Category Based Malware Detection for Android

Conference Paper *in* Communications in Computer and Information Science · September 2014

DOI: 10.1007/978-3-662-44966-0_23

CITATION
1

READS
244

4 authors, including:

Sanjay Rawat
University of Bristol
38 PUBLICATIONS **465** CITATIONS
SEE PROFILE

Shatrunjay Rawat
International Institute of Information Technology, Hyderabad
8 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Category Based Malware detection for Android

Vijayendra Grampurohit[1], Vijay Kumar[2], Sanjay Rawat[3], and Shatrunjay Rawat[4]

International Institute of Information Technology, Hyderabad

Abstract. Android, being the most popular operating system for the mobile devices, has attracted a plethora of malware that are being distributed through various applications (apps). The malware apps cause serious security and privacy concerns, such as accessing/leaking sensitive information, sending messages to the paid numbers, etc. Like traditional analysis and detection approaches for desktop malware applications, there have been many proposals to apply machine learning techniques to detect malicious apps. However unlike classical desktop applications, Android apps available on the "Google Play" [1] have a feature in "category" of app. In this initial work, we propose and investigate the possibility of improving the efficiency of machine learning approach for android apps by exploiting the category information. Experiment results performed over a large dataset, are encouraging which shows the effectiveness of our simple yet productive approach.

Keywords - mobile security; Android; malware detection; machine learning; data mining

1 Introduction

Android is an open source Linux-based mobile operating system distributed by Google. According to the latest statistics, android powers hundreds of thousands mobile devices over 190 countries [2]. Google Play [1] is the official android centralized market place maintained by Google, where any independent application developer can submit his/her android app and make it available to the users. The growing popularity of this android ecosystem also is becoming a worthy target for security and privacy violations. Highly sensitive and confidential information such as text messages, private and business contacts, calendar data, etc may be leaked through an application. Sensors such as GPS present in the phones allow applications to provide context-sensitive user experience, they also create additional privacy concerns it can exploit the data for tracking or monitoring. Apart from these issues, smart phones are also susceptible to various malware threats such as viruses, Trojan horses, worms, etc. [3].

Android security model relies highly on permission-based mechanism. There are about 130 permissions that govern access to different resources. Whenever an user tries to install a new application, he/she is prompted to approve or reject all the permissions requested by the application. The application will be installed only after the user accepts all the necessary permissions requested by it.

In this paper, we use the permissions and api level information from the apps as the features to detect malicious applications. Further we observe that, android store [1] defines a category for every published application. We have done extensive studies and discovered that, certain categories are highly prone to malicious acts compared to other categories. We explicitly incorporate this information in our model and learn a naive bayes classifier for each category using the features that encode information about permissions and api calls. Given a new test application with a known

category, we apply an appropriate classifier to detect if the application is malicious. We created a large data set of android applications and achieve an improvement of $3-4\%$ by incorporating category level information.

The rest of the paper is organized as follows: We discuss the Preliminary background in section 2. section 3, describes our method. section 4, contains our experiment results and discussions. In section 5 we describe related works and finally conclude in section 6.

2 Preliminary Background

In this section we discuss Android architecture and basics of android application in brief.

2.1 Android Architecture

Android platform is an open source software stack for building and running applications with various layers running on top of each other with lower-level layers providing services to upper level layers. It consists of an operating system, native libraries, application framework, and core applications.

The kernel derived from linux kernel is the first layer of software that interacts with the device hardware. Android kernel handles power and memory management, device drivers, process management, networking and security. It also ensures certain separation between applications by creating different processes for different applications.

On top of the kernel are the native libraries. The libraries component acts as a translation layer between kernel and application framework. The library component shares its space with the runtime. Some of the core libraries include the Surface Manager (responsible for graphics on the devices screen), 2D and 3D graphics libraries, WebKit (the web rendering engine that powers the default browser) etc.

The run-time component consists of the Dalvik virtual machine that interacts and runs an application. The virtual machine is an important part of the Android operating system and executes system and third-party applications.

