Chapter 7
Towards Neural Network Based Malware Detection on Android Mobile Devices

Wei Yu, Linqiang Ge, Guobin Xu, and Xinwen Fu

7.1 Introduction

The rapid growth of smart mobile devices has led to a renaissance for mobile services. These devices can augment cognitive abilities with multi-function applications related to web, education, travel, game, financial, and many others. For example, face recognition applications can help identify or verify a person to enhance human cognitive abilities. The Android platform is an open source operating system for smart mobiles and provides services, including security configuration, process management, and others [1]. With 48 % of smartphone subscribers using Android mobiles, Android leads the smartphone market in the U.S. [2].

Nonetheless, the popularity of Android mobile devices has led to enormous security challenges. Malware, as a malicious application that can be installed on mobile devices, can gain access to these devices and collect user sensitive information. Malware has proven to be a serious problem for the Android platform because malicious applications can be distributed to mobile devices through an application market. From the defender's perspective, how to effectively detect malware and enhance the cognitive performance of users and system administrators becomes a challenging issue. Traditional static analysis techniques heavily rely on capturing malicious characteristics and bad code segments embedded in software. This makes it infeasible to deal with a large population of unknown malware. Hence, it is critical to develop a machine learning-based system that can dynamically learn the

W. Yu (✉) • L. Ge • G. Xu
Department of Computer & Information Sciences, Towson University, Towson, MD, USA
e-mail: wyu@towson.edu; lge2@students.towson.edu; gxu2@students.towson.edu

X. Fu
Department of Computer Science, University of Massachusetts, Lowell, MA, USA
e-mail: xinwenfu@cs.uml.edu

© Springer International Publishing Switzerland 2014

R.E. Pino et al. (eds.), *Cybersecurity Systems for Human Cognition Augmentation*, Advances in Information Security 61,
DOI 10.1007/978-3-319-10374-7_7

IPR2023-01160
KASPERSKY EX1030 PAGE 1

behavior of malware and augment the human cognition process of defending against malware attacks in the battle of mobile security.

In this chapter, we propose an Artificial Neural Network (ANN)-based malware detection system that uses both permissions and system calls to detect unknown malware. In our system, we consider two types of ANNs: Feedforward Neural Networks (FNN) to learn the patterns of permissions and Recurrent Neural Networks (RNN) to understand the structure of system calls. Permission requests are collected from applications to distinguish between benign applications and malware. We also collected system calls associated with application execution to capture the runtime behaviors of benign applications and malware. Through the training process, the ANN can learn the anomaly behaviors of malware in terms of permission requests and system calls. The resulting model can be further used to detect unknown malware. To evaluate the effectiveness of our malware detection system, we used real-world malware and benign applications to conduct experiments on Android mobile devices. The resulting data shows that our system can effectively detect malware.

The remainder of the chapter is organized as follows: We introduce the ANN in Sect. 7.2 and two types of data sources for malware detection: permission and system calls in Sect. 7.3. In Sect. 7.4, we present our ANN-based malware detection system. Experimental results were demonstrated to validate the effectiveness of our proposed detection system in Sect. 7.5. We then discuss the issues related to our work in Sect. 7.6. We review related work in Sect. 7.7 and conclude the chapter in Sect. 7.8.

7.2 Artificial Neural Networks

We consider ANN to conduct malware detection. Generally speaking, a neural network refers to a network or circuit that mimics the structure and behavior of biological neurons [3]. The parameters of a neural network are set through a training process that uses known data sets as inputs. After that, the trained neural network can be used as a classifier to conduct detection.

7.2.1 Feedforward Neural Networks (FNN)

FNNs are a well-known and widely used type of neural network [4–8]. An FNN consists of a certain number of layers and a number of units called artificial neurons or nodes that are organized in layers. In a typical setting, an FNN has an input layer, an output layer, and one or more hidden layers between the input and the output layer. In an FNN, all data and computation flows are in one direction: from input to output data. Except for input units, each unit in a layer is connected to all the units in the previous layer and receives inputs directly from the nodes in the previous layer. Each connection may have a different strength or weight. During the training process, the

IPR2023-01160
KASPERSKY EX1030 PAGE 2

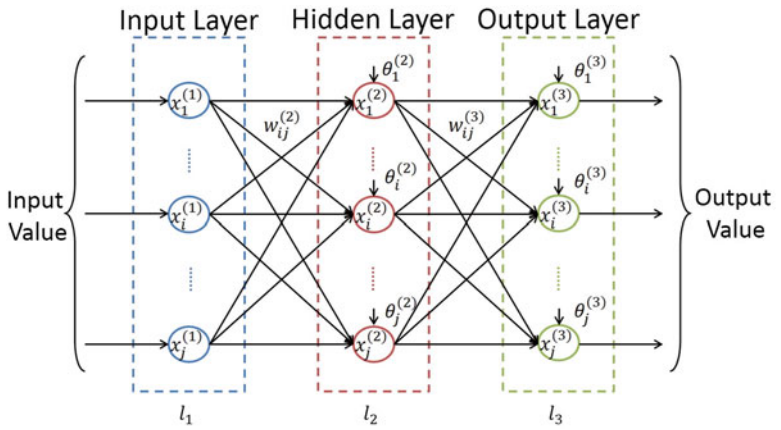

Fig. 7.1 A typical structure of FNN

weight can be adjusted through learning algorithms such as BackPropagation (BP). The typical structure of an FNN is illustrated in Fig. 7.1.

