This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Sarnoff 2016 proceedings.

DroidDeepLearner: Identifying Android Malware Using Deep Learning

Zi Wang, Juecong Cai, Sihua Cheng, Wenjia Li*
Department of Computer Science
New York Institute of Technology (NYIT)
New York, NY 10023, USA
{zwang31, jcai07, scheng10, wli20}@nyit.edu

***Abstract*— With the proliferation of Android apps, encounters with malicious apps (malware) by mobile users are on the rise as vulnerabilities in the Android platform system are exploited by malware authors to access personal or sensitive information with ill intentions, often with financial gain in mind. To uphold security integrity and maintain user confidence, various approaches have been studied in the field of malware detection. As malware become more capable at hiding its malicious intent through the use of code obfuscation, it becomes imperative for malware detection techniques to keep up with the pace of malware changes. Currently, most of the existing malware detection approaches for Android platform use semantic pattern matching, which is highly effective but is limited to what the computers have encountered before. However, their performance degrades significantly when it comes to identifying malicious apps they have never tackled before. In this paper, we propose DroidDeepLearner, an Android malware characteri-zation and identification approach that uses deep learning algorithm to address the current need for malware detection to become more autonomous at learning to solve problems with less human intervention. Experimental results have shown that the DroidDeepLearner approach achieves good performance when compared to the existing widely used malware detection approaches.**

Keywords*— *Android; malware; machine learning; deep learning

I. INTRODUCTION

In recent years, as smartphones become a dominant device of choice for performing most of our daily activities, with Android claiming 82% market share in Q2 of 2015 [1], a rapidly increasing number of Android applications (apps) are being developed to make user experience more convenient. Unfortunately, more convenient usually also means less secure, which is particularly true for Android platform. Android is open source with low barriers to enter; while this may help to grow a huge collection of third-party apps quickly that makes up a major component of the Android experience, it also makes the Android platform an easy target for malware intrusion. According to the mobile threat report released by F-Secure in 2014 Q1 [2], over 95% of malicious apps were distributed on the Android platform. Moreover, as the major sources of Android apps, third-party markets contain a lot of cracked or tampered apps, and there is no sign to indicate whether the apps have been checked for security risks or not in these third-party markets. The lack of security inspection on Android apps intensifies the spreading of malicious apps (malware) on the Android platform. Zhou et al. [3] performed some Android malware analysis by popular security software tools, and the detection rate was only between 20.2% and 79.6%, which clearly indicates an urgent and rapidly growing demand on malware detection for Android applications.

To address the increasingly wide-spread security risks, the Android platform itself provides several security solutions that harden the installation and execution of malware, such as the Android permission system and Google "Bouncer". To perform certain tasks on Android devices, such as sending a SMS message, each Android app has to explicitly request the corresponding permission from the user during the installation process. However, many users tend to arbitrarily grant permissions to unknown Android apps without even looking at what types of permissions they are requesting and thereby significantly weaken the protection provided by the Android permission system. As a result, it is very difficult to limit the propagation of malicious apps by the Android permission system itself in practice. On the other hand, Google "Bouncer" is a service provided by Google Play, the official Android market, in 2012, which aims to automatically scans apps (both new and previously uploaded ones) and developer accounts in Google Play with its reputation engine and cloud infrastructure. Even if Bouncer adds another line of defense for Android security, it still has many limitations. First, Bouncer can only scan Android apps for limited time, which means a malicious app can easily bypass it by not doing anything malicious during the scan phase. Second, no malicious code needs to be included in the initial installer when it gets scanned by Bouncer. In this case, the malicious app can have an even higher probability to evade Bouncer's detection. Once the application passes Bouncer's security scan and gets installed on a real user's Android device, then the malicious app can either download additional malicious code to run or connect to its remote control and command (C&C) server to upload stolen data or receive further commands.

Recently, there have been some research efforts on detecting Android malware by using various machine learning algorithms.

*Wenjia Li is the corresponding author.