The next layer in the stack is the application framework. The framework provides a suite of services to the developers for writing applications. The layer supports entities such as the Package Manager for managing applications on the phone, and the Activity Manager for loading activities and managing the activity stack.

Finally, at the top of the stack resides the user applications. These include applications that is written by developers. Applications such as Contacts, Phone, Messaging, and Angry Birds apps are executed in this space by using the Api libraries and the Dalvik virtual machine.

2.2 Android Application Basics

Android applications or simply apps are written in Java programming language. The Android SDK tools compile the source code, any data and resource files into an Android package - an archive file with an *.apk* suffix. A single *.apk* file which contains all the necessary code of an application is used to install the application in the Android powered devices.

An app must declare its components in a *manifest* file which must be at the root of the application project directory. The manifest file also states the user permissions that the application requires, such as internet access, read-access to the user's contacts, access to user's location, etc. We show an example of manifest file in listing 1.1. The example manifest file provides information

about required permissions to read phone state, SMS, Internet access, etc. We extract and encode the information from the manifest file as features to detect the malicious applications.

Listing 1.1: Overview of Manifest file

```
<manifest android:versionCode="15" android:versionName="1.3.1"
    android:installLocation="auto" package="com.oe.crazycorns"
 xmlns:android="http://schemas.android.com/apk/res/android">
 <uses-sdk android:minSdkVersion="8" />
 <uses-permission android:name="android.permission.INTERNET" />
 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
 <uses-permission android:name="android.permission.READ_PHONE_STATE" />
  .....
  .....
</manifest>
```

3 Proposed Method

In this section, we explain the feature extraction and our proposed Naive Bayes classifier which exploits the category information of an application.

3.1 Reverse engineering Android App

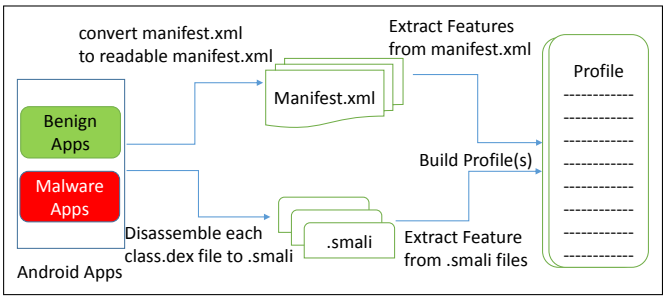

Fig. 1: Different Stages in Feature Extraction

Reverse Engineering is a process by which one can discover and understand the complete working of an application by learning its operation, structure and functions. In this work, we use tools like ApkTool [4], Smali/Baksmali [5], Dex2Jar [6] and Android SDK for reverse engineering a Android Application.

The ApkTool [4] is a 3rd party tool that is used to analyze closed Android application binaries. We show the steps involved in Figure 1. To parse the .dex file, we use a tool called Baksmali [5] which is a disassembler for the dex format used by Dalvik. Baksmali disassembles .dex files into multiple files with .smali extensions. Each .smali file contains only one class information which is equivalent to a Java .class file.

3.2 Static Feature Extraction and Refinement

In order to build features for an application, we extracted the (binary) APK file using ApkTool [4] and Smali disassembler [5]. We extracted all the relevant api invocations in the Android application along with the permissions in the manifest file. Instead of using all the api calls, we use only a subset of them namely sensitive api calls, which are governed by an Android permission settings. To obtain the set of sensitive api's, we relied on the work of Felt *et al.* [7], who identified and used the mapping between permissions and Android methods. Further, a sensitive api is considered only if it is declared in the binary and if its corresponding permission is requested in the manifest file. This resulted in the elimination of large number of api calls. We used the Android Asset Packaging Tool (aapt) to extract and decrypt the data from the AndroidManifest.xml file, provided by the Android SDK.