Here, l represents the layer of the FNN, where $l=1$ is for the input layer, $l=2$ is for the hidden layer, and $l=3$ is for the output layer. In principle, the output values are compared with the correct answer to compute the value of a predefined error-function that is then sent back through the network. With the backward propagation errors between real and estimated values from the output layer to the hidden layer and from the hidden layer to the input layer, errors in each layer can be estimated and the assigned weights $\omega_{ij}^{(l)}$ can be updated correspondingly. After repeating this procedure many times, the neural network eventually reaches a state where the computed error is small. At this moment, the training process is complete.

7.2.2 Recurrent Neural Networks (RNN)

Unlike the FNN, the fundamental feature of an RNN is that the network contains at least one feedback connection. This makes an RNN useful for handling temporal classification problems or learning sequences. Similar to an FNN, an RNN consists of a number of units and multiple layers: input layer, output layer, and one or more hidden layers. When the data is fed to an RNN, a state activation is generated in the hidden layers. In the next time slot, the previous state activation is fed back to the hidden layer, combining with new input data. During the training process, the weight of unit connections and feedback connections can be adjusted through learning algorithms such as Back Propagation Through Time (BPTT). The BP algorithm used in an FNN cannot be directly applied to an RNN because of the inherent cycles present. Hence, BPTT unfolds the network over time, eliminating cycles and allowing the neural network to be trained as if it consists of several connected FNNs where the BP algorithm can be used.

7.3 Permissions and System Calls

In this section, we first review the typical malware detection techniques. Then we examine in detail how permissions and system calls can be used as the fundamental detection data source.

7.3.1 Overview

There are several types of detection techniques. Static analysis [9] has been used to carry out malware detection through the process of decompiling executable software, generating source code, and then using code analysis tools to inspect the recovered source code. Static analysis is limited by the capability of code analyzers and can only deal with applications that involve a small number of permissions and system calls.

Permission and dynamic analysis schemes are promising techniques to defend against a large class of unknown malware. To be specific, permission-based detection sets security policy rules. When an application is installed, the permission-based detection extracts security configurations and checks them against security policy rules [10]. Conversely, dynamic analysis-based detection [11] executes the mobile application and monitors the applications dynamic behavior. Based on the runtime behavior, the malware can be detected. As malicious behavior is always difficult to hide and can be used as a feature to identify malware, we can use ANN techniques to accurately characterize the behavior of applications.

7.3.2 Permissions

Android provides third-party applications that have the capability of accessing resources such as phone hardware, settings, user data, and others through permissions. For example, the INTERNET permission allows applications to open network connections. Each application must declare in advance what permissions it requires, and users are notified during the installation about the permissions that it will obtain. Users can cancel the installation process if they do not want to grant a permission to the application, but they might not have the knowledge to determine which permissions should be requested by and granted by a particular application. Usually, different types of applications request reasonable permissions. Nonetheless, even an application requesting a reasonable permission might conduct malicious behavior. For example, a social network application that requests to only access the contact may additionally copy contacts personal information to a remote server.

To show the potential of using permissions to detect malware, we investigated the distribution of permissions requested by electronic books - one class of

