Multimed Tools Appl (2019) 78:3979–3999
DOI 10.1007/s11042-017-5104-0

Android malware detection based on system call sequences and LSTM

Xi Xiao[1] · Shaofeng Zhang[1] · Francesco Mercaldo[2] ·
Guangwu Hu[3] · Arun Kumar Sangaiah[4]

Received: 22 March 2017 / Revised: 14 June 2017 / Accepted: 11 August 2017 /
Published online: 2 September 2017

Abstract As Android-based mobile devices become increasingly popular, malware detection on Android is very crucial nowadays. In this paper, a novel detection method based on deep learning is proposed to distinguish malware from trusted applications. Considering there is some semantic information in system call sequences as the natural language, we treat one system call sequence as a sentence in the language and construct a classifier based on the Long Short-Term Memory (LSTM) language model. In the classifier, at first two LSTM models are trained respectively by the system call sequences from malware and those from benign applications. Then according to these models, two similarity scores are computed. Finally, the classifier determines whether the application under analysis is malicious or trusted by the greater score. Thorough experiments show that our approach can achieve high efficiency and reach high recall of 96.6% with low false positive rate of 9.3%, which is better than the other methods.

Keywords Android malware detection · System call sequences · Deep learning · LSTM language model

1 Introduction

Nowadays, mobile phones prevail in our daily life and Android becomes the most popular operating system on mobile devices [18]. However, the Android platform has been the main

✉ Guangwu Hu
hugw@sziit.edu.cn

1 Graduate School At Shenzhen, Tsinghua University, Shenzhen 518055, China

2 Institute for Informatics and Telematics, National Research Council of Italy, 56124 Pisa, Italy

3 School of Computer Science, Shenzhen Institute of Information Technology, Shenzhen 518172, China

4 School of Computing Science and Engineering, VIT University, Vellore 632014, India

target of malicious attackers for its popularity and openness. The survey [13] reports that millions of malicious applications have been found on mobile phones and 96% of they aim at the Android system. Thus malware detection on Android is very crucial and it has been an active research topic in recent years. Researchers have proposed several methods to identify Android malware with static or dynamic analysis. Most of these approaches adopt machine learning techniques, in which features should be extracted at first. In static analysis, the features are drawn from the .apk files (i.e., the installation package of the Android application). However, if these files are obfuscated, it is very difficult to obtain these features. Dynamic analysis can overcome the drawback of static analysis. In dynamic analysis, the *n*-gram model has been used to extract features from system call sequences such as MALINE [10] and BMSCS [30]. MALINE considers the distance between each pair of system calls in the sequence and defines the summation of the reciprocal of the distances as weight features. BMSCS regards the transition probability of each pair of system calls as features. However, the *n*-gram model only considers relationships between objects in one limited window, leading to an unsatisfied result.

In order to improve the detection result, we propose a dynamic analysis method based on the Long Short-Term Memory (LSTM) language model. In our method, the source code of the application is not required. We use system call sequences issued by the running application to characterize its dynamic behaviors. There is some semantic information in system call sequences as the natural language which can be utilized to distinguish malware from trusted applications. The LSTM model can improve the state of art of *n*-gram models and the standard recurrent neural network language model [25] since it can take advantage of longer contexts. Motivated by the works [22, 25], which employ a language model to find out hidden semantic information in the natural language, we design a deep learning classifier based on the LSTM language model to analyze system call sequences.

In this work, we regard a system call as one word and one system call sequence as a sentence in the language model. The system call sequences from malware and those from benign applications are used to train two different LSTM language models, i. e., the Malicious Model (MM) and the Trusted Model (TM). Then a likelihood probability can be calculated for one sequence with one model. Based on these probabilities, we define the similarity score to describe the extent of the application belonging to MM or TM. By comparing two similarity scores from different models, the application can be classified as malicious or trusted. We evaluate our method from several aspects, including the length of system call sequences, the structure of the model and the time cost. Experiments show that our approach can achieve high efficiency and reach high recall of 96.6% with low False Positive Rate (FPR) of 9.3%, which are better than that of the *n*-gram models [10, 30].

In summary, we make the following innovations:

(1) We analyze the system call sequences from a new point of view. A system call sequence is treated as one sentence in the language model. The semantic information in those sequences is taken into account as that in sentences of the natural language.
(2) We design a deep learning classifier based on the LSTM language model. Unlike the *n*-gram, the LSTM language model leverages all the system calls before the current call in the sequence to predict the next system call. It can capture more context information from the system call sequences than the *n*-gram language model [25]. To the best of our knowledge, it is the first time that the LSTM language model is applied to analyze system call sequences in Android malware detection.

(3) We define a novel criterion, the similarity score, to judge whether the behaviors of one application are malicious or not. Furthermore, the detailed experiments are conducted and the good results verify the reasonability of our definition.

The rest of the paper proceeds as follows. In Section 2, we review current literature. In Section 3, we introduce backgrounds, such as system call sequences and the LSTM language model. In Section 4, our method is explored in detail, including the construction of our new classifier, the training and the testing. In Section 5, we evaluate our approach. Finally, conclusions and the future work are provided.

2 Related works

There are large bodies of research on Android malware identification (e.g., [1–3, 6–10, 12, 16, 17, 20, 21, 23, 24, 28–33]). We only introduce some closely related works. Different works employ different features of applications to detect malware. We classify them into two categories according to their detecting features.

In the first category method, features are extracted from the static source code. These features mainly include Application Programming Interface (API) calls, opcode sequences, permission requests, control flows or data flows. Some methods utilize API calls to analyze behaviors of applications. For example, DroidChain [28] applies API call graphs to analyze relationships between API calls. DroidADDMiner [17] uses API dependent relationships and the machine learning techniques to identify malware. StormDroid [9] combines both static and dynamic analysis. Static features in this method are from the .apk file including some API calls. Its dynamic features are drawn from the executing log files containing the network activity, the file system access, and the interaction with the operating system. Opcode sequences are also taken as detection features. The method [7] designs a Hidden Markov Model (HMM) detector using smali opcode sequences dissembled from the .apk file to recognize malware. The proposed HMM model is trained with lots of code sequences from malware, and then the model calculates a generative probability for each new coming opcode sequence. The new sequences will be classified according to their generative probabilities. Some other works take permission requests as detection features. DREBIN [1] and the work [2] consider permissions requests, and then use the machine learning algorithms to classify applications. Besides that, the data flow and the control flow are also regarded as features in malware detection. The work [16] describes a structured tree based on data characteristic, and employs the similarity between characteristic trees to detect malware. ICCDetector [31] extracts some Inter Component Communication (ICC) patterns from the source code to recognize malicious applications which utilize inter-component communication to launch stealthy attacks. Apposcopy [12] detects malware by checking specified semantics signatures from applications' control flows or data flows. Recently, the possibility to identify the malicious payload in Android malware using a model checking-based approach has been explored in [3, 20, 21]. Starting from the payload behavior definition, these methods formulate logic rules and are tested in a real world dataset composed of Ransomware, DroidKungFu, Opfake families and update attack samples. The above works capture features from .apk files. Their major drawback is that it is very difficult to extract static features from the code correctly when the code is obfuscated.

