Effective Android Malware Detection Based on Deep Learning

Yueqi Jin[1,2,3], Tengfei Yang[1](✉), Yangyang Li[1], and Haiyong Xie[1,3]

[1] National Engineering Laboratory for Public Safety Risk Perception and Control by Big Data (NEL-PSRPC), Beijing, China
yangtengfei1@cetc.com.cn
[2] Key Laboratory of Electromagnetic Space Information, Chinese Academy of Sciences, Beijing, China
[3] University of Science and Technology of China, Hefei, China

Abstract. Android, the world's most widely used mobile operating system, is the target of a large number of malwares. These malwares have brought great trouble to information security and users' privacy, such as leaking personal information, secretly downloading programs to consume data, and secretly sending deduction SMS messages. With the increase of malwares, detection methods have been proposed constantly. Especially in recent years, the malware detection methods based on deep learning are popular. However, the detection methods based on static features have a low accuracy, and others based on dynamic features take a long time, all this limits its scope.

In this paper, we proposed a static feature detection method based on deep learning. It extracts specific API calls of applications and uses DNN network for detection. With the dataset composed about 4000 applications and extremely short time, it can achieve an accuracy rate of more than 99%.

Keywords: Android security · Deep learning · Malware detection

1 Introduction

Mobile phones are now an integral part of our lives, and the most widely used mobile operating system is Android, which accounts for about 85% of the world [1]. Android is used not only on mobile phones, but also on various mobile platforms, such as tablets. Obviously, these mobile devices store a huge amount of users' personal information and even money, so they need to be protected carefully. Is different from the PCs, however, only few years, the popularity of mobile devices in security is not popularized the decades of the PCs, which leads to more and more malwares aimed at the android. According to the report by Symantec [2], 20% Android apps are malwares.

Initially, the idea was to use signature files for detection. But with the outbreak of malwares, this kind method witch depends on huge signature library has gradually become ineffective. Later, people tried running applications with virtual machines to monitor applications' malicious behavior directly ([3,4]). Obviously this method is the most direct, but there may be missing information, and takes a long time. And new malwares increased the delay time in starting up or even directly detect the virtual machine environment for counter-monitoring. Around 2010, the concept of machine learning emerged, and a series of detection methods using machine learning classifier appeared, such as support vector machine(SVM). It extracts some static feature codes in the application or feature logs generated by runtime, and uses various classifier of machine learning to discriminate malwares. This method also achieved great results.

Deep learning is the strengthening of the machine learning. The concept of using deep learning networks to detect Android malware was first proposed in 2014. With AlphaGo beat Sedol Lee in 2016, the value of deep learning was seen by more people, and detection methods based on deep learning were constantly proposed. In general, the effect of deep learning is closely related to features and network.

Our Contribution: We used 1092 sensitive system API calls as features and Deep Neural Network for training. When tested in a dataset consisting of about 4,000 applications, the accuracy was consistently over 99% with minimal time consumption.

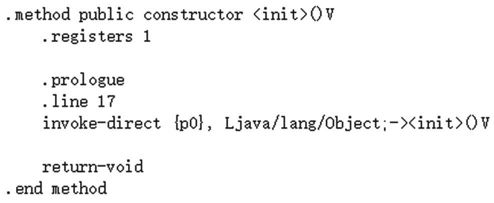
The Reminder of the Paper Is Organized as Follows: Section 2 reviews previous work on android malware detection. Section 3 introduces the application features and DNN network related preliminaries. Section 4 and Sect. 5 gives the implementation details of our experiments and results with evaluation.

2 Related Work

2.1 Static-Feature Detection

The earliest tests using completely static features were published in 2016 ([5–7]). The first two papers are similar in that they use DBN to learn using the permissions requested by applications and the APIs called as features. [7] introduced the concept of API-blocks and achieved 96% accuracy. In 2017, [8] was the first to introduce CNN, which reshaped feature vectors into matrices and trained them by image recognition. This idea also appeared in [9] and [10] of the same year, achieving nearly 100% accuracy.