IPR2023-01160
KASPERSKY EX1030 PAGE 4

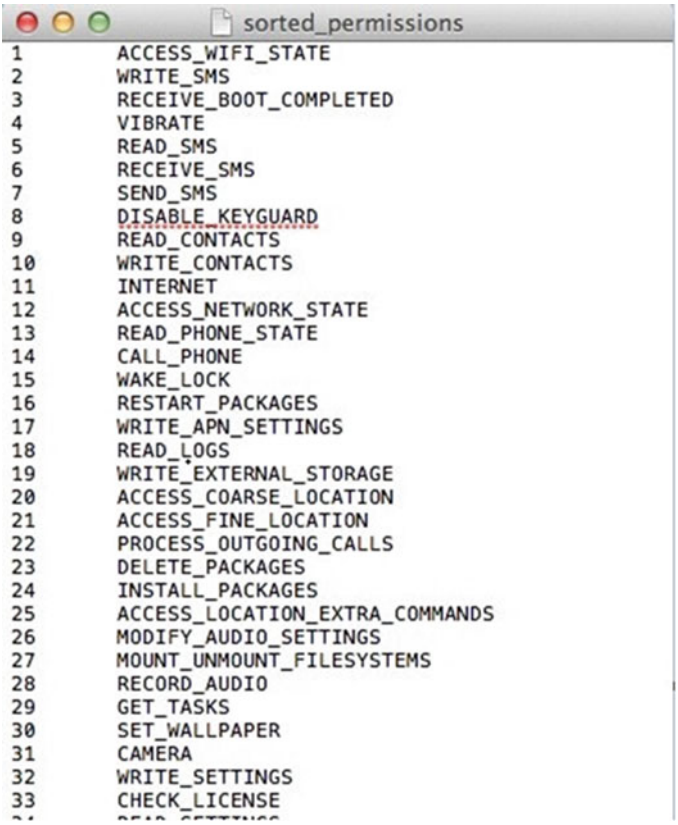

Fig. 7.2 An example of mapped permissions

applications. We installed 96 benign applications from Google Play and used 92 digital book malware samples from the Android Malware Genome Project (http://www.malgenomeproject.org/). For each Android application, we extracted permissions from the corresponding application package (APK) file. The details of the retrieving process will be presented in Sect. 7.4.1. We define each captured permission as one feature and map it to an integer. Figure 7.2 shows an example of mapped permissions.

After retrieving the permissions from all applications, the distribution of permissions can be computed. One such example is shown in Fig. 7.3. As we can see, most malware samples heavily request permissions 1–20, which are WRITE SMS, SEND SMS, READ CONTACT, etc. We can conclude that electronic book applications that request permissions 1–20 are probably malware. Hence, the permissions requested by an application can be used to recognize whether the application contain malware.

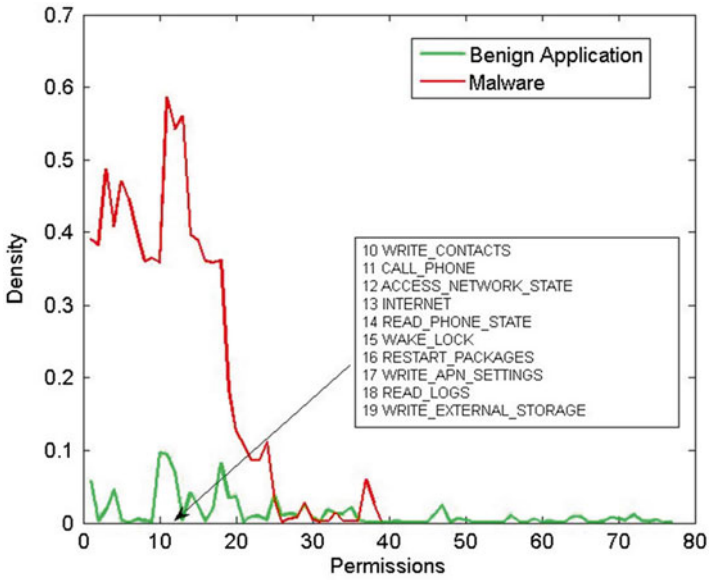

Fig. 7.3 Distribution of permissions

7.3.3 System Calls

A system call is the mechanism used by applications to request a service from the operating system kernel. System calls provide the interface between the process and operating systems. The operating system provides services, including the creation and execution of new processes and access control of resources. The sequence of system calls occurs consecutively over time and can capture actions performed by applications during execution. As system calls provide an essential interface between the application and operating system, we shall examine system calls to capture the runtime behavior of the interactions between applications and the operating system.

7.4 An ANN-Based Malware Detection System

We now present the workflow of our proposed ANN-based malware detection system as shown in Fig. 7.4. We would like to emphasize that the workflow is general and can be used for both permission-based detection and system call- based detection. In the offline training phase, we first collected real-world benign and malicious applications. Next, we executed the collected applications and dumped the data sources. In order for machine learning algorithms to learn the feature patterns of

IPR2023-01160
KASPERSKY EX1030 PAGE 6

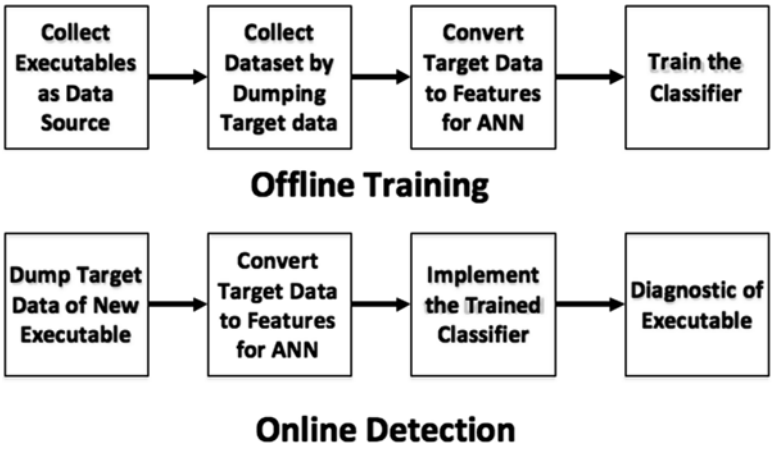

Fig. 7.4 Workflow

malware and benign applications, all data sources needed to be parsed and mapped to the format required by the FNN and RNN algorithms described in Sect. 7.2. Using the mapped data as input, we then trained the neural network. In the online detection phase, we dumped the data sources from new applications and the trained neural network would be used to determine whether the new application is malware or benign. As permissions and system calls contain different features and have different formats, we first introduce permission-based detection and then system call-based detection in the following subsections.