3.3 Application categories

When a developer publishes an application in Google Play, one needs to picks the category under which the application will be published. Currently, Google play has around 30 categories which are shown in Table 1. For each category, applications are ranked based on a combination of ratings, reviews, downloads, country, and other factors. We have done extensive study and found out that number of malwares is not uniform across all the categories. Certain categories such as Entertainment, Games, Tools, etc. are highly prone to malwares while categories such as medical, social have few malwares. In our work, we explicitly learn a model that exploits this information.

3.4 Bayesian Classification Model

One of the simplest and powerful machine learning techniques is Bayes classifier. This is probably due to their simplicity, linear computational complexity and accuracy. It is also referred a *Naive Bayesian Classifier* as it makes the assumption that all the features representing the data are independent for a particular choice of the behavior one is trying to learn.

The naive bayesian classifier consists of learning and detection stages. The learning stage learns a model from sufficient number of training data containing both benign and malicious android applications. During detection or testing, one can infer whether the given test application is malign or not using the model learnt during the training.

We extract the desired features from each application in the corpus. The feature set is further reduced by a feature reduction function. Each application X is represented as a vector $X = [x_1, \ldots, x_m]$, where $x_i \in \{0, 1\}$, $\forall i = 1, \ldots, m$ are the random variables indicating a particular characteristic feature of the android application. We consider the api calls and permission as the characteristic features. If a particular api call/permission is present in the application then the corresponding feature x_i is defined as 1 otherwise as 0.

Let Y denote the label of each application suspicious, $Y \in \{malign, benign\}$. We define the application category as $C \in \{1, 2, \ldots, K\}$ where K denote the number of categories available in the android store. We exploit the information from both api calls and categories and thus we define the posterior density of Y using bayes rule as,

$$P(Y = y_i | X = \mathrm{x}_j, C = c_k) = \frac{P(X{=}\mathrm{x}_j | C{=}c_k, Y{=}y_i) P(C{=}c_k | Y{=}y_i) P(Y{=}y_i)}{P(X{=}\mathrm{x}_j | C{=}c_k) P(C{=}c_k)} \tag{1}$$

where the probabilities, $P(X|C,Y)$, $P(C|Y)$, $P(C)$, $P(X|C)$ and $P(Y)$ are estimated from the training data.

During inference, any test app is classified as malign if P(Y=malign|X,C) >P(Y=benign|X,C). Since the category of a test application will be known apriori, we classify the app using the classifier trained on the corresponding category.

4 Experiment Results and Discussions

In this section, we describe the dataset and report our experimental results.

4.1 Data set

Our data set consists of 25865 apps collected from Google Play [1] and Android Malware Genome Project [8]. We collected 24335 apps from Google Play and 1530 applications from Genome Project as shown in Table 1. We collected only the top free apps in each category for creating a benign set. For the benign applications, we used VirusTotal [9] to make sure that they are genuinely benign. Each of these benign and malware applications belong to 30 categories as defined in android market (Table 1).

Table 1: Benign & Malware App Categories

Category	Geniune	Malware	Category	Geniune	Malware
Arcade	1409	123	Medical	499	5
Books & References	884	10	Music & Audio	1287	30
Brain	1342	117	News & Magazine	545	20
Business	574	13	Personalization	2131	16
Cards	545	21	Photography	324	37
Casual	1658	140	Productivity	728	85
Comics	517	19	Racing	615	90
Communication	280	83	Shopping	169	10
Education	959	30	Social	683	5
Entertainment	1546	173	Sport	800	4
Finance	403	20	Sport Games	633	30
Health & Fitness	703	27	Tools	1227	275
Libraries & Demo	564	32	Transportation	397	17
Lifestyle	1112	26	Travel & Local	602	23
Media & Video	827	37	Whether	404	12

4.2 Evaluation measures

To evaluate the effectiveness of proposed approach, we calculate true positive, true negative, false positive and false negative rates, precision and recall rates and F-measure in our experiments. These measures are defined as follows. Let TP (true positive) be the number of Android malware apps that are correctly detected, FN (false negative) be the number of malware apps that are detected

as benign, TN (true negative) be the number of benign apps that are correctly classified; and let FP (false positive) be the number of benign app that are incorrectly detected as Android malware. In terms of classification error, two cases can occur: (a) A benign app may be misclassified as suspicious and (b) a suspicious app may be misclassified as benign. For our problem, the latter case is more crucial as it is more important to prevent a malicious app in reaching the end device than excluding a benign app from the distribution chain. We use the following measures to check the performance of our proposed approach.