The biggest drawback of CNN and DBN is the high time complexity, which is not suitable for real-time scanning. In 2018, [11] first used DNN to train the static permissions-api feature, which is also the most similar to our work. The DNN network had a significant speed advantage on the (relative to the image) low-dimensional feature set, and they had a 95.67% F1-score.



```
.method public constructor <init>()V
    .registers 1

    .prologue
    .line 17
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method
```

Fig. 1. Smali code

2.2 Dynamic-Feature Detection

Detection that relies entirely on dynamic features is less frequent because it takes a long time. In 2016 and 2017, [12] adopted the method of emotional analysis to realize malware identification by monitoring system calls and treating system call sequence as text. [13] used Stacked Autoencoders (SAEs) to scan malware and identify new malwares according to the graphical representation of extracted system calls in Linux.

2.3 Hybrid-Feature Detection

[14] in 2014 was the first attempt to apply deep learning to android malware detection. It extracted permissions and sensitive API calls as static feature, used actions monitored by running applications in sandbox as dynamic features, and used DBN for training. This method obtained a good accuracy about 93.5%. In [15], a detection method based on HADM is proposed, which uses Self-Encoder to study the features of applications, and then uses SVM for classification, achieving 95% accuracy. [16] extracts the dynamic and static features, then uses LSTM to analyze, and good results has been achieved.

3 Our Method Description

3.1 Feature Extraction

We can get the "smali" code by decompiling the application file, as Fig. 1. The function calls in the smali code have a format similar to *Ljava/long/Object;-><init>()V*. What we care about is which system sensitive APIs the application calls. For example, *SmsManager;->sendTextMessage* is used to send text messages, *ITelephony$Stub$Proxy;->call* is used to make phone calls. Of course, these APIs are called in benign applications, so we want to use neural networks to help us analyze the underlying information between these calls.

– **Step1:** After decompressing the Apk file to get the "classes.dex" file, we decompile it using the "baksmali" program to get the smali code.
– **Step2:**[1] Document all statements in smali code that contain ";->".

[1] Step 2 is not necessary, but it improves the efficiency of the search in step 3.

– **Step3:** Using our list of sensitive APIs, we organize the application's corresponding smali code into boolean vectors.

Here we will explain step 3. Assumes that our list of sensitive APIs contains n values, in order $[f_1, f_2, ..., f_n]$, so our Boolean vector from step 3 should have n dimensions, let's call that $[x_1, x_2, ..., x_n]$. For each $i \in \{1, 2, ..., n\}$, $x_i = 1$ if and only if f_i appears in the smali code. In other words, the program calls the API f_i. For example, smali code contains f_1 and f_3, not f_2, so the corresponding boolean vector is [1,0,1,...]. We don't care about the order or time of API calls here, just whether they appeared in the smali code.

3.2 Sensitive API List

The core of our results is a more typical sensitive API list. These APIs are selected from the PSCout dataset [17], which contains a large number ofs system APIs and their corresponding permissions. We counted how often these APIs appeared in thousands of benign applications and malwares (The application dataset used here is completely different from the experimental dataset in next Section). We find APIs that appear more frequently in malwares and less frequently in benign applications that make up this list[2]. Compared with other similar methods, we dropped the less distinguishing features of permission and expanded the list of API features in selecting features. These APIs perfectly represent the malicious nature of the program. Based on this list, we can steadily improve the accuracy rate to 99%. At the same time, we can use a simpler network than other methods, thus reducing its running time while maintaining accuracy.

3.3 Training DNN Model

DNN training process, like other networks, is divided into forward propagation and back propagation.

Forward Propagation. For each neuron in each hidden layers in Figure ??, we can think of it as a perceptron like Fig. 2. Its output is the following:

$$b_{in} = \sum_{i=1}^{m} w_i a_i$$

m is the number of neurons in the upper layer. Unlike perceptrons, its output can only be used as the input of the next layer by introducing nonlinear factors through activation function. The RELU function is used in our network. So the output of the neurons in each of the hidden layers of DNN is

$$b_{out} = relu(b_{in}) = relu(\sum_{i=1}^{m} w_i a_i)$$

[2] In appendix, we will show part of this list.