Other methods log dynamic activities to recognize malware. Some works record the high-level activities such as Internet communication and API call sequences. XDroid [23] assigns the Android application a resource risk based on behavior sequences consisting of the API calls, and sensitive permission requests to build a comprehensive HMM model and calculates a risk score. The work [8] detects malware according to the communication characteristics on the Internet without a complex detection environment. DroidDolphin [29] uses logcat during running time to monitor specific activities (API call sequences), and then it adopts an *n*-gram model to analyze the co-occurrence of these activities. The work [32] improves DroidDolphin with the Convolutional Neural Networks (CNNs). It converts features from DroidDolphin into flattened data with two-dimensional features so that the CNN can be trained to detect malicious behaviors effectively. With the combination property and the proposed flattened input format, it can perform a *k*-skip-*n*-gram dimensionality reduction which learns some patterns comparing to the traditional solutions. It does not capture the long context of the sequences. Other works log the kernel level activities, such as system calls. The method [33] uses both permission requests and system calls to detect malware based on the Feedforward Neural Network (FNN) and the Recurrent Neural Network (RNN). MALINE [10] and BMSCS [30] both use the *n*-gram model to extract features from system call sequences and adopt maching learning techniques to detect Android malware. MALINE considers the distance between each pair of system calls in the sequence and defines the summation of the reciprocal of the distances as weight features. While BMSCS treats the transition probability of each pair of system calls as features. The work [6] considers the system call subsequence as an element and regards the co-occurrence of system calls as features to describe the dependent relationship between system calls. MADAM [24] exploits machine learning techniques to identify malware from different aspects. It only selects 11 system calls to construct feature vectors, which is insufficient and neglects the relationships between system calls as well. Our work is different from those above approaches. As a matter of fact, we design a new classifier based on the LSTM language model, which can predict the next system call in a sequence with all the preceding system calls.

3 Background

In the following section, we provide background notions about the system call sequences and the LSTM language model adopt in the work.

3.1 System call sequence

Android is a Linux-based, open-source, mobile phone platform. The system call is the interface provided by the Linux kernel. In Android, functions of system calls mainly include hardware-related services (using a microphone, accessing a disk drive or a camera, etc.), creation of new processes, and so on [26]. Malicious applications and trusted applications have different behaviors, such as that malicious Android applications request more permissions or access sensitive resources more frequently. What's more, the individual system call cannot be assumed to be independent for that a behavior on the Java code level often involves several system calls sequentially. It is essential to capture which system calls an application involves and the dependencies among those system calls. The system call sequences reveal the dynamic behaviors of applications and different system call sequences reflect different behaviors. In consequence, system call sequences can be used to extract features for malware detection.

3.2 LSTM language model

The statistical language model can solve the prediction problem in sequential data by constructing a language model.

The *n*-gram model cannot catch the pattern not in the training set because of the curse of dimensionality [5]. The work [5] introduces a neural probabilistic language model which can improve the state of art of *n*-gram models since it can take advantage of longer contexts. In the feed forward network, a fixed number of preceding words are used to predict the next one [22]. For example, a sentence $W_1^N = (w_1, \ w_2, w_3, \dots, w_N)$ is factored as (1):

$$p\left(W_1^N\right) = \prod_{m=n}^{N} p\left(w_m | W_{m-n+1}^{m-1}\right) \tag{1}$$

where W_1^N is a sentence from w_1 to w_N, w_m is the *m*th word; W_{m-n+1}^{m-1} is a subsentence from w_{m-n+1} to w_{m-1}.

The work [22] proposes an RNN language model which makes greater progress than the previous works. The RNN does not use the limited size of context and the information can cycle in these networks for an arbitrarily long time by using recurrent connections. Hidden units are fed back to themselves, and they can learn some prior patterns. The recurrent neural network is a rich structure which can be highly context-dependent [11].

In the RNN, a sentence W_1^N can be factored as (2):

$$p\left(W_1^N\right) = \prod_{m=2}^{N} p\left(w_m | W_1^{m-1}\right) \tag{2}$$

A typical structure of the RNN is illustrated in Fig. 1. It includes three layers, the input layer, the output layer, and the hidden layer. The hidden layer consists of many self-connected units.

However, it is claimed that learning long-term dependencies by the stochastic gradient descent can be quite difficult with the standard RNN [4]. The problem is addressed by the LSTM network, which redesigns the RNN. The LSTM network replaces the units in the hidden layer in the RNN with memory blocks [14]. The work [25] introduces the LSTM

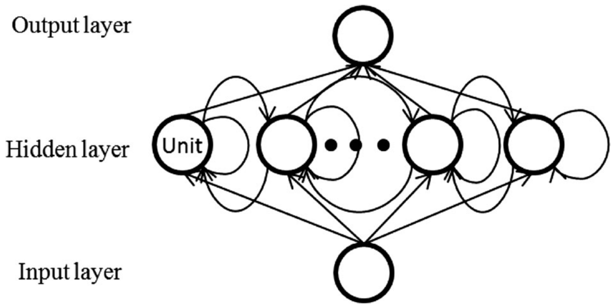


Fig. 1 Structure of RNN

network to model language and achieve great improvements over the standard recurrent neural network. The theoretical proof of the LSMT model can be found in [4, 5, 11, 14].

The structure of the memory block (unit) in the LSTM is shown in Fig. 2.

There are three gates in the memory block of the LSTM network, including the input gate, the output gate, and the forget gate. The input gate and the output gate multiply the input and the output of the cell while the forget gate multiplies the previous state of the cell. The activation function *'f'* is usually the logistic sigmoid. The gates allow the LSTM memory cell to store information for a long time.

In the neural network language model, the input words are encoded by 1-of-*k* coding where *k* is the number of different words in the vocabulary. For example, if there are *k* different words in a sentence, according to 1-of-*k* coding, each different word is represented by a vector with the length of *k*. Only one element is 1 and the left elements are 0 in the vector. As the training criterion, the cross entropy error is used which is equivalent to maximum likelihood. The LSTM adopts the forward algorithm and the Back Propagation Through Time (BPTT) backward algorithm [14].

4 Our method

Firstly, we obtain ten system call sequences for each application. One system call sequence can be regarded as one sentence. We use sequences from malware to train the Malicious Model (MM), and sequences from trusted applications to train the Trusted Model (TM). Indeed we adopt the LSTM classifier to classify the system call sequence. Finally, malware can be identified according to the classification results. The architecture of our method is illustrated in Fig. 3. The first part is to track of system call sequences, and the second one is the LSTM language model classifier.