Drebin [4] extracts permissions, APIs and IP address as features and uses the Support Vector Machine (SVM) algorithm to learn a classifier from the existing ground truth datasets, which can then be used to detect unknown malware. DroidMat [5] alternatively applies KNN (K-Nearest Neighbor) algorithm to permissions and intents to identify malware. In addition, DoridAPIMiner [6] focuses on providing several lightweight classifiers based on the API level features.

However, Android malicious apps are becoming increasingly difficult for traditional malware detection programs to identify, with the ability to perform code obfuscation. Moreover, during our exploration and analysis to Android apps for dissection studies, we have discovered that some of the benign apps also possess seemingly malicious characteristics, which makes them difficult to tell apart. One of the challenges of this study will be to minimize false positives. Android applications are permission based, and use intent and intent-filters as a mechanism for inter-process communication between functions. Malicious apps can, however, gain access to high security permissions through intent interception or intent spoofing. Or they can make several indirect functions calls that do not require the program itself to explicitly declare the number of permission access to resources required to run the app. These apps claim to be one thing, but quietly perform "extra" API functions in the background, attempting to retrieve personal information, resulting in unauthorized sharing of user information, such as location, preferences, to sensitive information like hardware information or personal information that may have negative financial/safety impact. This is why deep learning is of significant interest in malware detection for Android platform, because it is expected to be able to learn to detect evolving malicious apps that the traditional malware detection approaches cannot cope with.

In this paper we propose *DroidDeepLearner*, an Android malware characterization and identification approach using deep learning algorithm to distinguish malicious Android apps from the benign ones. To validate the correctness and performance of the proposed approach, we have conducted a variety of experiments, and compared the performance of the proposed DroidDeepLearner with that of SVM, which is one of the most effective and widely used algorithms for malware detection.

II. Related Work

Recently, various research efforts have been made on android malware detection, in which static analysis, dynamic analysis, and machine learning based approaches are the three major directions among these research works.

A. Malware Detection based on Static Analysis

The first approach for detecting Android malware is motivated by concepts from static program analysis. Several methods have been proposed that statically inspect mobile apps and disassemble the code. Decompiling and data flow tracking are two main techniques in most of the static analysis methods.

Kirin [7] checks the permission of Android apps for indications of malicious activity. Similarly, Stowaway [8] analyzes API calls to detect over-privileged apps, and RiskRanker [9] statically identifies Android apps with different security risks. There are also some common open-source tools for static analysis such as Smali [10] and Apktool [11], which enable dissecting the content of apps with little effort.

B. Malware Detection based on Dynamic Analysis

Dynamic analysis aims to identify Android malware when Android apps are being executed, and most of them monitor the behaviors of apps in terms of accessing private data or using restricted API calls. For instance, both TaintDroid [12] and DroidScope [13] dynamically monitor Android apps when they are actually executed, where the former focuses on taint analysis and the latter aims to introspection at different layers of Android system.

Dynamic analysis usually cannot be performed on the smartphones themselves to directly identify malware, because they would incur a large amount of computational overhead. Thus, they are mainly used to detect Android malware offline via scanning and analyzing a lot of Android applications. For example, malware detection schemes such as AppsPlayground [14] and DroidRanger [15] have been proposed to dynamically analyze Android apps and detect the possible malicious behaviors they may perform.

C. Malware Detection Using Machine Learning Techniques

It is generally challenging to manually specify and update detection patterns for Android malware in static analysis process. Therefore, some recent research works have been focusing on using various machine learning techniques to automatically extract app features efficiently distinguish malicious apps from benign ones by models without manual operation. For example, DroidMat [5], and DroidAPIMiner [6] build models with different machine learning algorithms on features, including permissions, API calls, and so on.

Moreover, Drebin [4] first pointed out the high efficiency of machine learning techniques on Android malware detection using feature sets from permissions in the manifest.xml and API calls. Permission analysis has been widely used in malware detection, and it has shown high reliability assurance. However, the main challenge is actually finding the "right" permissions to use. If an Android app requests a combination of specific permissions, it is likely that the app is malicious. A typical app can indicate numerous permissions it intends to invoke in the XML manifest file, but not all will be used during initial run time, and do not appear in the source code. To counter this, additional features have to be extracted from the API-level to overcome the shortcomings of permission-based malware detection system [16].