True Positive Rate (TPR) = $\frac{TP}{TP+FN}$ False Positive Rate (FPR) = $\frac{FP}{TN+FP}$

Recall (Rec) = $\frac{TP}{TP+FN}$ Precision (Prec)= $\frac{TP}{TP+FP}$

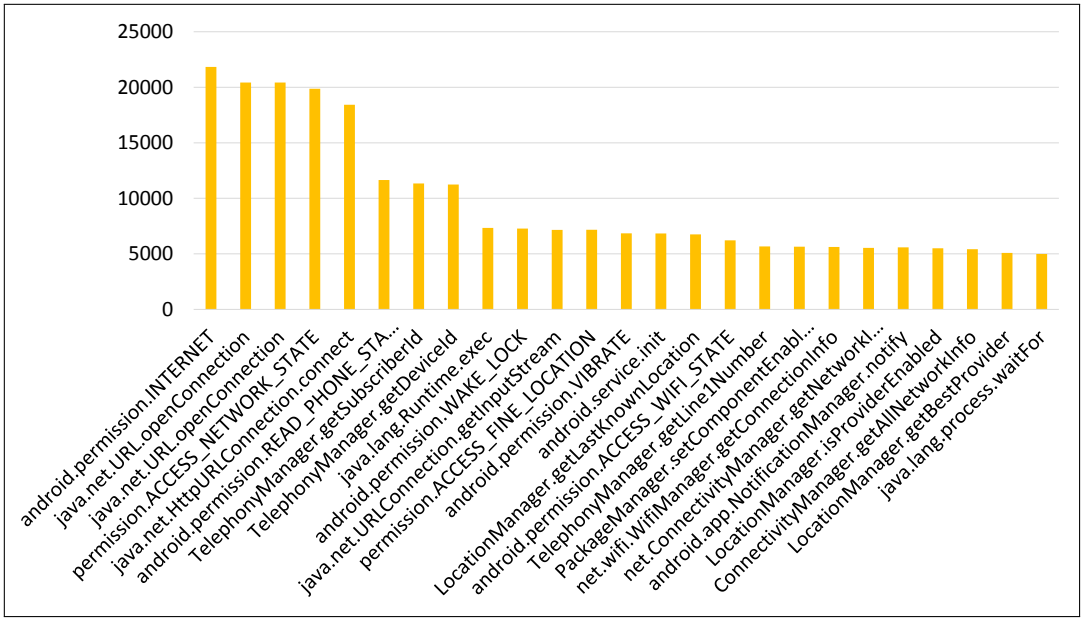

Fig. 2: Frequently occurring permissions & Api calls

Figure 3 shows the frequently occurring api calls and permissions in the Android application. We consider only the api calls and permission that are highly frequent as features. We adopted a ten-fold cross validation strategy for our expeirments. We trained our model using 9 folds and tested on remaining fold. We repeated the experimented 10 times and report the average accuracy. This ensures a wider range of samples for the testing of the classifier.

We also conducted our experiments as Aafer *et al.* [10] but with four different sets of top features namely, 5, 10, 15 and 25 respectively. These top features are selected based on frequently occurring features in our samples as shown in Fig 4. We refer top 10, 15 and 25 ranked features as $10Tf$, $15Tf$ and $25Tf$ respectively and five lowest ranked features as $5Lf$.