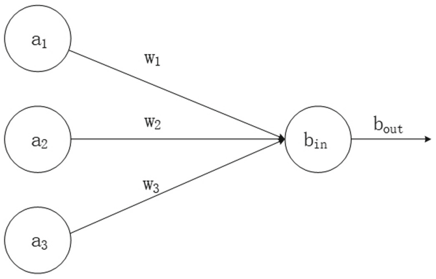


Fig. 2. Propagation

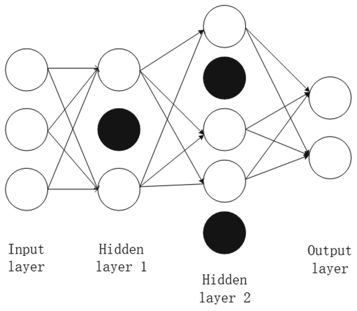


Fig. 3. Dropout

Input the n-dimensional vector obtained in the previous step into the input layer, the output of the output-layer is calculated layer by layer, which completes forward propagation.

Back Propagation. Forward propagation is to sum the output of the upper layer by the weight of edges to get the input of the next layer. Backpropagation is the process of updating the weight of each edge, so that the final value obtained by forward propagation is close to the actual label. The function that measures this approximation is called the loss function. The purpose of back propagation is to reduce the value of loss function. Our network uses the cross entropy loss function. To achieve this goal, we need to use the gradient descent method to change the weight of each edge iteratively. The specific calculation of back propagation is relatively complex, which can be referred to the paper, but will not be described here.

3.4 Dropout

Overfitting is a common problem in deep learning. The specific manifestation of overfitting is that the trained model has a higher accuracy in the training set, but a lower accuracy in the test set. In order to solve this problem, Hinton proposed *Dropout* in 2012 [18], which can effectively alleviate the overfitting phenomenon.

Dropout simply means that during the forward propagation, some neurons stop working at a certain probability (i.e., output value is 0). Compared with Figure ??, black neurons shown in Fig. 3 are inactive. In this way, the generalization performance of the model is stronger and it is less likely to be overly dependent on some localb features. During the training, we set a probability to prevent overfitting, and the difference in accuracy in the test set can be clearly seen during the test. (See next section for details)

4 Experiment Result and Evaluation

4.1 Application Dataset

We selected a dataset composed of about 4,000 applications. Among them, about 2,000 samples of benign applications are from XiaoMi-APPStore (after scanning by anti-virus software, there may still be malwares in them, but we temporarily consider this dataset reasonable), about 2000 malwares come from the Drebin Dataset ([19,20]). We selected almost all kinds of applications and made sure there were no duplicate elements in our dataset. We take 600 applications (300 in each case) as the test set and the rest as the training set.

4.2 Sensitive API List

The list of sensitive APIs refers to the PSCout Dataset [17], which contains about 35000 system APIs and their corresponding permissions in the latest version. We have selected 1092 representative ones for our sensitive API list.

4.3 Runtime Environment

In order to prove that our model is efficient, we did not adopt a more efficient GPU, instead, model training was done using a PC CPU. Our experiment ran in Windows 10, and hardware setting are 4 GB DDR4 RAM and Intel(R) Core(TM) i5-3470 CPU.

4.4 Experimental Results Under Different Parameters

Compared to other papers, we only selected the static feature of API calls, but we chose a larger list of sensitive APIs than others. Here we test three metrics: precision, recall, accuracy, and time consumption.

- **Precision:** Currently classified into positive sample categories, the proportion correctly classified.
- **Recall:** The percentage of all positive samples that are currently assigned to a positive sample category.
- **Accuracy:** The ratio of correctly predicted samples to the total predicted samples.