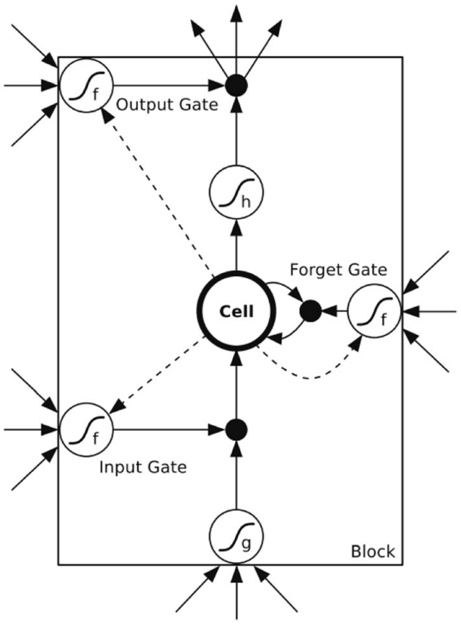


Fig. 2 Structure of the memory block

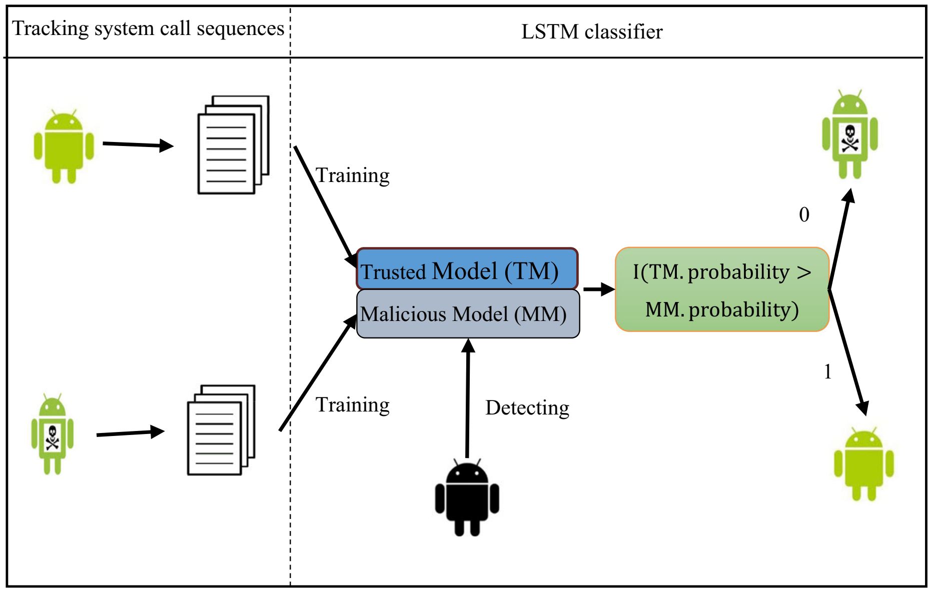


Fig. 3 Architecture of our work

4.1 Tracking system call sequences

Monkey and *Strace* scripts are used to track system call sequences. The applications run on a real mobile device, and *Monkey* generates a number of User Interface (UI) interactions and system events during the execution. Then we use *Strace* to obtain the system call sequence of the running application. Repeating this process ten times, we get ten system call sequences from each application, and these system call sequences reflect dynamic behaviors of applications. In fact, any actions performed by applications or the Android framework are eventually translated into system call sequences.

4.2 LSTM classifier

We design a LSTM language model-based classifier. The classifier consists of two models: one model trained by real-world malware and another one by real-world trusted applications. Both models are LSTM neural networks with the same architecture but different parameters.

The LSTM network is composed of three parts, the input layer, the hidden layers, and the output layer. There are several hidden layers and each layer has many memory blocks shown in Fig. 2. The input layer is a vector encoded with 1-*k* coding, and the output layer is a vector of the probability distribution. In our classifier, one system call is treated as a word in the language model and one system call sequence is taken as a sentence.

As in the Natural Language Processing (NLP), each system call is encoded by 1-*k* coding. By this way, we use the LSTM language model to analyze the system call sequence. For one system call sequence, we predict each next system call according to all the preceding system calls in the sequence. The probability of the system call sequence S_1^N can be factored as follows:

$$p\left(S_1^N\right) = \prod_{m=2}^{N} p\left(s_m | S_1^{m-1}\right) \tag{3}$$

In the above equation, S_1^N is a system call sequence with the length of N, in which s_m is the mth system call. S_1^{m-1} is the subsequence from s_1 to s_{m-1}.

During the training phase, we employ the trusted dataset and the malicious one respectively to train two models which are thus called the Trusted Model (TM) and the Malicious Model (MM). In the testing step, two similarity scores are calculated from the two models for one system call sequence, and we can identify an application by comparing these two scores.

4.2.1 Training

Our classifier is based on two LSTM neural networks. As a matter of fact, we train the neural networks at first. We use all the ten sequences of each application in the training set to train the network. Suppose that there are 5 different system calls labeled as q_1, q_2, q_3, q_4 and q_5. We extract a system call sequence as $q_1, q_5, q_3, q_5, q_2, q_5, q_3, q_2, q_5, q_4, q_2$. According to 1-$k$ coding [25], each system call is encoded into a vector. $q_1, q_2, q_3,$ q_4 and q_5 are encoded into V_1: $[1, 0, 0, 0, 0]^T$, V_2: $[0, 1, 0, 0, 0]^T$, V_3: $[0, 0, 1, 0, 0]^T$, V_4: $[0, 0, 0, 1, 0]^T$, V_5: $[0, 0, 0, 0, 1]^T$.

Firstly, system calls encoded as vectors are fed into the LSTM neural network. After feeding one vector into the network, then we get an output vector of the probability distribution V_p according to the forward pass algorithm [14]. A multiplicative input gate protects the memory contents stored in the memory cell from perturbation by irrelevant inputs. A multiplicative output gate protects other units from perturbation by currently irrelevant memory contents stored in the memory cell and a multiplicative forget gate protects other units from perturbation by previous irrelevant memory contents [15]. At last the next predicted system call can be obtained according to V_p. This process can be factored as (4):

$$Predicted\ system\ call = \left(q_i | v_i = max\left(V_p\right), 1 \leq i \leq n\right) \tag{4}$$

where v_i represents the ith element in V_p, which is equal to $p(q_i|subsequence\ before\ q_i)$, and n is the number of different system calls or the length of V_p.

Secondly, the loss function $Loss(V_p, vector\ of\ target\ system\ call)$ describes the difference between the predicted next system call and the target system call in the sequence. Then the backward pass algorithm [14] calculates the error and updates the weights of the network. After several iterations, the weights converge, and the training phase is stopped.

For example, for the above system call sequence $q_1, q_5, q_3, q_5, q_2, q_5, q_3, q_2, q_5,$ q_4, q_2, we feed $[1, 0, 0, 0, 0]^T$ into the network and get a probability distribution of V_p: $[0, 0.1, 0.8, 0.1, 0]^T$ is generated. Consequently, the next system call is determined as q_3 by the neural network according to (4). In fact in the beginning of the sequence the system call after q_1 is q_5, and the vector of the target system call, q_5, is $[0, 0, 0, 0, 1]^T$.Then the loss function $Loss([0, 0.1, 0.8, 0.1, 1]^T, [0, 0, 0, 0, 1]^T)$ demonstrates the difference between the predicted output and the actual system call in the sequence. Note that, all the ten system call sequences from the application in the training set are fed into the network to do the training.