Figure **??** shows the average error rates and accuracy for different feature sets with and without category information. We observe an increasing accuracy and decreasing error rates when larger

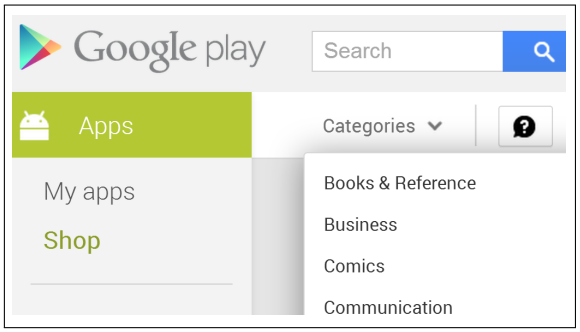

Fig. 3: Category

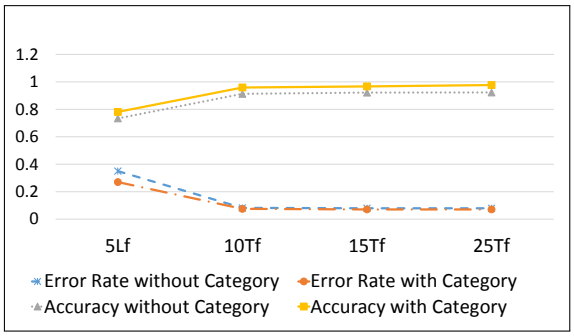

Fig. 4: Error Rate & Accuracy

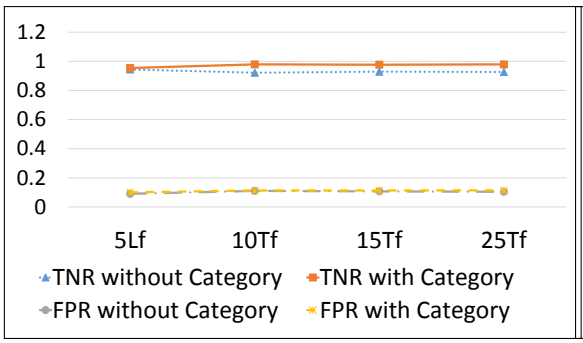

Fig. 5: True Negative & False Positive Rate

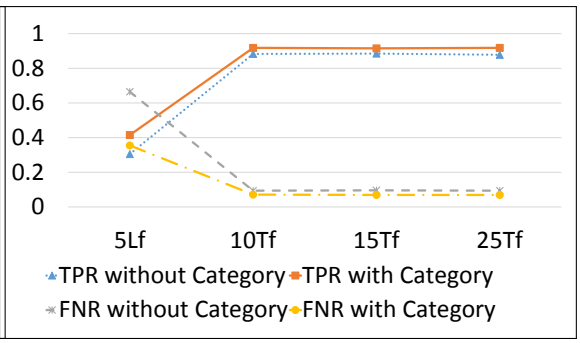

Fig. 6: True positive & False Negative Rate

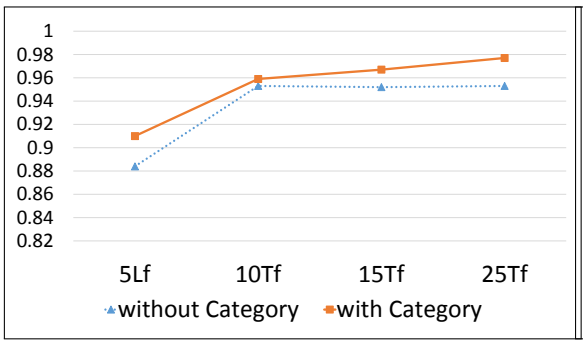

Fig. 7: Precision

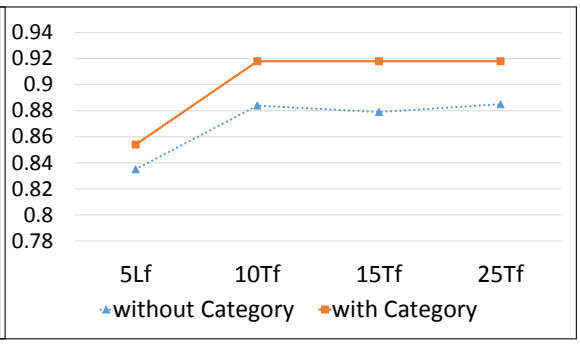

Fig. 8: Recall

number of features are used to train the classifier. It is also evident, by exploiting the category information, there is a clear improvement in the accuracy and error rates. Also, note that there is almost a difference of 20% in the performance using $5Lf$ and $10Tf$ feature sets indicating the importance of feature ranking based on the frequency of api calls.