Table 1. Results with different network model

Malware/Benign = 1 : 1, Train : Test = 3400 : 600, Epochs = 200					
Network	Learning rate	Precision(%)	Recall(%)	Accuracy(%)	Time cost(s)
[500]	0.008	99.67	99.33	99.5	67
[1000, 100]	0.005	99.01	99.67	99.33	114
[1000, 500, 100]	0.0005	99.33	99.67	99.5	155
[1000, 600, 200, 50]	0.00005	98.36	100	99.167	176

Network [500] represents a network with only one hidden layer of 500 neurons, others in a similar way

Table 2. Results with different network model

Malware/Benign = 1 : 1, Train : Test = 3000 : 1000, Epochs = 200					
Network	Learning rate	Precision(%)	Recall(%)	Accuracy(%)	Time cost(s)
[500]	0.008	99.2	99.6	99.4	51
[1000, 100]	0.005	99.4	99.4	99.4	99
[1000, 500, 100]	0.0005	99.01	99.8	99.4	144
[1000, 600, 200, 50]	0.00005	99.2	99	99.1	169

Because the accuracy of the experiment may fluctuate slightly, in this section, we give the most common accuracy in the experiment and its corresponding precision and recall. In appendix 2, we give the accuracy list of 10 consecutive experiments under different conditions.

Table 1 describes the influence of network depth on the results when the number of positive and negative examples is equal. It can be seen from the table that all networks can achieve good results. Our training set has about 3400 samples. When the *BATCH_SIZE* is set to 50, there are about 65 iterations per epoch. In fact, our algorithm has basically converged after the first few iterations of the first epoch, and Fig. 4 below shows the accuracy curve of the first epoch. It can be seen that the accuracy rate after a epoch has been basically stable, and the iteration time of a epoch only takes a few seconds, which is almost negligible.

In the above experiment, Train: Test is 34:6. Next, we adjust this ratio to 3:1. Take 1000 samples as the test set, and the rest as the training set. Also use the above several networks for training. Table 2 shows the experimental results. It can be seen that under such conditions, all networks can still maintain an average accuracy rate above 99%.

In practice, the number of benign applications is obviously greater than that of malwares, so we reduced the proportion of malwares in the training set to conduct training. In the following experiments, $Malware : Benign = 1 : 2$, and the number of samples in the test set is 600. Table 3 shows the experimental results. From the results, the network accuracy rate of 2 or 3 hidden layers is stable at 99%–99.5%.

In Table 4, we further reduce the proportion of malicious programs and set $Malware : Benign = 1 : 5$. It can be seen from the results that the accuracy

Table 3. Results with different network model

Malware/Benign = 1 : 2, Train : Test = 2550 : 600, Epochs = 200					
Network	Learning rate	Precision(%)	Recall(%)	Accuracy(%)	Time cost(s)
[500]	0.008	99.33	99.67	99.5	50
[1000, 100]	0.005	99.01	99.67	99.33	99
[1000, 500, 100]	0.0005	99.33	99.67	99.5	148
[1000, 600, 200, 50]	0.00005	98.03	100	99	159

Table 4. Results with different network model

Malware/Benign = 1 : 5, Train : Test = 2040 : 600, Epochs = 200					
Network	Learning rate	Precision(%)	Recall(%)	Accuracy(%)	Time cost(s)
[500]	0.008	100	98.67	99.33	41
[1000, 100]	0.005	99.33	99	99.167	79
[1000, 500, 100]	0.0005	99.01	99.67	99.33	123
[1000, 600, 200, 50]	0.00005	97.39	99.67	98.5	138

rate is maintained at 98%–99%. In this experiment, due to the low proportion of malwares, the accuracy of the model trained from scratch fluctuates to a certain among. In general, increasing the proportion of benign procedures in training concentration will have a negative impact on the training effect, but the impact is limited.

In addition, we tested the accuracy of the test set after the model was trained from scratch with 20 epochs under various parameters. Table 5, 6, 7, 8 and 9 shows the results of 10 consecutive experiments under each parameter. The experiment of Table 5 uses networks like other tables but without the dropout layer. We can see that after removing the Dropout layer network is not stable. To see from the table, the new model, the minimal number of epochs can achieve good accuracy. This shows that our model can be retrained according to the updated training set at any time without consuming too much time (20 epochs take only a few seconds). In other words, our approach can add new applications to the training set and optimize our model by retraining.