7.4.1 Permission-Based Detection

Offline Training We now discuss the steps used for the offline training process.

Step 1: **Data source collection and classification**. The first step in the offline training phase is to collect the data source from the executing applications. With real-world benign applications and malware samples, we consider that applications in the same category should exhibit similar activities and we use such activities to learn the anomaly profile. Based on these learned profiles, we can categorize applications as benign or malicious.

Step 2: **Dumping Permissions of Data source**. Using the benign application and malware samples, we dump the permissions requested by each application. In the Android system, all permissions are included in the *Android-Manifest.xml* file. After collecting application *apk* files, we use a known reverse engineering tool *Android Asset Packaging Tool* (*aapt*) to reconstruct the source code and obtain the AndroidManifest.xml file for each application. An example is shown below:

```
Linqiangs-MacBook-Pro:tools linqiangge$
Linqiangs-MacBook-Pro:tools linqiangge$ ./aapt dump permissions QQ_for_Pad_v_1.9.3.a
pk
package: com.tencent.android.pad
uses-permission: android.permission.READ_PHONE_STATE
uses-permission: android.permission.CAMERA
uses-permission: android.permission.WRITE_EXTERNAL_STORAGE
uses-permission: android.permission.MOUNT_UNMOUNT_FILESYSTEMS
uses-permission: android.permission.VIBRATE
uses-permission: android.permission.INTERNET
uses-permission: android.permission.WAKE_LOCK
uses-permission: android.permission.ACCESS_NETWORK_STATE
uses-permission: android.permission.SYSTEM_ALERT_WINDOW
uses-permission: com.android.launcher.permission.INSTALL_SHORTCUT
uses-permission: android.permission.MODIFY_AUDIO_SETTINGS
uses-permission: android.permission.RECORD_AUDIO
uses-permission: com.tencent.android.pad.permission.IM_SERVICE
uses-permission: com.tencent.android.pad.permission.EXT_IM_SERVICE
permission: com.tencent.android.pad.permission.IM_SERVICE
permission: com.tencent.android.pad.permission.EXT_IM_SERVICE
permission: com.tencent.android.pad.permission.WRITE_SETTINGS
permission: com.tencent.android.pad.permission.READ_SETTINGS
Linqiangs-MacBook-Pro:tools linqiangge$
```

Fig. 7.5 Dumping permissions

```
  <manifest xmlns:
  android="http://schemas.android.com/apk/res/android"
package="com.android.app.QQ_for_Pad_v_1.9.3" >
  A: android:versionCode(0x0101021b)=(type 0x10)0x7
  A: android:versionName(0x0101021c)="2.1-update1"
  A: package="com.android.spare_parts"
  <uses-permission android:name="android.permission.READ_PHONE_
STATE"/> <uses-permission android:name="android.permission.CAMERA"/>
  … </manifest>
```

We then use the command *aapt* dump permission to collect all permissions requested by each application. Figure 7.5 shows an example of the dumping process and the corresponding result.

Step 3: **Feature extraction**. Next, we collect a set of files where each file consists of permissions requested by one application. For training, we process the data and map them to the format required by the ANN. To this end, we developed a mapping algorithm to convert the original permissions into usable input. As described previously, we use Algorithm 7.1 to define each permission as one feature and assign an integer to each feature.

Using the example shown in Fig. 7.5, we now explain Algorithm 7.1. In this algorithm, we care about the feature (i.e., permission name) and the feature value, defined as whether it was requested by the application. Note that one permission can be requested only once by an application. If a particular permission is requested, its feature value is 1; otherwise its feature value is 0. After the first *for loop* of Algorithm 7.1, we obtain the output shown in Fig. 7.6.