Figure **??** shows the true negative and false positive rates and Figure **??** shows the true positive and false negative rates using different sets of features. We can observe in both the cases, that there is a improvement in the performance when the category information is included in the model. Finally, we show the precision and recall in Figure **??** and **??** with varying number of features. As the features are increased, both precision and recall improved and when the category information is included in the model, the performance is even better.

We summarize the results of various measures without category information in Table 3 and with category information in Table 4. It can be observed from the Table 4 that an average improvement of $3-4\%$ across all the categories is achieved.

We also report the measure Area under Curve (AUC) which defines the total area under the Receiver Operation Characteristic (ROC) curve, for different number of features. We can see that AOC for $10f$, $15f$, and $25f$ is very close to 1 implying a very good performance.

Table 2: With Category

	5fL	10f	15f	25f
ERR	0.35	0.082	0.079	0.079
ACC	0.734	0.913	0.922	0.923
TNR	0.944	0.922	0.929	0.927
FPR	0.09	0.11	0.107	0.104
TPR	0.305	0.883	0.885	0.878
FNR	0.665	0.094	0.096	0.094
Prec	0.884	0.953	0.952	0.953
Rec	0.835	0.884	0.879	0.885
AUC	0.58	0.907	0.9	0.914
F-Mes	0.858	0.917	0.917	0.918

Table 3: Without Category

	5Lf	10Tf	15Tf	25Tf
ERR	0.27	0.075	0.071	0.071
ACC	0.781	0.959	0.967	0.977
TNR	0.954	0.979	0.977	0.979
FPR	0.1	0.113	0.113	0.114
TPR	0.415	0.918	0.915	0.918
FNR	0.355	0.071	0.069	0.069
Prec	0.91	0.959	0.967	0.977
Rec	0.854	0.918	0.928	0.928
AUC	0.635	0.956	0.956	0.959
F-Mes	0.881	0.938	0.941	0.958

5 Related Work

In this section, we describe some of the previous approaches employed by researchers for detecting the malacious applications. There are various methods in the literature adapting different strategies to detect the malware applications. These can be roughly grouped into static and dynamic analysis. Below, we give a brief review of various approaches belonging to these categories.

One of the important methods for analysing the malware is through static analysis which performs detection of malware applications before installation or run on the device. There are various approaches proposed for malware detection based on static analysis. Comdroid was proposed by Chin *et al.* [11] for detecting application communication based vulnerabilities in Android. ProfileDroid [12] and Risk Ranker [13] leverages static analysis for profiling and analyzing Android applications. ScanDroid [14] proposed by Fuchs *et al.* analyses the data policies in application manifest and data flows across content providers. Barrera *et al.* [15] propose a methodology for

identifying application clusters based on requested permissions. Dicerbo *et al.* [16] uses Android permissions in the manifest file to identify malicious Android applications. Zhou *et al.* [17] proposed a permission-based behavioral footprinting scheme and heuristics based filtering scheme to detect the malware. Other static analysis approaches exploit the information present in bytecode of the android application to predict its behavior [18]. Using the bytecode, they retrieve information ranging from coarse-grained levels as packages to fine-grained levels as instructions. However, this approach is computationally expensive and we thus focus on extracting package and api level information in our work, as they clearly capture the applications behavior.