4.2.2 Testing

During the testing phase, firstly, in one trained model for each system call in the system call sequence, the forward pass algorithm [14] can calculate a probability $p\left(s_m|Q_1^{m-1}\right)$. After processing the whole system call sequence, we get the probability $p\left(Q_1^N\right)$ of the whole system call sequence in the model according to (3). Both the TM and the MM are used to calculate the probability of each system call sequence. We repeat this process ten times, each time for one sequence of the application.

Secondly, we define the similarity score to depict the similarity degree between the detected application and the corresponding application category. In one model, let p_i be the probability of the ith system call sequence and C be the number of system call sequences for an application. The similarity score in the method is defined as

$$Similarity\ score = exp\left(\frac{\sum_{i=1}^{C} logp_i}{C}\right) \tag{5}$$

C is equal to 10 in our work as a case, and other numbers are also applicable. We collect 10 execution traces for each application, in order to mitigate the occurrence of rare conditions and to stress several running options of the application under analysis. Furthermore, considering the different execution of the same application under test, we increase the possibility to effectively activate the malicious payload. In addition, even in the application under test crashes, we mitigate this risk with the other 9 runs.

At last, we do the classification based on the similarity scores. If the similarity score of an application from the TM is greater than that from the MM, the application is determined as trusted. Otherwise, the application is determined as malicious as (6):

$$I(TM.Similarity\ score > MM.Similarity\ score) = \begin{cases} 0, & malicious \\ 1, & trusted \end{cases} \tag{6}$$

For example, suppose that ten system call sequences ($Q_1,\ Q_2,\ Q_3,\ Q_4,\ Q_5,\ Q_6,\ Q_7,\ Q_8,\ Q_9,\ Q_{10}$) are extracted from one application. We use the TM and the MM to calculate two probabilities for each sequence. Then two kinds of probabilities are obtained. Based on these probabilities, we calculate two similarity scores from the two models using (5). At last, the application can be judged by comparing two scores according to (6).

5 Evaluations

In this section, we describe the result of our evaluation process in order to demonstrate the effectiveness of the our model.

5.1 Dataset and metrics

In our experiments, the real-world dataset includes 3567 malicious applications and 3536 benign ones. The malicious applications are gathered from the Drebin project's dataset [7], a very well-known collection of malware used in many scientific works, which includes the most widespread Android families. In order to download trusted applications, we crawl the Google Play market using an open-source crawler [19]. The trusted dataset contains samples belonging to all the categories available on the Google official market. In order to apply the ten-fold cross validation, we divide the dataset into ten equalized subsets. A single subset is taken as the validation data to test the model, while the remaining 9 subsets are used as the training data. We repeat this process 10 times, and each of the 10 subsets of data is used once as the validation data. In detail, 3210 malicious applications and 3182 trusted applications are chosen as the training set, and 357 malicious applications and 354 trusted applications are taken as the test set. In order to mitigate the occurrence of rare conditions, we run one application 10 times and get 10 system call sequences traces for each application.

We send two different kinds of events to the application under analysis: the first ones are a set of random GUI events directed to the application, while the second kind of events are the system events, i.e., events directed to the Android operating systems not directed to the application. Only the second kind of events are able to activate the malicious payload [34, 35]. The random GUI events are just used to simulate the user behavior.

We configure Monkey to send 2000 random UI events in one minute and to stimulate the Android operating system with the following events (one every 5 s starting when the application under analysis is in foreground): (1) reception of SMS; (2) incoming call; (3) call being answered (to simulate a conversation); (4) call being rejected; (5) network speed changed to GPRS; (6) network speed changed to HSDPA; (7) battery status in charging; (8) battery status discharging; (9) battery status full; (10) battery status 50%; (11) battery status 0%; (12) boot completed. This set of events is selected because it represents an acceptable coverage of all possible events which an app can receive. Moreover, this list takes into account the events which most frequently trigger the payload in Android malware [34, 35].

Finally, we get 71,030 system call sequences in total, in which there are 129 different system calls.

There are mainly four metrics, i. e., precision, recall, accuracy, and FPR are used to evaluate a detection system.

Precision is defined as (7):

$$precision = \frac{TP}{TP + FP} \tag{7}$$

We define recall as the following formular:

$$recall = \frac{TP}{TP + FN} \tag{8}$$

Accuracy can be described as (9):

$$accuracy = \frac{TP + TN}{TP + NP + TN + FN} \tag{9}$$

False Positive Rate (FPR) is defined as follows:

$$FPR = \frac{FP}{FP + TN} \tag{10}$$

In the above formulas, True Positive (TP) indicates the number of malware identified correctly; False Negative (FN) is the number of malware taken as trusted applications; False

Positive (FP) is the number of trusted applications detected as malware; True Negative (TN) is the number of trusted applications identified correctly. Recall represents the ability to identify malware correctly in the malicious applications. FPR measures the percentage of trusted applications that are incorrectly labeled as malicious ones. It is more dangerous that we take a malicious application as a trusted application, so recall is more concerned. Precision and accuracy are comprehensive measurements combining with TP and FP.

5.2 Effect of the length of system call sequences

In this part, first of all we analyze similarities and differences between system call sequences and natural language sentences, and then we investigate the effect of different lengths of system call sequences.

Both system call sequences and natural language sentences can be treated as sequences. But one natural language sentence is always shorter than a system call sequence. If we do not execute an application to extract a long enough sequence, the system call sequence cannot cover application's malicious or normal behaviors. Hence, it is necessary to get a long system call sequence. There are many different words in natural language. However, only hundreds of different system calls are available in the Android operating system. In our dataset, there are 129 different system calls, and the length of the longest sequence is about 200,000. In a typical natural language dataset, the number of different words is 10,000, and the maximum length of the sentence is about 80. If we use rectangles to describe the both datasets we can figure the differences between these two datasets apparently. The length of the rectangle represents the maximum length of sentences (sequences), and the width represents the number of words (system calls). Then the difference between system call sequences and natural languages can be shown in Fig. 4:

In our LSTM language classifier, we construct the network with 4 hidden layers and 1000 units in each layer. Then the classifier is used to do the detection under different maximum lengths of system call sequences, i.e., 50, 100, 500, 1000, 5000, 10,000, 50,000. The results are shown in Table 1 and Fig. 5.

Figure 5 and Table 1 indicate that when the length is 100 the classifier can achieve high recall of 96.6% with precision of 91.3%, accuracy of 93.7% and low FPR of 9.3%. Recall and accuracy metrics in other cases are inferior to those of 100. When the system call sequence has no more than 50 system calls, accuracy and precision are the worst. The reason is that the sequences do not cover all the malicious and normal behaviors when the maximum length is 50. The network is unable to memorize more information and it is disturbed by additional noises when the sequence is very long, such as that maximum lengths of sequences are 500, 1000, and 5000. Fig. 5 and Table 1 indicate that our classifier can distinguish malware from trusted applications and achieves good precision.