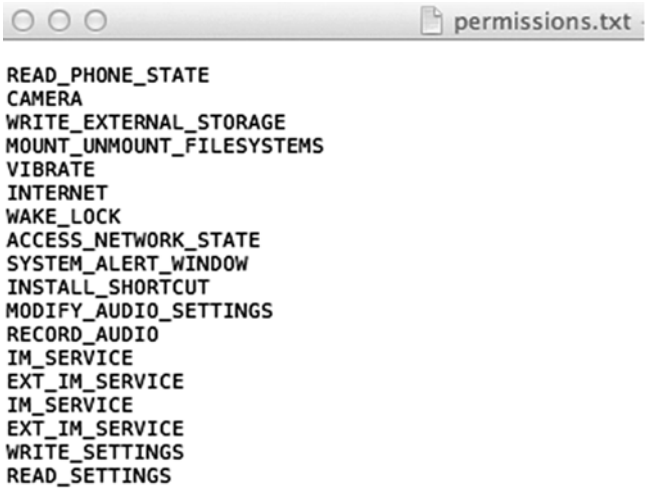

Fig. 7.6 An example of permissions

Input: Original Android Permissions stored in raw data
Output: Permission Feature Vector **A**
n = gram number; **A**=[]; t=Total number of permissions
foreach *file in raw data folder* do
  foreach *line in file* do
    remove all information except the permissions name;
  end
  store file in permissions name folder;
end
foreach *file in permissions name folder* do
  foreach *line in file* do
    map permission name to integers i as feature name;
  end
  store file in mapped-integer data folder;
end
foreach *m=1:t* do
  A[m]= 0;
end
foreach *i in mapped-integer data folder* do
  A[i]=1;
end
if *n>1* ***then***
  map to n-gram format;
end

Algorithm 7.1 Permission Mapping Algorithm

Because the ANN only accepts integers as input, we map each permission name to an integer number after processing the name sequence of permissions. After the second *for loop* in Algorithm 7.1, the mapping produces output similar to "*01*,*02*,*03*,*06*,*09*,*15*,*20*". As examples, INTERNET is mapped to 11, READ PHONE STATE is mapped to 13, and SEND SMS is mapped to 7. We can extend this idea to use 2-gram as a detection feature by applying two contiguous permissions instead of one. As an example, we combine every two contiguous integers and the mapping produces output similar to "*0102*,*0203*,*0304*,*0405*" where "*0102*" represents the permissions ACCESS WIFI STATE and WRITE SMS requested sequentially.

After we have the input to the ANN mapped as an integer sequence, the next step is to obtain the value for the each feature. Recall that we use the appearance of a permission as the feature value. For each feature that appears, its value is assigned as 1. For the features that do not appear, we assign their values as 0. After the last two *for loops* in Algorithm 7.1, we obtain a feature vector for the input of ANN as follows:

```
  1,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,1,0, 0,0,0,0,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,1,0
```

Step 4: **Classifier learning**. In this step, we use the learning module established in the neural network to learn the application behavior from training data. We input the feature vectors to the Matlab Neural Network Toolbox built-in Matlab R2013a (8.1.0.604) to implement permission-based detection. We set the number of nodes in the hidden layer to 10 and then 20.

Online Detection The workflow of the online detection phase is similar to the one described in the offline training phase. Similarly, to classify an application, the first step is to dump permissions and map the permission sequence to the format required by the ANN. We can then use the trained ANN to determine whether a new application is either malware or benign. We use the established ANN and test data as input from new applications. The test file has the same format as the training file, which consists of the feature vector associated with each application. The online detection process outputs the result file which contains the classification result. In our implementation, the result is either +1 or −1. Here, when the number is positive, the ANN classifies it as a benign application; when the number is negative, the ANN classifies it as malware.

7.4.2 System Call-Based Detection

The workflow of the detection system based on system calls is similar to the detection system based on permissions. The major difference is to use a different data source. In the following, we briefly introduce the workflow of system call- based detection.

Offline Training As before, we now discuss the steps for offline training.

Step 1: **Data set collection and classification**. The first step is to collect the data set. After we collect real-world benign applications and malware samples, we categorize them into different groups.

Step 2: **System calls recording**. We record the system calls used by our benign applications and malware samples by applying a known tool *Strace*. In order to install *Strace*, we use the *Nexus Root Tookit v1.6.2* to obtain root permission on Android devices. Next, we run *Strace* and capture the system calls used by the benign applications and malware. To install malware on an Android device from a remote computer, we use the *Android Debug Bridge* (*ADB*).

Step 3: **Feature extraction**. We then record a set of files where each file contains the system calls generated by each executed application. To use the ANN, we need to process the data and map them to the required format described previously. Using Algorithm 7.1, we map each system call named to an integer. As an example, *clock-gettime* is mapped to 1, *recvfrom* is mapped to 5, and *ioctl* is mapped to 7. Again, we can extend this idea to use 2-gram as a detection feature by applying two contiguous system calls as a detection feature instead of one. To construct the mapping for 2-gram, we combine each pair of contiguous integers and generate output similar to "*0101 0101 0105 0507 0701 0117 1717 1717 1717 1706*" where "*0105*" represents system calls *clock-gettime* and *recvfrom* executed sequentially. We then capture the density of system calls by computing the ratio of the number of instances of each system call to the total number of system calls generated by the application. We can then express a feature and its value as feature: value such as "*1*:*0.2283 2*:*0.0369 3*:*0.0387 4*:*0.0267 5*:*0.0848*" where feature 1 has a density of 0.2283, feature 2 has a density of 0.0369, etc.