Table 5. Results with network model without dropout

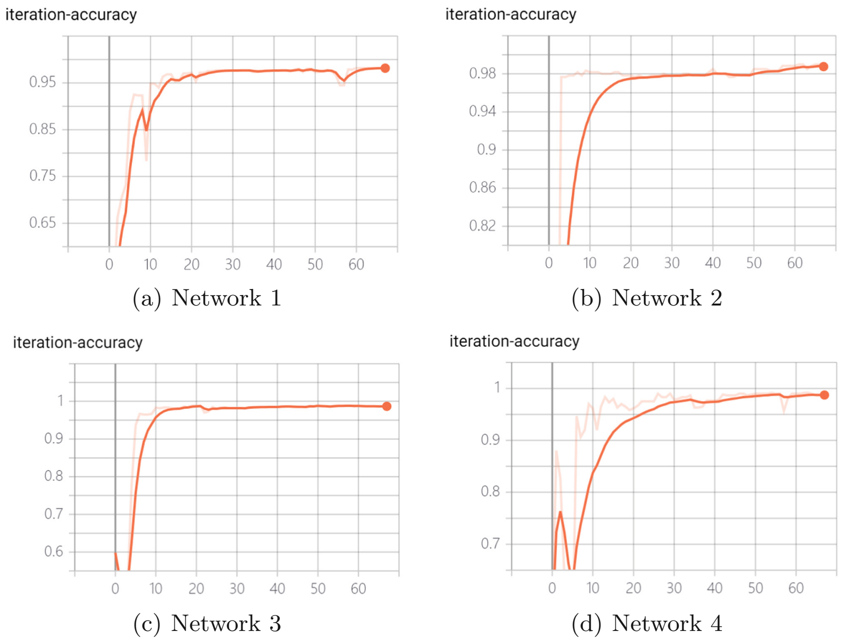
	Malware/Benign = 1 : 1 Train : Test = 3400 : 600, Epochs = 20									
[500]	97.83	98.33	97.67	98	99.167	98.5	97	98.33	97.5	98
[1000, 100]	99	99.167	98.83	98.83	99	99.167	99.167	98.83	99.33	98.5
[1000, 500, 100]	99	98.67	98	98.83	98.33	99	99.167	98.67	99	98.83
[1000, 600, 200, 50]	96.83	98.67	96.83	99.33	98.5	97.34	98.67	97.83	98.67	98.67

Table 6. Results with different network model

	Malware/Benign = 1 : 1, Train : Test = 3400 : 600, Epochs = 20									
[500]	99.67	99.33	99.17	99.5	99.5	99.167	99.83	99.17	99.33	99.33
[1000, 100]	99.17	99.67	99.5	99.5	98.83	99.67	99.5	99.5	99.33	99.33
[1000, 500, 100]	99.5	99.33	99.67	99.5	99.5	99.5	99.5	99.67	99	99.33
[1000, 600, 200, 50]	99.5	99	98.67	98.83	99.167	99.33	98.5	98.5	98.67	99.5

Table 7. Results with different network model

	Malware/Benign = 1 : 1, Train : Test = 3000 : 1000, Epochs = 20									
[500]	99.30	99.30	99.40	99.40	99.50	99.20	99.40	99.40	99.30	99.30
[1000, 100]	99.30	99.50	99.30	99.40	99.50	99.50	99.60	99.40	99.10	99.30
[1000, 500, 100]	99.40	99.40	99.40	99.30	99.60	99.30	99.10	99.20	99.50	99.60
[1000, 600, 200, 50]	99.2	98.5	99	98.7	98.9	99.20	99.40	99	98.40	99.10



(a) Network 1 (b) Network 2

(c) Network 3 (d) Network 4

Fig. 4. Iteration-accuracy at first epoch (Malware/Benign = 1 : 1 , Train : Test = 3400 : 600)

Table 8. Results with different network model

	Malware/Benign = 1 : 2, Train : Test = 2550 : 600, Epochs = 20									
[500]	99.33	99	99	99.167	99	98.33	98.83	99.5	99.167	98.67
[1000, 100]	99.33	98.83	99.5	99.83	99.5	99.33	99.67	99.167	99.5	99.33
[1000, 500, 100]	99.67	99.5	99.67	99.167	99.5	99.5	99.5	99.33	99	99.5
[1000, 600, 200, 50]	99	99	99.167	97.83	98.67	98.67	99	99.167	99.167	98.83