Wang et al. [17] have concluded from their studies that there are indeed certain permissions that are invoked by malicious apps

more frequently than benign ones, enough to make them characteristic features for malicious classification. They profiled the top 40 riskiest Android permissions ranked by correlation coefficient, mutual information and T-test.

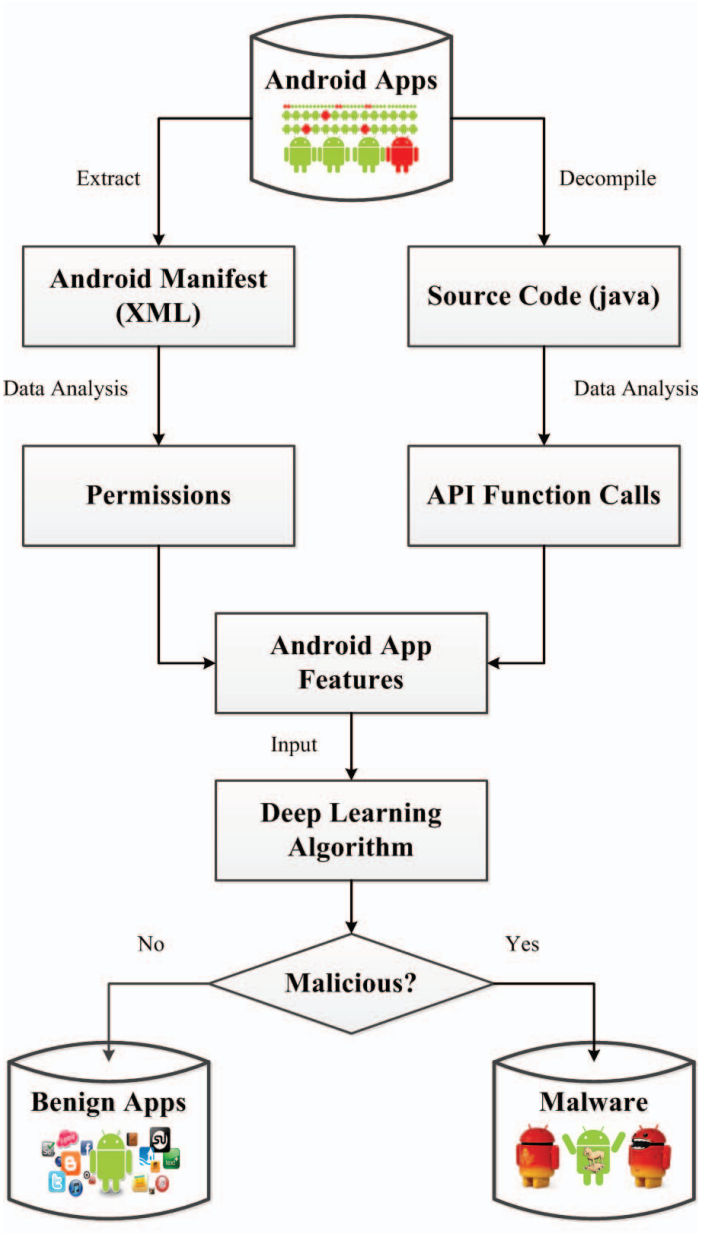

Fig. 1. Overview of the DroidDeepLearner Approach

III. DroidDeepLearner: Deep Learning Based Malware Detection

In this section, we present the proposed malware detection approach in details. The goal of using deep learning algorithm for malware detection in Android platform is so that the machine can teach itself to identify malware and catch malicious intents "just in time", keeping up with how malware evolve in the wild, instead of waiting for manual updates. Deep Learning is the brainchild of Geoff Hinton [18], where the first two layers are Restricted Boltzmann Machine (RBM) and the rest of the hidden layers are Sigmoid Boltzmann Machine (SBM) [19] using the Bayesian form. RBM by nature suffers from the Vanishing Gradient Problem, and the benefit of DBN is that the way it trains data resolves said problem. Training is performed in a bottom-up fashion where the SBMs are trained in pairs, moving up the hierarchy layers, ending with the two RBM layers at the top. This section will explore the use of the Deep-Belief Network (DBN) in malware detection for Android platform.