Step 4: **Classifier learning**. This step is the same as Step 4 for permission-based detection. Afterwards, we have completed the training process of the ANN and are ready to use it to conduct online detection.

Online Detection The workflow of the online detection phase is similar to the one in the offline training phase. Similarly, to classify an application, we execute it, dump the system calls, and map the sequence of system calls to the format required by the ANN. Using the ANN established through the offline training phase, we can determine whether a new application is malware or benign.

7.5 Performance Evaluations

Using real-world malware and benign applications collected on the Android platform, we show the effectiveness of our developed detection system. We installed 96 benign software applications from Google Play and evaluated 92 digital book malware samples from the *Android Malware Genome Project* (http: //www.malgenomeproject.org/).

IPR2023-01160
KASPERSKY EX1030 PAGE 11

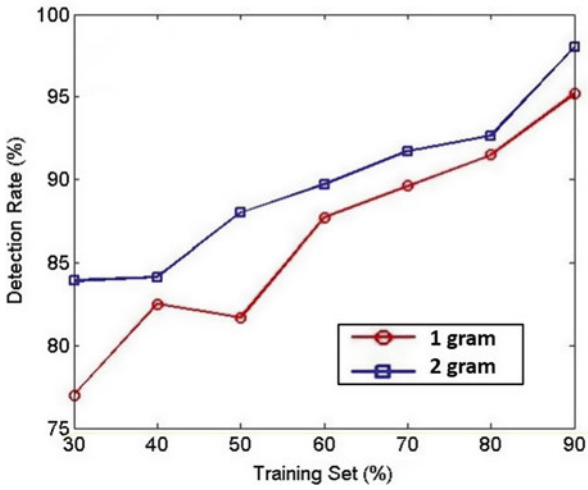

Fig. 7.7 Detection rate for permission based detection vs. Training set ratio (FNN with 10 nodes)

We installed and executed applications on the *Sumsang Galaxy Nexus* and *Google Nexus 7* smartphones in our experiments. First, we collected and transmitted each application's permission requests and system calls to a remote computer which conducted both the offline and online detection processes described in Sect. 7.4. A *Samsung Notebook NP700G* equipped with Intel Core i7 2.40GHZ processor, 16GB RAM, and 320GB hard drive served as our detection computer. Again, we used the Matlab Neural Network Toolbox built-in Matlab R2013a (8.1.0.604) that contains both of the FNN and RNN implementations used in our experiments. The number of hidden nodes in the FNN and the RNN are set to 10 and then 20.

With a larger training set, more information can be used to train the ANN classifier, leading to higher detection accuracy. To validate this hypothesis, we let $p \in [0, 1]$ which define the training set ratio as the ratio of the number of training samples to the total number of samples. If n is the number of total applications then np is the number of applications used for training and $n(1-p)$ is the number of applications used to validate the accuracy of the trained ANN. To measure the effectiveness of our detection system, we define the detection rate as the probability of correctly classifying the malware. That is, the ratio of the number of malware correctly detected to the total number of malware samples. We also define the error rate as the probability of falsely classifying applications. That is, the ratio of the number of applications falsely classified to the total number of applications.

Permission-Based Detection: Figure 7.7 illustrates the relationship between the detection rate and the training set ratio in terms of the length of grams when an FNN with ten hidden nodes is used. As we can see, in general, the detection rate rises as the training set ratio increases. The permission-based detection with 2-gram data as input can achieve a better detection rate than the permission-based detection with 1-gram data as input. For example, when the training set ratio is 60 %, the detection rate reaches almost 90 % when 2-gram are used while the detection rate is 85 %

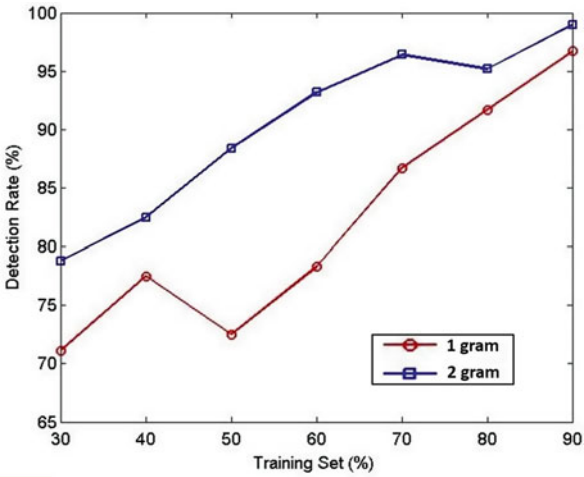

Fig. 7.8 Detection rate for permission based detection vs. Training set ratio (FNN with 20 nodes)

when 1-gram are used. As we expected, when using more training data, more knowledge of malware can be obtained, leading to increased detection accuracy.