Table 9. Results with different network model

	Malware/Benign = 1 : 5, Train : Test = 2040 : 600, Epochs = 20									
[500]	98.83	98	98	98.83	99	99	99.33	98.5	97.83	98.33
[1000, 100]	99.5	99.5	99.5	99.33	99	98.33	99.167	98.83	99.33	99.33
[1000, 500, 100]	99.33	99.5	99.5	98.83	99.67	99.67	99.5	99.5	99.167	99.167
[1000, 600, 200, 50]	99.167	98.83	97.83	99.33	99.167	99.33	98.33	99	99.33	98.83

5 Conclusion

In this paper, we use DNN to train feature vectors composed of 1092 sensitive APIs to achieve the purpose of detecting android malwares. Compared with other papers, we did not adopt very complex features, but selected more direct API calls as features. The only difference is that we have expanded the list of sensitive APIs, which gives our method an average accuracy of over 98.5%. To summarize, our approach has the following advantages and disadvantages.

5.1 Advantages

– **Features are simple:** All the features can be extracted by reading the smali code once.
– **Less time consuming:** Both feature extraction and model training take very short time. It only takes several iterations for the model from initial training to stable accuracy.
– **Easy to maintain:** After being put into use in the future, applications can be collected to expand training set, which ensures the feasibility of retraining model at low consumption.

5.2 Disadvantages

– **Being attacked:** Static features are easily disguised to allow malwares to evade detection.
– **Exceptional case:** When the proportion of positive and negative examples in training concentration is too large, the accuracy of the model decreases.
– **Behavioral uncertainty:** Unable to determine the specific malicious behavior.

In general, our approach is satisfactory from the experiment. Aiming at the first disadvantage, our future research direction is to introduce dynamic features to identify malwares after disguising. In addition, the third disadvantage is common to all current deep learning based approaches. To address this shortcoming, we have two ideas that may be implemented in the future. The first, combined with dynamic analysis, is to obtain specific malicious behavior based on the calls of sensitive apis in dynamic analysis. The second is to turn a binary task into a multi-categorization task, which classifies malwares into malicious behaviors, which requires more complex datasets. Finally, we think that our results help to promote the development of the android security.

Acknowledgment. This work was supported by the CETC Joint Advanced Research Foundation (No. 6141B08020101).

1 Appendix. Part of our list

Table 10 lists part of the sensitive apis that we use.

Table 10. Part of our sensitive API list

1	android/media/MediaPlayer;->create
2	android/graphics/Picture;-><init>
3	android/os/Handler;->dispatchMessage
4	android/media/MediaPlayer;->prepare
5	android/media/MediaPlayer;->reset
6	android/os/Looper;->loop
7	android/media/Ringtone;->setStreamType
8	android/webkit/WebView;->capturePicture
9	android/webkit/WebView;->destroy
10	android/view/accessibility/AccessibilityNodeInfo;->getChild
11	android/view/accessibility/AccessibilityNodeInfo;->focusSearch
12	android/view/accessibility/AccessibilityRecord;->getSource
13	android/view/accessibility/AccessibilityNodeInfo;->findFocus
14	android/view/accessibility/AccessibilityNodeInfo;->findAccessibilityNodeInfosByText
15	android/view/accessibility/AccessibilityNodeInfo;->performAction
16	android/view/accessibility/AccessibilityNodeInfo;->getParent
17	junit/framework/TestResult;->endTest
18	com/android/internal/telephony/gsm/SmsMessage;->getSubmitPdu
19	com/android/internal/telephony/gsm/SmsMessage;->calculateLength
20	com/android/internal/telephony/cdma/sms/BearerData;->calcTextEncodingDetails
21	android/view/View;->addFocusables
22	android/webkit/WebView;->reload

(*continued*)

Table 10. (*continued*)

