/2017-08-10/1092.html>

[3] IBanking Mobile Bot, <http://blogs.rsa.com/ibanking-mobile-bot-source-code-leaked/>

[4] MouaBad, <https://blog.lookout.com/blog/2013/12/09/mouabad-p-pocket-dialing-for-profit/>

[5]J.Forristal Android: One root to Own Them all. In: Black Hat USA 2013 <https://mediablackhat.com/us-13/us-13-forristal-android-one-root-to-own-them-allSlides.pdf>

[6] Liu X, Liu J. A two-layered permission-based Android malware detection scheme. In: Proc. of the 2014 2nd IEEE Int’l Conf. on Mobile Cloud Computing, Services, and Engineering (Mobile Cloud 2014). 2014. 142-148. [doi: 10.1109/MobileCloud.2014.22]

[7] Sarma BP, Li N, Gates C, Potharaju R, Nita-Rotaru C. Android permissions: A perspective combining risks and benefits. In: Proc. of the ACM Symp. on Access Control Models and Technologies (SACMAT 2012). 2012. 13-22. [doi: 10.1145/2295136.2295141]

[8] Wolfe B, Elish K, Yao DF. High precision screening for Android malware with dimensionality reduction. In: Proc. of the IEEE 2014 13th Int’l Conf. on Machine Learning and Applications (ICMLA 2014). 2014. 21-28. [doi:10.1109/ICMLA.2014.10]

[9] Grace M, Zhou Y, Zhang Q, et al. Riskranker: scalable and accurate zero-day android malware detection[C]//Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM, 2012: 281-294.

[10] Yang C, Xu Z, Gu G, et al. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications[C]//European symposium on research in computer security. Springer, Cham, 2014: 163-182.

[11] Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. Andromaly: A behavioral malware detection framework for Android devices.Journal of Intelligent Information Systems, 2012,38(1):161-190. [doi: 10.1007/s10844-010-0148-x]

[12] Amos B, Turner H, White J. Applying machine learning classifiers to dynamic Android malware detection at scale. In: Proc. of the 2013 9th Int’l Wireless Communications and Mobile Computing Conf. (IWCMC). 2013. 1666-1671.[doi: 10.1109/IWCMC.2013.6583806]

[13] Peng H, Gates C, Sarma B, Li N, Qi Y, Potharaju R, Nita-Rotaru C, Molloy I. Using probabilistic generative models for ranking risks of Android apps. In: Proc. of the 19th ACM Conf. on Computer and Communications Security (CCS 2012). 2012. 241-252.[doi: 10.1145/2382196.2382224]

[14] Wu DJ, Mao CH, Wei TE, Lee HM. Droidmat: Android malware detection through manifest and API calls tracing. In: Proc. of the 7th Asia Joint Conf. on Information Security (Asia JCIS). 2012. 62-69. [doi: 10.1109/AsiaJCIS.2012.18]

[15] Felt A P, Chin E, Hanna S, et al. Android permissions demystified[C]//Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011: 627-638.

[16] Sebastián M, Rivera R, Kotzias P, et al. Avclass: A tool for massive malware labeling[C]//International Symposium on Research in Attacks, Intrusions, and Defenses. Springer International Publishing, 2016: 230-253.

[17] Zhou Y, Jin X. Dissecting android malware: Characterization and evolution. In: Proc. of the 2012 IEEE Symp. on Security and Privacy (SP). 2012. 95-109. [doi: 10.1109/SP.2012.16]

[18] Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification[C]//Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009: 235-245.

[19] Zhang M, Duan Y, Yin H, et al. Semantics-aware android malware classification using weighted contextual api dependency graphs[C]//Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014: 1105-1116.

[20] Moghaddam S H, Abbaspour M. Sensitivity analysis of static features for Android malware detection[C]//Electrical Engineering (ICEE), 2014 22nd Iranian Conference on. IEEE, 2014: 920-924.

[21] Dhaya R, Poongodi M. Detecting software vulnerabilities in android using static analysis[C]//Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on. IEEE, 2014: 915-918.

[22] Yan L K, Yin H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis[C]//USENIX security symposium. 2012: 569-584.

[23] Hoffmann J, Ussath M, Holz T, et al. Slicing droids: program slicing for smali code[C]//Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, 2013: 1844-1851.

[24] Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[J]. Acm Sigplan Notices, 2014, 49(6): 259-269.

[25] Enck W, Gilbert P, Han S, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones[J]. ACM Transactions on Computer Systems (TOCS), 2014, 32(2): 5.

[26] Portokalidis G, Homburg P, Anagnostakis K, et al. Paranoid Android: versatile protection for smartphones[C]//Proceedings of the 26th Annual Computer Security Applications Conference. ACM, 2010: 347-356.

[27] Desnos A. Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!)[J]. 2013-03-26]. http://code, google, com/p/androguard, 2015.

[28] Breiman L. Random forests[J]. Machine learning, 2001, 45(1): 5-32.

[29] Yang Y, Wei Z, Xu Y, et al. Droidward: an effective dynamic analysis method for vetting android applications[J]. Cluster Computing, 2016: 1-11.

[30] Bugiel S, Heuser S, Sadeghi A R. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies[C]//USENIX Security Symposium. 2013: 131-146.

[31] Arp D, Spreitzenbarth M, Hubner M, et al. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket[C]//NDSS. 2014.

[32] Moore D S. Chi-Square Tests[R]. PURDUE UNIV LAFAYETTE IND DEPT OF STATISTICS, 1976.

[33] Jolliffe I T. Principal component analysis and factor analysis[M]//Principal component analysis. Springer, New York, NY, 1986: 115-128.

[34]Manifest.permission, <https://developer.android.com/reference/android/Manifest.permission.html#inhmethods>

[35] N. J. Percoco and S. Schulte. Adventures in Bouncerland: Failures of Automated Malware Detection within Mobile Application Markets. Proc. of Black Hat USA12, 2012

[36] HUANG Y, ZHA W. Comparison on classification performance between random forests and support vector machine[J]. Software, 2012, 33(6): 107-110.