Figure 7.8 shows the detection rate versus training set ratio when the number of hidden nodes of the FNN is set to 20. Similar to Fig. 7.7, as we increase the size of the training set, the detection rate increases. Like before, detection using 2-gram data as input achieves better performance than detection using 1-gram data as input. In the case of 2-gram data as input, when the training set ratio is higher than 50 %, the FNN with 20 hidden nodes performs better than the one with 10 hidden nodes. We also observed that, in the case of 1-gram data as input, the FNN with 10 hidden nodes performs better than the FNN with 20 hidden nodes. One reason may be caused by limited malware samples.

Figure 7.9 illustrates the result of an RNN with 10 hidden nodes. In comparison with Fig. 7.7, we can see that the FNN achieves better performance in both the 1-g and 2-gram cases than when using the RNN. Hence, we conclude that the FNN is more effective for permissions-based detection.

System Call-Based Detection: Figures 7.10, 7.11 and 7.12 illustrate the relationship between the detection rate and training set ratio in terms of the length of data grams when we take system calls as input. Similar to the permission-based detection shown in Figs. 7.7, 7.8 and 7.9, when more samples are used in the training process, a higher detection rate can be achieved. For example, when we use a training set of 90 %, both the FNN and the RNN achieved detection rates of more than 93 %. When the hidden nodes are set to 10, the RNN obtains better detection accuracy than the FNN for both permission-based and system call-based detection.

We also study the accuracy of our detection system using another metric: error rate. We expect that with a larger training set, our detection will produce a lower error rate. Figures 7.13 and 7.14 illustrate the relationship between error rate and the training set ratios when we take permissions and system calls as inputs to an FNN and an RNN. In our evaluation, we selected two scenarios to validate that our detection system obtains low error rates; other scenarios are essentially similar.

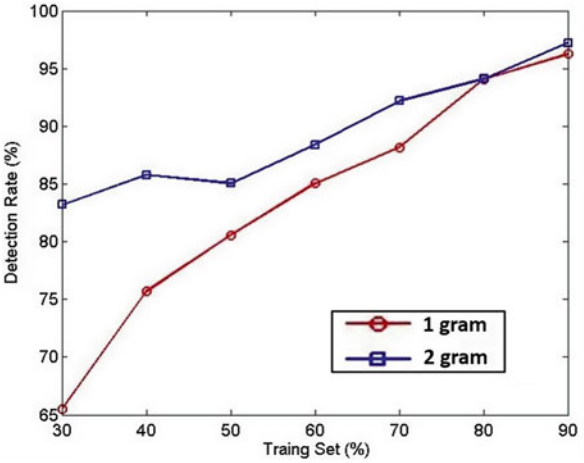

Fig. 7.9 Detection rate for permission based detection vs. Training set ratio (RNN with 10 nodes)

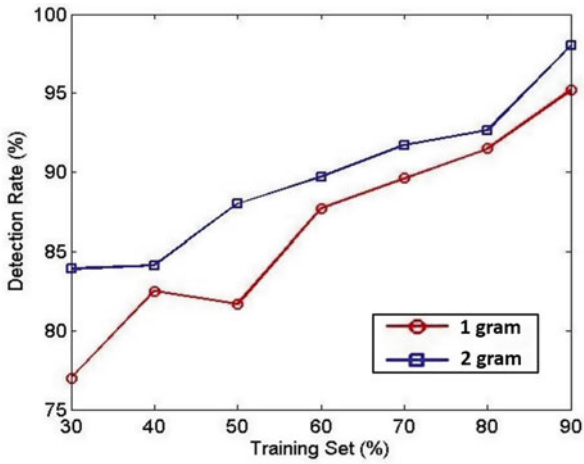

Fig. 7.10 Detection rate for system call based detection vs. Training set ratio (FNN with 10 nodes)

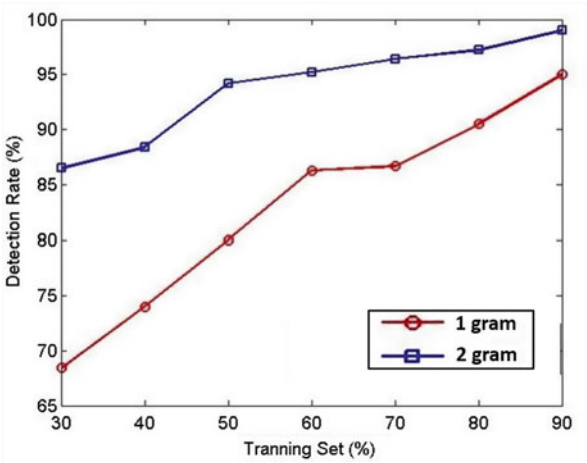

Fig. 7.11 Detection rate for system call based detection vs. Training set ratio (FNN with 20 nodes)