23	android/webkit/WebView;->stopLoading
24	android/webkit/WebView;->canGoBack
25	android/widget/AbsListView$LayoutParams;-><init>
26	android/webkit/WebView;->goBack
27	android/webkit/WebView;->canGoForward
28	android/webkit/WebView;->goForward
29	android/webkit/WebView;->saveState
30	android/webkit/WebView;->loadData
31	android/webkit/WebView;->loadDataWithBaseURL
32	android/view/View;->focusSearch
33	android/webkit/WebView;->postUrl
34	android/webkit/WebView;->restoreState
35	android/webkit/WebView;->loadUrl
36	android/webkit/WebView;->getProgress
37	android/webkit/WebView;->pauseTimers
38	android/webkit/WebView;->resumeTimers
39	android/webkit/WebView;->getTitle
40	android/widget/AbsoluteLayout$LayoutParams;-><init>

References

1. Smartphone, O.: Market share, 2015 q2 (2016). IDC [on-line].[dostkep 22.08. 2015]. Dostkepny w: http://www.idc.com/prodserv/smartphone-os-market-share.jsp
2. Wood, P., Nahorney, B., Chandrasekar, K., Wallace, S., Haley, K.: Internet security threat report 2015. Symantec, California (2015)
3. Enck, W., et al.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. Acm Trans. Comput. Syst. **32**(2), 1–29 (2014)
4. Hornyack, P., Han, S., Jung, J., Schechter, S.E., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Acm Conference on Computer & Communications Security (2011)
5. Su, X., Zhang, D., Li, W., Zhao, K.: A deep learning approach to android malware feature learning and detection. In: IEEE Trustcom/BigDataSE/ISPA, vol. 2016, pp. 244–251. IEEE (2016)
6. Wang, Z., Cai, J., Cheng, S., Li, W.: Droiddeeplearner: identifying android malware using deep learning. In: IEEE 37th Sarnoff Symposium, vol. 2016, pp. 160–165. IEEE (2016)
7. Hou, S., Saas, A., Ye, Y., Chen, L.: DroidDelver: an android malware detection system using deep belief network based on API call blocks. In: Song, S., Tong, Y. (eds.) WAIM 2016. LNCS, vol. 9998, pp. 54–66. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47121-1_5
8. Ganesh, M., Pednekar, P., Prabhuswamy, P., Nair, D.S., Park, Y., Jeon, H.: CNN-based android malware detection. In: 2017 International Conference on Software Security and Assurance (ICSSA), pp. 60–65. IEEE (2017)

9. McLaughlin, N., et al.: Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 301–308. ACM (2017)
10. Nix, R., Zhang, J.: Classification of android apps and malware using deep neural networks. In: International joint conference on neural networks (IJCNN), 2017, pp. 1871–1878. IEEE (2017)
11. Li, D., Wang, Z., Xue, Y.: Fine-grained android malware detection based on deep learning. In: 2018 IEEE Conference on Communications and Network Security (CNS). IEEE, pp. 1–2 (2018)
12. Martinelli, F., Marulli, F., Mercaldo, F.: Evaluating convolutional neural network for effective mobile malware detection. Procedia Comput. Sci. **112**, 2372–2381 (2017)
13. Hou, S., Saas, A., Chen, L., Ye, Y.: Deep4maldroid: a deep learning framework for android malware detection based on linux kernel system call graphs. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW), pp. 104–111. IEEE (2016)
14. Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-sec: deep learning in android malware detection. In: ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, pp. 371–372. ACM (2014)
15. Xu, L., Zhang, D., Jayasena, N., Cavazos, J.: HADM: hybrid analysis for detection of malware. In: Bi, Y., Kapoor, S., Bhatia, R. (eds.) IntelliSys 2016. LNNS, vol. 16, pp. 702–724. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-56991-8_51
16. Vinayakumar, R., Soman, K., Poornachandran, P., Sachin Kumar, S.: Detecting android malware using long short-term memory (Lstm). J. Intel. Fuzzy Syst. **34**(3), 1277–1288 (2018)
17. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228. ACM, 2012
18. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors (2012). arXiv preprint arXiv:1207.0580
19. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26 (2014)
20. Michael, S., Florian, E., Thomas, S., Felix, C.F., Hoffmann, J.: Mobilesandbox: looking deeper into android applications. In: Proceedings of the 28th International ACM Symposium on Applied Computing (SAC) (2013)
21. Naway, A., Li, Y.: A review on the use of deep learning in android malware detection (2018). arXiv preprint arXiv:1812.10360