A. *Scheme Overview*

We first present the overall scheme architecture of the proposed DroidDeepLearner approach, which is shown in Figure 1. As is shown in Figure 1, we aim to obtain two types of features from the Android apps, namely permissions and API function calls. To achieve this goal, the Android apps should be first analyzed to get their corresponding manifest files (*.xml) and the source files (*.java). Then, the permissions can be extracted from the manifest file, and the API function calls are obtained from the source files, accordingly. Once both of them become available, we will integrate them together into an Android app feature set, which serves as the input for the deep learning algorithm for both training and testing purposes. Depending on the classification results by the deep learning algorithm, we can distinguish malicious apps (malware) from the benign ones.

B. *App Decompilation and Feature Extraction*

Android apps are packed into *apk* format, and the features we are interested in are encrypted in the *apk* file, such as permissions, APIs, actions, intents, IP addresses, and URLs. To extract these features, we implement a decoder based on an open source recompilation tool [11], which unpacks apps to readable manifest (*.xml) files. The readable manifest files contain the permissions that an Android app requests, some of which would be potentially risky, such as *android.permission.CALL_PHONE*, etc. The following are some examples of risky Android permissions that have been widely used by Android malware.

1. ***android.permission.CALL_PHONE***: this permission allows an Android app to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed. It is of high danger to grant this permission to an app because a malicious Android app can easily call a premium-rate number (such as a 1-900 number in USA and Canada), and this whole dialing process is automatic and hidden to the mobile phone user.
2. ***android.permission.SEND_SMS***: this permission lets an app send an SMS on behalf of the mobile user, and similar to the phone call permission, it could cost you money by sending SMS to for-pay numbers. Certain SMS numbers work much like 1-900 numbers and automatically charge

your phone company money when you send them an SMS. Therefore, this permission is also very dangerous.

3. ***android.permission.WRITE_EXTERNAL_STORAGE***: when granted this permission, an app can read, write, and delete anything stored on your phone's SD card. Note that this permission is also frequently requested by a lot of legitimate apps, such as camera apps, audio/video apps, and also document processing apps. Thus, it is very important to use additional evidences (such as the combination of other dangerous permissions or API calls) to further distinguish malicious apps from benign ones because both of them can request for this permission.
4. ***android.permission.ACCESS_LOCATION***: this makes the current location information accessible to the app, which may introduce potential privacy leakage when the location information is disclosed to unwanted third-party without the mobile user's consent. However, many benign apps may also ask for the permission to access your current location, such as social network apps, service (such as restaurant) recommendation apps, etc.
5. ***android.permission.READ_CONTACTS***: this let an app read the mobile user's contact data. This may also be dangerous because of the privacy concerns when a mobile user's contact data (including name, phone number, email address, etc.) are accessed and even shared without the user's awareness and approval. Sometimes, benign apps, such as social networking apps and contact management apps may also request for this permission.

From the description of various dangerous Android permissions, we clearly find that many of those permissions can be used by both malicious apps and benign ones. Therefore, it is very important to combine multiple dangerous permission requests as well as other evidences such as API calls to identify malware in an accurate and efficient manner.

Extracting API-calls required 4 steps to access the Java source code files. First, all the apk files had to be unzipped to obtain the class.dex file for each of them. Then, the dex files had to be converted into jar files using dex2jar which was embedded in Santoku [20]. After that, the jar files were converted into java source files using the jd-cli which was downloaded from Github [21]. Finally, the top 20 API features were extracted according to DroidAPIMiner [6], touted as the top 20 most invoked API-calls by malicious apps.