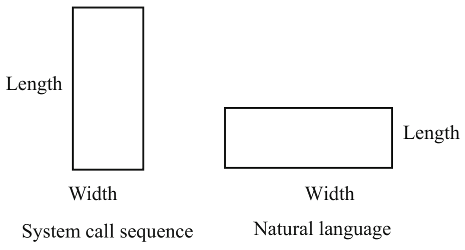


Fig. 4 Comparison of two datasets

Table 1 Results under different lengths of system call sequences

Length	50	100	500	1000	5000	10,000	50,000
Precision	0.854922	0.912698	0.938953	0.927954	0.916201	0.909091	0.913408
Recall	0.92437	0.966387	0.904762	0.901961	0.918768	0.92437	0.915966
Accuracy	0.883263	0.936709	0.922644	0.915612	0.917018	0.915612	0.914205
FPR	0.158192	0.09322	0.059322	0.070621	0.084746	0.09322	0.087571

5.3 Effect of the number of hidden layers

In this part, we want to know if the number of hidden layers affects the detection results. We analyze the performance of the LSTM classifier under different number of hidden layers in the network from 1 to 6. Each layer of all the networks has 1000 hidden units, and the maximum length of the system call sequences is 100. The results are shown in Table 2. As we can see in Table 2, the highest recall of 96.6% and accuracy of 93.7% are achieved when there are 4 layers. When the number of hidden layers is less than 4 the neural network cannot catch enough useful information from the sequence. While if the number is greater than 4, the network is overfitting. Thus, the network with 4 layers is appropriate for our work.

5.4 Effect of the number of hidden units in each layer

To find out the relationship between the number of hidden units in each layer and the detection performance in the LSTM classifier, we set the number of hidden layers in the network as 4, and alter the number of hidden units in each layer. Then we assess

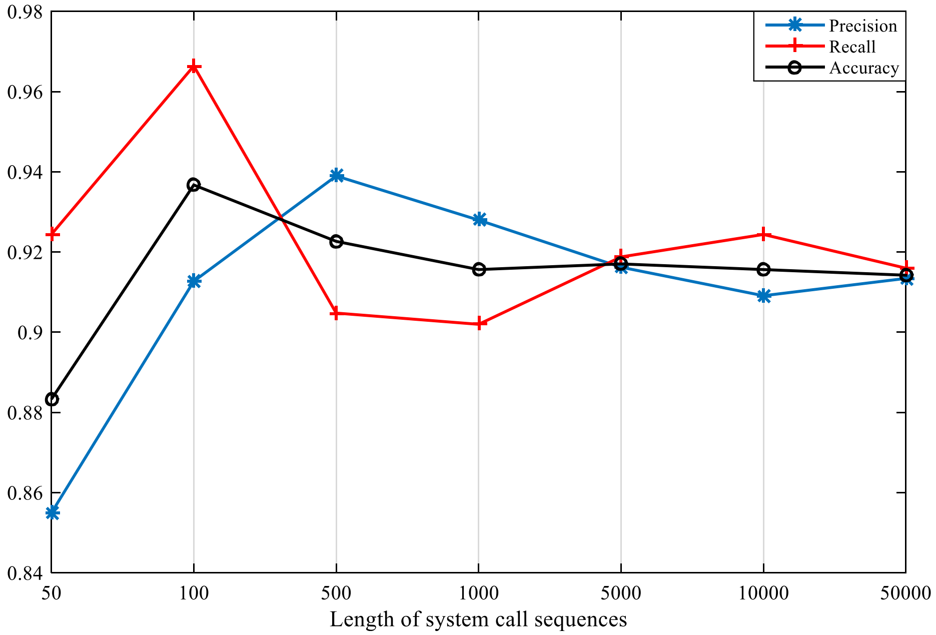


Fig. 5 Precision, recall, and accuracy under different lengths of system call sequences

Table 2 Metrics under a different number of hidden layers

Number of hidden layers	1	2	3	4	5	6
Precision	0.896739	0.897297	0.89011	0.912698	0.902439	0.915361
Recall	0.92437	0.929972	0.907563	0.966387	0.859944	0.817927
Accuracy	0.908597	0.911392	0.897328	0.936709	0.895921	0.870605
FPR	0.107345	0.107345	0.112994	0.09322	0.067797	0.076271

these networks on the system call sequences with maximum length of 100. The results are shown as follows:

Table 3 demonstrates that when the number of hidden units is 1000 in each layer our method gets the highest recall of 96.6% and the highest accuracy of 93.7%. The reason is that when the number is less than 1000, the model is underfitting, and when the number is more than 100 the model is overfitting.

5.5 Time cost

In this section, we discuss the time cost of the classifier including the time of building the classifier and that of classifying one application. Our model is implemented with *Tensorflow* [27] framework, and GPU is used to speed up. The overhead time mainly depends on the complexity of the network. We set the maximum length of system call sequences to 100.

At first, we evaluate the time of building the classifier under a different number of hidden layers in the network. The number of hidden units in each layer is the same, which is 1000. The results are displayed in Fig. 6. It can be seen from the figure that when there is only one hidden layer in the network, the classifier takes 7900 s to convergence after 15 iterations. When the number of hidden layers is 3, the time cost is about 18,500 s for the classifier to converge. The time cost is about 35,000 s when the number is 6. It can be drawn a conclusion that the time used to build a classifier increases as the number of hidden layers grows.

Furthermore, we use the same classifier to investigate the average time to identify one application, and the results are shown in Fig. 7. When there is only one hidden layer, the average time to identify one application is 0.082 s. The time becomes 0.193 s when the number of hidden layers is 3. If the number is 6, the time cost is 0.36 s. The average time used to deal with one application increases as the number of hidden layers grows. Our method can identify an application within 1 s which indicates that our method achieves high efficiency.

In addition, we also record how long it takes to build a classifier with different number of hidden units in each layer when the number of hidden layers is 4. The number of hidden units is the same in each layer, ranged from 200 to 1200 with the interval 200. The results are displayed in Fig. 8. It is obvious that the more hidden units in each layer, the more average

Table 3 Metrics under a different number of hidden units in each layer

Number of hidden units	200	400	600	800	1000	1200
Precision	0.918079	0.880435	0.903047	0.875325	0.912698	0.862338
Recall	0.910364	0.907563	0.913165	0.943978	0.966387	0.929972
Accuracy	0.914205	0.891702	0.907173	0.90436	0.936709	0.890295
FPR	0.08192	0.124294	0.09887	0.135593	0.09332	0.149718