A different direction for detecting Android malware relies on dynamic analysis where the malwares could be detected in run time. Zhao *et al.* [19] propose AntiMalDroid to detect Android malware that use logged behavior sequence as the feature and construct the models for further detecting malware and its variants effectively in runtime. Enck *et al.* [20] perform dynamic taint analysis to track the flow of private and sensitive data through third party applications and detect any leakage to remote servers.

Signature-based approaches detect the malwares by the sets of rules or policies. The advantage of such methods is that they can precisely detect the Android malware if matching any of signatures. Kim *et al.* [21] build a power consumption history from the collected samples, and generate a power signature from the constructed history for power aware malware detection. Enck *et al.* [22] proposed Kirin, security service that perform the certification of applications. They define a variety of potential dangerous permission combinations as rules to block the installation of potential unsafe applications. Our approach is different in a way that, these techniques are not adaptive to a new Android malware and they require continuous update of the signatures.

In [23], authors extract the function calls from binaries of applications and apply their clustering mechanism, called Centroid, for detecting unknown malware. In contrast, our approach is based on automated analyses of Android packages. A recent paper by Sahs and Khan [24] proposes a machine learning approach to Android Malware detection based on (SVM). They use the Android permissions in the Manifest files as the features and learn a single-class (SVM) model using benign samples alone. This is contrast to our approach which uses Api calls, permissions and categories as features for training the naive-Bayes model.

6 Conclusion

In this paper, we proposed a naive-Bayes approach for detecting Android malware application. Unlike the previous approach which uses only api calls for prediction, we combine various information from api calls, permissions and category information of an application. This is based on observation that, every application in the android market has a category assigned to it. We created a large dataset of 25865 applications from Google Play and Genome Project. We demonstrated the effectiveness of our approach on the dataset and showed that exploiting category information indeed improves the performance. As future work, we plan to explore if the static analysis can be combined with dynamic analysis to achieve better performance.

References

1. `https://play.google.com/store?hl=en`
2. `http://developer.android.com/about/index.html`

3. http://en.wikipedia.org/wiki/Mobile-virus
4. http://code.google.com/p/android-apktool/
5. http://code.google.com/p/smali/
6. https://code.google.com/p/dex2jar/
7. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: ACM conference on Computer and communications security. (2011)
8. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy. (2012)
9. https://www.virustotal.com/
10. Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: Security and Privacy in Communication Networks. (2013)
11. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: International conference on Mobile systems, applications, and services. (2011)
12. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Profiledroid: Multi-layer profiling of android applications. In: International conference on Mobile computing and networking. (2012)
13. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: International conference on Mobile systems, applications, and services. (2012)
14. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications. Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa (2009)
15. Barrera, D., Kayacik, H.G., van Oorschot, P.C., Somayaji, A.: A methodology for empirical analysis of permission-based security models and its application to android. In: ACM conference on Computer and communications security. (2010)
16. Di Cerbo, F., Girardello, A., Michahelles, F., Voronkova, S.: Detection of malicious applications on android os. In: Computational Forensics. (2011)
17. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: Trust and Trustworthy Computing. (2011)
18. Hao, H., Singh, V., Du, W.: On the effectiveness of api-level access control using bytecode rewriting in android. In: ACM SIGSAC symposium on Information, computer and communications security. (2013)
19. Zhao, M., Ge, F., Zhang, T., Yuan, Z.: Antimaldroid: An efficient svm-based malware detection framework for android. In: Information Computing and Applications. (2011)
20. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI. (2010)
21. Kim, H., Smith, J., Shin, K.G.: Detecting energy-greedy anomalies and mobile malware variants. In: International conference on Mobile systems, applications, and services. (2008)
22. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: ACM conference on Computer and communications security. (2009)
23. Schmidt, A.D., Bye, R., Schmidt, H.G., Clausen, J., Kiraz, O., Yuksel, K.A., Camtepe, S.A., Albayrak, S.: Static analysis of executables for collaborative malware detection on android. In: ICC. (2009)
24. Sahs, J., Khan, L.: A machine learning approach to android malware detection. In: Intelligence and Security Informatics Conference. (2012)