C. The Deep Belief Network (DBN) Model

In general, Deep Belief Network (DBN) is composed of RBM (Restricted Boltzmann Machine) [14] based on the following energy function:

$$E(v,h) = -h^T wv - c^T v - b^T h$$

where matrix w represents the weights on the edge between visible layer and hidden layer, vector v represents data on visible layer, vector h represents data on hidden layer, vector b and c are bias on hidden layer and visible layer respectively [19].

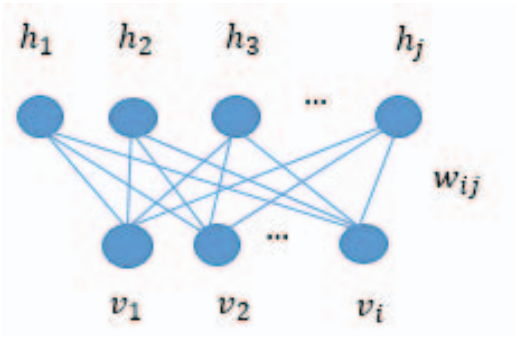

Fig. 1. Overview of the RBM Model

$$p(h_j = 1 \mid v) = sigm(b_j + W_j v)$$

$$p(v_i = 1 \mid h) = sigm(c_i + h^T W_i)$$

where i and j are index of the element of visible layer and hidden layer respectively [19]. Here, the sigmoid function is used as activation function in RBM model.

As illustrated in fig. 1, the RBM will train the input data directly and obtain the values of the neurons on the first hidden layer, then take the first hidden layer as visible layer to train the second hidden layer and repeat the process until all hidden layers are trained. After RBM constructing the layers, DBN will process fine-tuning via k-step CD (Contrastive Divergence) learning.

$$CD_k(\theta, v^{(0)}) = \sum_h p(h \mid v^{(k)}) \frac{\partial E(v^{(k)}, h)}{\partial \theta} - \sum_h p(h \mid v^{(0)}) \frac{\partial E(v^{(0)}, h)}{\partial \theta}$$

where θ is the gradient of the log-likelihood for training pattern [19]. In our RBM model, we use 1-step CD.

IV. Performance Evaluation

To validate the malware detection scheme, we conducted a variety of experiments on actual Android app datasets, and the results show that the scheme is able to accurately identify malware in Android platform.

A. Performance Comparison between Deep Learning and SVM Algorithm

First, we applied the real Android app datasets to both DroidDeepLearner, which is DBN-based, and various versions of the SVM algorithm to compare their performance. We compared the values of precision (P), recall (R), and F1-score for DeepDroidLearner (which uses DBN algorithm) against those of several SVM models using 10-folder cross validation. Precision, recall and F1-score are defined as follows.

The results are shown in Table I, which are sorted by descending order, based on F1-score values. Note that HL here means "Hidden Layer", which is a key feature for DBN and other

deep learning techniques, and SVM type refers to the kernel function it is using in the experiments.

$$P = \frac{\text{Num of Malware Detected}}{\text{Total Num of Android Apps Classified as Malware}}$$

$$R = \frac{\text{Num of Malware Detected}}{\text{Total Num of Actual Malware in the Dataset}}$$

$$F1 = 2 * \frac{P * R}{P + R}$$

As can be seen from Table I, the DBN model performed much better than various SVM models in our experiments. The nu-SVC classifier with polynomial kernel have the best performance among the different SVM models, but their recall scores are much lower when compared to DBN, which means they may have higher count of false negatives on the total number of data asked to be classified, even though they can recognize most of the malicious apps correctly.

Table I. Performance Comparison between Deep Learning and SVM Algorithm

Algorithm	Type	Precision	Recall	F1-score
DBN	2 HL	92.67%	95.40%	93.96%
DBN	1 HL	93.09%	94.50%	93.71%
DBN	5 HL	92.72%	94.10%	93.31%
DBN	4 HL	92.47%	93.90%	93.15%
DBN	3 HL	92.29%	94.00%	93.11%
DBN	10 HL	89.03%	93.20%	91.01%
nu-SVC	Polynomial	97.59%	87.73%	88.74%
C-SVC	Polynomial	86.67%	84.65%	85.65%
C-SVC	Radial basis	80.42%	89.77%	84.84%
C-SVC	Sigmoid	80.08%	89.77%	84.65%
nu-SVC	Sigmoid	84.43%	83.26%	83.84%
nu-SVC	Linear	84.36%	82.79%	83.57%
nu-SVC	Radial basis	81.45%	83.72%	82.57%
C-SVC	Linear	80.57%	79.07%	79.81%
1-class SVM	Polynomial	20.41%	74.42%	32.03%
1-class SVM	Linear	20.17%	77.67%	32.02%
1-class SVM	Sigmoid	20.02%	77.67%	31.84%
1-class SVM	Radial basis	9.56%	26.05%	13.98%