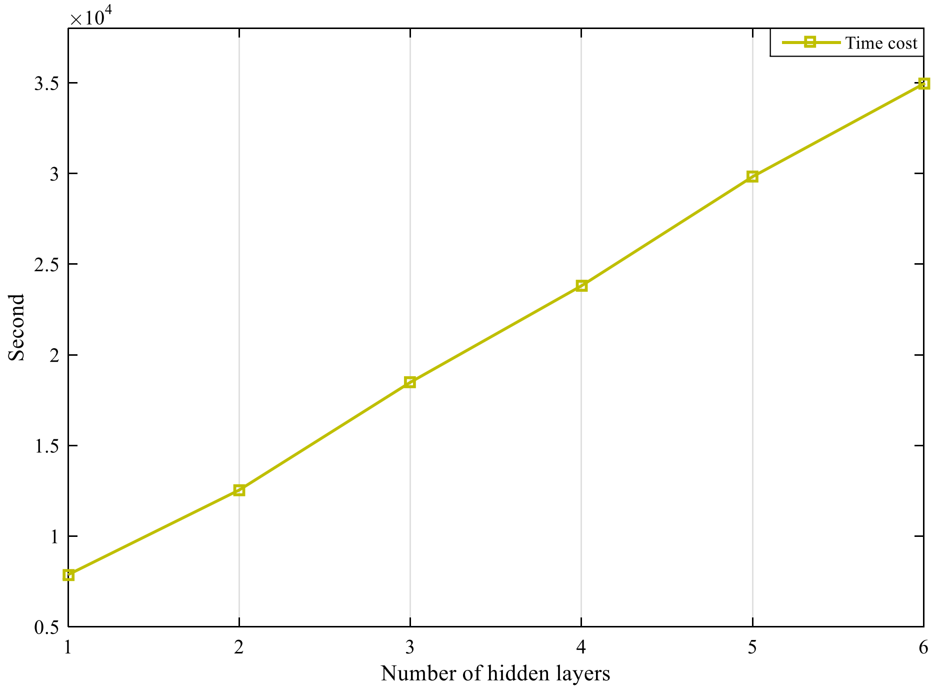


Fig. 6 Time of building a classifier under different number of hidden layers

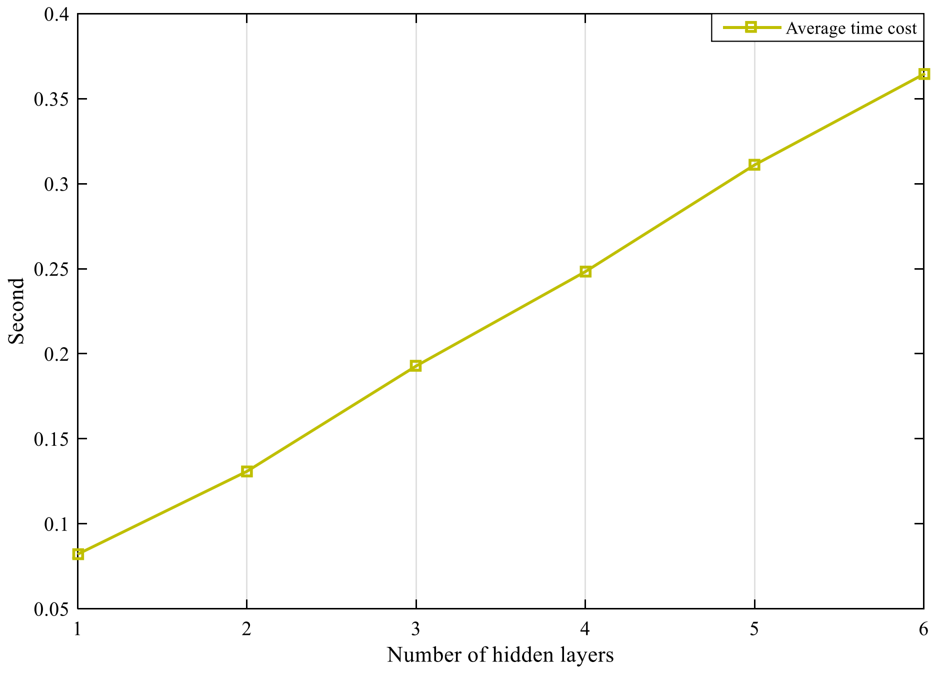


Fig. 7 Average time of analyzing an application under a different number of hidden layers

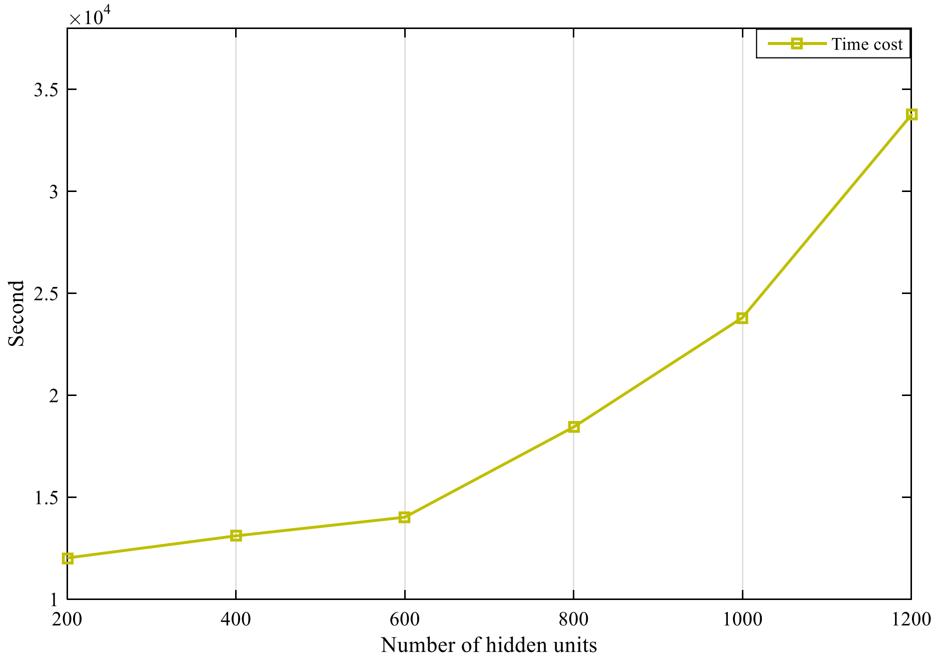


Fig. 8 Time of building a classifier under a different number of hidden units

time is required to build a classifier. When the number of hidden units is 1200, it cost about 33,000 s to build one classifier. The time cost is acceptable.

5.6 Comparison with *n*-gram models

In order to make a comparison, we implement two *n*-gram models, MALINE [10] and BMSCS [30], which are typical detection methods using system call sequences. We evaluate them with the same dataset as ours. MALINE considers the dependent relationship between system calls. It defines the dependent weight $w_{g,h}$ between each pair of different system calls q_g and q_h as follows:

$$w_{g,h} = \begin{cases} 0, & if\ g = h \\ \sum_{s_i = q_g, s_j = q_h,}^{i<j<len} \frac{1}{d(s_i, s_j)}, & otherwise \end{cases} \tag{11}$$

In (11), s_i is the *i*th system call in a sequence, *len* represents the length of the system call sequence, and $d(s_i, s_j)$ is the number of system calls between s_i and s_j, which equals to $j-i$.

BMSCS takes each system call sequence as a homogeneous stationary Markov chain and feeds the transition probability between each pair of system calls into the back-propagation neural network to do the classification.

We connect ten system call sequences from each application into one sequence for simplifying the comparison and do the comparison in view of precision, recall, accuracy, and FPR. Table 4 shows the four metrics of MALINE, BMSCS and our LSTM classifier under

Table 4 Comparison between LSTM classifier and MALINE

Method	Length	50	100	500	1000	5000	10,000	50,000
MALINE	Precision	0.79427	0.82065	0.81962	0.83244	0.82414	0.82368	0.82368
	Recall	0.85434	0.84593	0.86554	0.87675	0.87955	0.87675	0.87675
	Accuracy	0.81575	0.82981	0.83685	0.84950	0.84528	0.84388	0.84388
	FPR	0.22316	0.18644	0.19209	0.17796	0.18926	0.18926	0.18926
BMSCS	Precision	0.7815	0.7899	0.8099	0.7917	0.8021	0.8107	0.8145
	Recall	0.8515	0.8739	0.8711	0.8515	0.8403	0.8515	0.8487
	Accuracy	0.8059	0.82	0.8326	0.8129	0.8158	0.8256	0.827
	FPR	0.2401	0.2345	0.2062	0.226	0.209	0.2006	0.1949
LSTM classifier	Precision	0.90084	0.94767	0.93390	0.91977	0.91061	0.93063	0.91573
	Recall	0.89075	0.91316	0.91036	0.89915	0.91316	0.90196	0.91316
	Accuracy	0.89592	0.93108	0.92264	0.90998	0.911392	0.91701	0.91420
	FPR	0.09887	0.05084	0.06497	0.07909	0.09039	0.06779	0.08474

different lengths of sequences, and Fig. 9 plots three lines about precision, recall and accuracy respectively.

From the table and the figure, when the length of the system call sequence is 1000, MALINE gets the highest precision of 83.2%. BMSCS gets the highest precision of 81.45% with the length of 50,000. When the length becomes 100, our LSTM classifier reaches the highest recall of 91.3% and the highest precision of 94.76% with the lowest FPR of 5%. When the length is 5000, MALINE obtains the highest recall of 88% with FPR of 18.9%. BMSCS

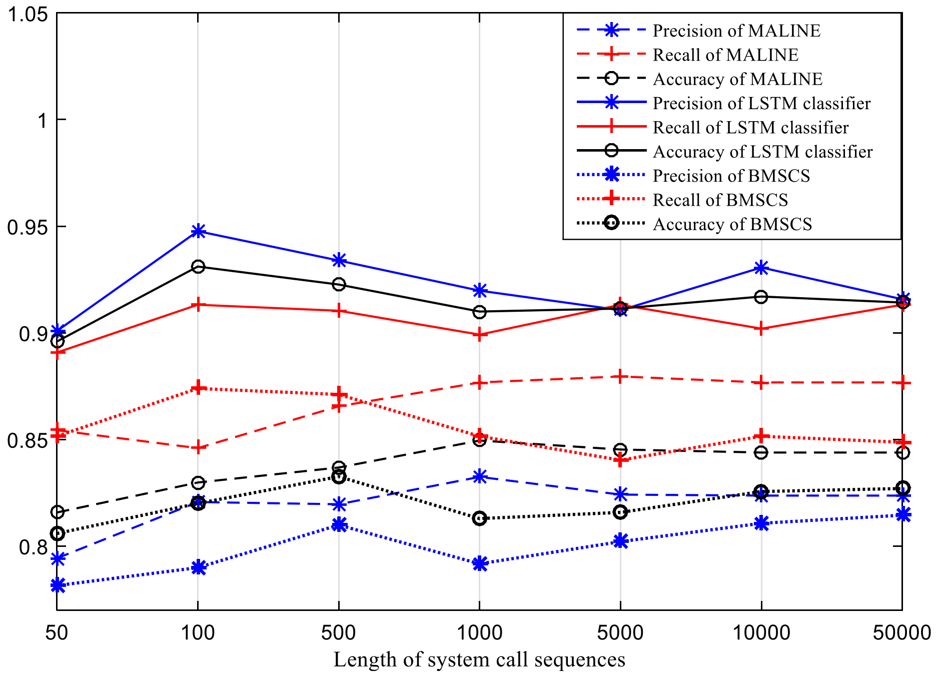


Fig. 9 Comparison between our LSTM classifier, MALINE and BMSCS

achieves the highest recall of 87.39% with the FPR of 23.45%. FPRs of MALINE and BMSCS are much higher than that of our detection method. The performance of BMSCS is close to that of MALINE. It can be obviously seen from Fig. 9 that precision, recall and accuracy of our classifier are higher than those of MALINE and BMSCS regardless of the length of the system call sequence. From Table 4, FPR of our classifier is lower than that of MALINE. So it can be drawn a conclusion from Table 4 and Fig. 9 that our LSTM model is better than that of the two *n*-gram models, MALINE [10] and BMSCS [30].

6 Conclusion and future work

In this paper, based on the LSTM networks we build a new classifier to detect malware on Android with system call sequences. In our classifier, there are two LSTM networks, which are trained by malware and trusted applications respectively. When a new sequence comes, we calculate two similarity scores from the two networks to classify the corresponding application. Comprehensive experiments have been done to test the performance of our new classifier. The results show that our method can achieve the highest recall of 96.6% and the highest precision of 91.3% with the accuracy of 93.7%. Furthermore, the comparison with the *n*-gram models, MALINE [10] and BMSCS [30], is made, which indicates our classifier is superior to the detection method based on the *n*-gram model.

Limitation of the work is that we have not preprocessed the system call sequences, and the sequences may include some noises. In the future work, we can try other sequence analysis techniques. We only consider the dynamic features, and do not use the static features such as opcode sequences. The dynamic features and static features can be combined to improve the accuracy. Besides system call sequences, there are other features, such as the smali code or the source code. Deep learning methods combined with more features can be used to detect malware in the future work.

Acknowledgements This work is supported by the NSFC projects (61375054, 61402255, 61202358), the National High-tech R&D Program of China (2015AA016102), Guangdong Natural Science Foundation (2015A030310492, 2014A030313745) and the RD Program of Shenzhen (JCYJ20150630170146831, JCYJ20160301152145171, JCYJ20160531174259309, JSGG20150512162853495, Shenfagai [2015] 986), and Cross fund of Graduate School at Shenzhen, Tsinghua University (JC20140001).

References

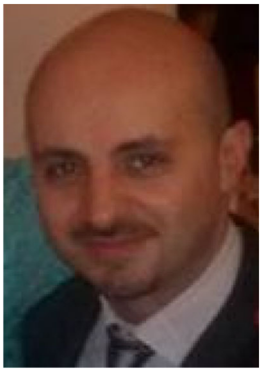
1. Arp D, Spreitzenbarth M, Hubner M, et al (2014) DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket, in: Proceeding of 21th Annual Network and Distributed System Security Symposium (NDSS), San Diego, 2014
2. Aung Z, Zaw W (2013) Permission-based android malware detection. Int J Sci Technol Res 2:228–234
3. Battista P, Mercaldo F, Nardone V, et al (2016) Identification of Android Malware Families with Model Checking, in: Proceeding of International Conference on Information Systems Security and Privacy, Rome, 2016
4. Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult, IEEE Press, Neural Networks, 5(2) (1994), pp. 157–166
5. Bengio Y, Schwenk H, Senécal J et al (2003) Probabilistic language models. J Mach Learn Res 3:1137–1155