Moreover, the DBN model is more consistent with its performance, with relatively high recalls and relatively high precision to match, which means compared to the SVM algorithm; it can identify more malicious apps correctly with a relatively low and tolerable level of false positives. The only exception is the scenario with 10 HL, which has very poor precision despite a high recall, seemingly suffering from the effects of diminishing returns.

B. Performance of DBN Model in Varying Cases

Second, we also performed some experiments to conduct a closer inspection on the proposed DroidDeepLearner approach. DBN model revealed some interesting details regarding the varying number of layers. We divided the totally 6334 apps into 5 groups. In each group, 1000 benign apps and 4000 malicious apps were used to train data, and 100 benign apps and 200 malicious apps were used as test data. The experimental results are shown in Fig. 2 and Table II.

From Fig. 2 and Table II, we can see that neither precision nor recall improved despite the use of more hidden layers. The DBN models with less than 5 HL all performed better than the model with 10 HL. The precision and F1-score did not show significant difference among cases with less than 5 HL. The average recall performs very well, however, the standard deviation shows a spike when using 5 HL, with a SD score 2 times that of other number of hidden layers and almost 4 times as high as 3 HL case. A high standard deviation shows that the data is widely spread, making it less reliable, while a low standard deviation shows that the data are clustered closely around the mean, making it more reliable. The recall of 3 HL has the lowest standard deviation, so it is the most stable, even though the f1-score indicates that 2 HL is slightly more effective.

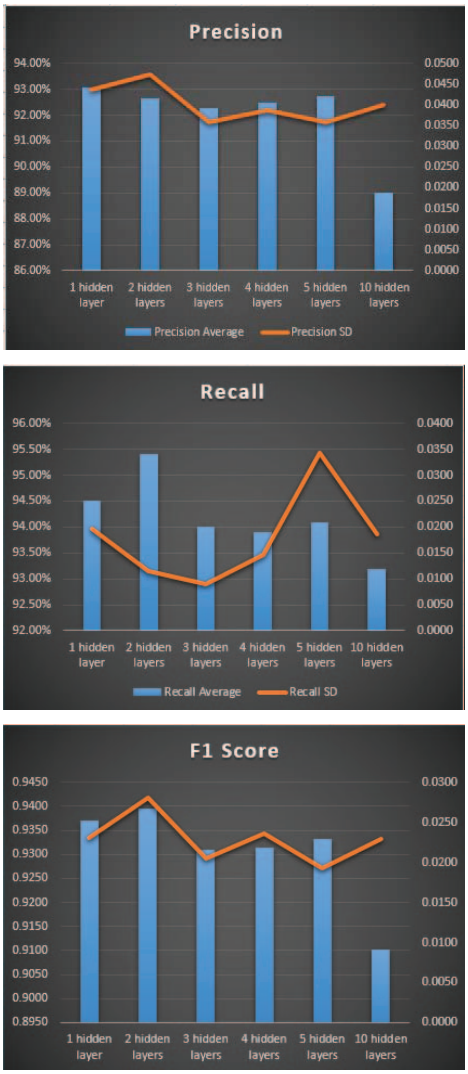

Fig. 2. Precision, Recall and F-1 Score for Different Number of Hidden Layers in DBN