6. Canfora G, Medvet E, Mercaldo F, et al (2015) Detecting Android malware using sequences of system calls, in: Proceeding of International Workshop on Software Development Lifecycle for Mobile (DeMobile), 2015, pp 13–20
7. Canfora G, Mercaldo F, Visaggio CA (2016) An HMM and structural entropy based detector for android malware: an empirical study. Comput Secur 61:1–18
8. Chen PS, Lin SC, Sun CH (2015) Simple and effective method for detecting abnormal internet behaviors of mobile devices. Inf Sci 321:193–204
9. Chen S, Xue M, Tang Z, et al (2016) StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware, in: Proceeding of ACM on Asia Conference on Computer and Communications Security(ASIACCS), Xian, 2016, pp 377–388
10. Dimja, Marko, Atzeni S, et al (2016) Evaluation of Android Malware Detection Based on System Calls. In: Proceedings of ACM on International Workshop on Security and Privacy Analytics (IWSPA), New Orlean, pp 1–8
11. Elman JL (1990) Finding structure in time 1990. Cogn Sci 14:179–211
12. Feng Y, Anand S, Dillig I, et al (2014) Apposcopy: semantics-based detection of Android malware through static analysis, in: Proceeding of 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE14), Hong Kong, 2014, pp 576–587
13. FireEye, Out of Pocket (2015): A Comprehensive Mobile Threat Assessment of 7 Million iOS and Android Apps, < https://www.fireeye.com/rs/fireeye/images/rpt-mobilethreat assessment.pdf>, (accessed 17.08.27)
14. Graves A (2012), Supervised Sequence Labelling with Recurrent Neural Networks. Studies in Computational Intelligence
15. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9:1735–1780
16. Li Q, Li X (2015) Android Malware Detection Based on Static Analysis of Characteristic Tree, in: Proceeding of International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, Xian, 2015, pp 84–91
17. Li Y, Shen T, Sun X, et al (2015) Detection, Classification and Characterization of Android Malware Using API Data Dependency, in: Proceeding of International Conference on Security and Privacy in Communication Systems(SecureComm2015), Dallas, 2015, pp 23–40
18. Lunden I. (2015) 6.1b smartphone users globally by 2020, overtaking basic fixed phone subscriptions.< http://techcrunch.com/2015/06/02/6-1b-smartphone-users-globally-by-2020-overtaking-basic-fixed-phone-subscriptions >, (accessed 17.08.27)
19. android-market-api-py. < https://github.com/liato/android-market-api-py> (accessed 17.08.27)
20. Mercaldo F, Nardone V, Santone A, et al (2016) Download Malware? No, Thanks. How Formal Methods Can Block Update Attacks, in: Proceeding of Fme Workshop on Formal Methods in Software Engineering, Austin, 2016, pp 22–28
21. Mercaldo F, Nardone V, Santone A, et al (2016) Ransomware Steals Your Phone. Formal Methods Rescue It, in: Proceeding of International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Heraklion, Crete, Greece, 2016, pp 212–221
22. Mikolov T, Karafiat M, Burget L, et al (2010) Recurrent neural network based language model, in: Proceeding of the Annual Conference of the International Speech Communication Association (Interspeech 2010), Makuhari, 2010, pp 1045–1048
23. Rashidi B, Fung C, Bertino E (2016) Android resource usage risk assessment using hidden Markov model and online learning. Comput Secur 65:90–107
24. Saracino A, Sgandurra D, Dini G, et al (2016) MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention. IEEE Transactions on Dependable & Secure Computing, 2016, pp 1–1
25. Sundermeyer M, Schluter R, Ney H (2012) LSTM Neural Networks for Language Modeling, in: Proceeding of the Annual Conference of the International Speech Communication Association (Interspeech2012), Portland, 2012, pp 601–608
26. System call https://en.wikipedia.org/wiki/System_call > (accessed 17.08.27)
27. Tensorflow http://www.tensorflow.org > (accessed 17.08.27)
28. Wang Z, Li C, Guan Y, et al (2015) DroidChain: A novel malware detection method for Android based on behavior chain, in: Proceeding of Communications and Network Security (CNS), FLORENCE, 2015, pp 727–728
29. Wu W C, Hung S H (2014) DroidDolphin: a dynamic Android malware detection framework using big data and machine learning, in: Proceeding of 2014 Conference on Research in Adaptive and Convergent Systems, Towson, 2014, pp 247–252
30. Xiao X, Wang Z, Li Q et al (2017) Back-propagation neural network on Markov chains from system call sequences: a new approach for detecting android malware with system call sequences. IET Inf Secur 11:8–15

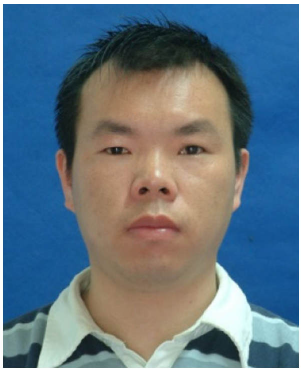
31. Xu K, Li Y, Deng RH (2016) ICCDetector: ICC-based malware detection on android. IEEE Trans Inf Forensics Secur 11:1252–1264
32. Yeh C W, Yeh W T, Hung S H, et al (2016) Flattened Data in Convolutional Neural Networks: Using Malware Detection as Case Study, in: Proceeding of International Conference on Research in Adaptive and Convergent Systems, Odense, 2016, pp 130–135
33. Yu W, Ge L, Xu G, et al (2014) Towards Neural Network Based Malware Detection on Android Mobile Devices, Cybersecurity Systems for Human Cognition Augmentation, 2014, pp 99–117
34. Zhou Y, Jiang X (2012) Dissecting Android Malware: Characterization and Evolution, in: Proceedings of 33rd IEEE Symposium on Security and Privacy, SAN FRANCISCO, 2012, pp. 95–109
35. Zhou Y, Jiang X (2013) Android malware, Springer, New York, USA, 2013

Xi Xiao is an associate professor in Graduate School At Shenzhen, Tsinghua University. He got his Ph.D. degree in 2011 in State Key Laboratory of Information Security, Graduate University of Chinese Academy of Sciences. His research interests focus on information security and the computer network.

Shaofeng Zhang is pursuing his Master degree in Computer and Science at Tsinghua University. His research interests focus on information security and the computer network. His recent research focuses on Android malware detection with the deep learning.



Francesco Mercaldo obtained his PhD in 2015 with a dissertation on malware analysis using machine learning techniques. The core of his research is finding methods and methodologies to detect the new threats applying the empirical methods of software engineering as well as studying the current mechanisms to ensure security and private data in order to highlight the vulnerability. Currently he works as post-doctoral researcher at the Institute for Informatics and Telematics, National Research Council (CNR), Pisa, Italy



Guangwu Hu is a lecturer of School of Computer Science, Shenzhen Institute of Information Technology, 518,055, Shenzhen, China. He received the Ph.D. degree in Computer Science and Technology from Tsinghua University in 2014, then he became a postdoctor in the Graduate School at Shenzhen, Tsinghua University. His research interests include software defined networking, next-generation Internet and Internet security.

Arun Kumar Sangaiah had received his Doctor of Philosophy (PhD) degree in Computer Science and Engineering from the VIT University, Vellore, India. He is presently working as an Associate Professor in School of Computer Science and Engineering, VIT University, India. His area of interest includes software engineering, computational intelligence, wireless networks, bio-informatics, and embedded systems.