Table II. Effect of Varying Number of Layers for DBN Model

	Precision		Recall		F1-score	
	Average	SD	Average	SD	Average	SD
1 HL	93.09%	0.0436	94.50%	0.0197	93.71%	0.0230
2 HL	92.67%	0.0473	95.40%	0.0116	93.96%	0.0281
3 HL	92.29%	0.0359	94.00%	0.0089	93.11%	0.0206
4 HL	92.47%	0.0386	93.90%	0.0146	93.15%	0.0236
5 HL	92.72%	0.0358	94.10%	0.0343	93.31%	0.0193
10 HL	89.03%	0.0399	93.20%	0.0186	91.01%	0.0230

V. Conclusion and Future Work

In this paper, we propose DroidDeepLearner, a malware detection approach for Android platform using deep learning algorithm, which uses both dangerous API calls and risky permission combinations as features to build a DBN model, which can automatically distinguish malicious Android apps (malware) from legitimate ones. Experiment results show that the proposed scheme is able to identify malware in an accurate manner when compared to the traditional SVM-based solutions.

Based on our experiments, we noticed that the test data set might be fuzzy in real life. If the features of the new test dataset do not totally match those 168 features, association rules discovery is eagerly needed. So the fuzzy association rule mining would also be our research direction in the future.

References

[1] Ramon Llamas, Ryan Reith, Kathy Nagamine (2015). *Smartphone OS Market Share, 2015 Q2*. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of 2012 IEEE Symposium on Security and Privacy (IEEE S&P 2012)*, pages 95–109. IEEE, 2012.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of 2014 Network and Distributed System Security Symposium* (*NDSS 2014*), February 2014.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of *2014 Network and Distributed System Security Symposium* (*NDSS 2014*), February 2014.

[5] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of Seventh Asia Joint Conference on Information Security* (*Asia JCIS 2012*), pages 62–69. IEEE, 2012.

[6] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in android, in *Proceedings of 9th International ICST Conference on Security and Privacy in Communication Networks* (*SecureComm 2013*), pages 86– 103. Sydney, Australia, September 2013.

[7] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security* (*ACM CCS '09*), Chicago, IL, USA, 235-245.

[8] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (*ACM CCS '11*)., Chicago, IL, USA, 627-638.

[9] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (*ACM MobiSys '12*)., Low Wood Bay, Lake District, United Kingdom, 281-294.

[10] J. Freke. An assembler/disassembler for android's dex format. Google Code, https://github.com/JesusFreke/smali, visited May 2016.

[11] C. Tumbleson and R. Wiśniewski, et al. http://ibotpeaches.github.io/Apktool/, visited May 2016.

[12] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." *ACM Transactions on Computer Systems* (*TOCS*) 32, no. 2 (2014): 5.

[13] Lok-Kwong Yan, and Heng Yin. "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis." In *Proceedings of USENIX security symposium*, pp. 569-584. 2012.

[14] Vaibhav Rastogi, Yan Chen, and William Enck. "AppsPlayground: automatic security analysis of smartphone applications." In *Proceedings of the third ACM conference on Data and application security and privacy*, pp. 209-220. ACM, 2013.

[15] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets." In *Proceedings of 2012 Network and Distributed System Security Symposium* (*NDSS 2012*), February 2012.

[16] L. Cen, C. S. Gates, L. Si and N. Li (2015). A Probabilistic Discriminative Model for Android Malware Detection with Decompiled Source Code. In *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 4, pp. 400-412.

[17] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. "Exploring Permission-induced Risk in Android Applications for Malicious Application Detection". *IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY*, pp 1869 – 1882.

[18] Geoffrey E. Hinton, Simon Osindero, Yee-Whye The. "A Fast Learning Algorithm for Deep Belief Nets". *Neural Computation*, Vol. 18, , Massachusetts Institute of Technology, pp. 1527-1554, 2006.

[19] Asja Fischer, Christian Igel. "An Introduction to Restricted Boltzmann Machines". Springer-Verlag Berlin Heidelberg. *CIARP 2012*, LNCS 7441, pp. 14-36.

[20] Santoku (2016). https://santoku-linux.com/

[21] Github: kwart/jd-cmd (2015). *Command line Java Decompiler*. https://github.com/kwart/jd-cmd