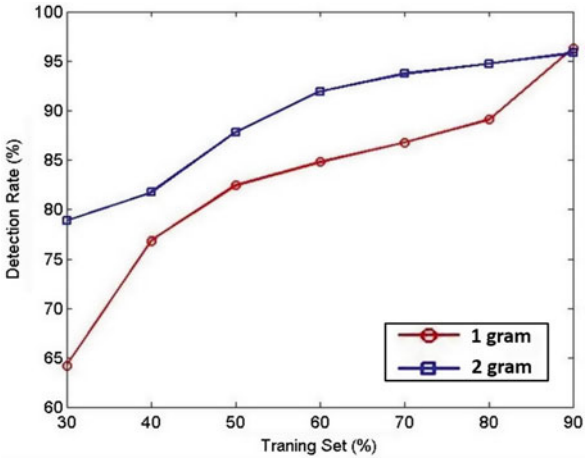

Fig. 7.12 Detection rate for system call based detection vs. Training set ratio (RNN with 10 nodes)

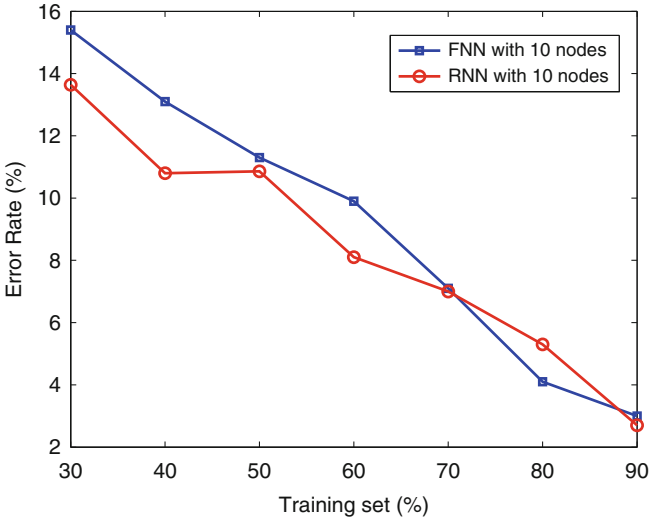

Fig. 7.13 Error rate for permission based detection vs. Training set ratio (1-g)

We used 1-gram for data input and set the hidden layer of the FNN and RNN to contain ten nodes. We have several observations from Figs. 7.13 and 7.14. First, for both permissions-based and system call-based detection, the error rates of both the FNN and RNN decrease as the training set ratio increases. This can be explained by observing that as we use more data in the training process, the FNN and RNN have a better chance to learn input data. This leads to the generation of a more accurate network for classification and a lower error rate. Second, the error rates are low for both the FNN and RNN in our detection system. For example, using a training set of 60 % with permissions-based detection, the error rate is 10 % using the FNN and

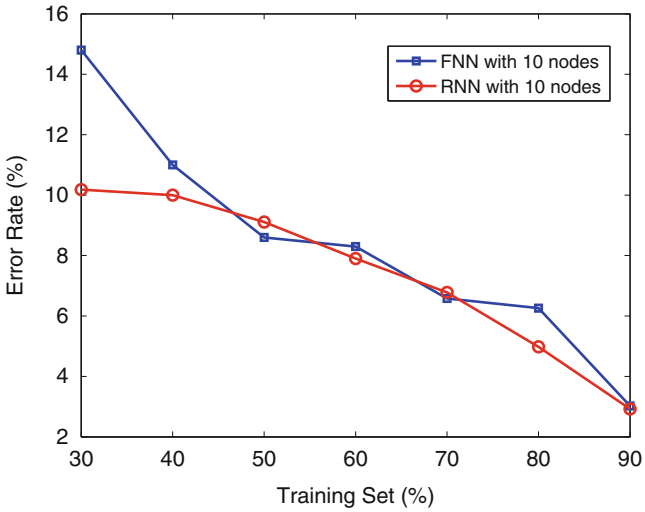

Fig. 7.14 Error rate for system call based detection vs. Training set ratio (1 g)

8 % using the RNN. Similar results have been obtained using system call based detection. Thus, we have confirmed that our detection system obtains high detection rates as well as low error rates, ensuring detection accuracy.

7.6 Discussions

In this section, we discuss some issues related to our malware detection system.

7.6.1 Overhead of Training Process

The major overhead of our ANN-based detection system comes from the training process. It is worth noting that the training process consists of procedures for collecting data sources, mapping data sources, and training the neural network. After the network is well trained, the online detection procedure can be fast. Overhead for the training process can be presented by $T = np(T_d + T_m) + T_l$, where n is the number of total applications, p is the training set ratio, and T_d, T_m, T_l are the average overhead for: dumping permissions and system calls from one application, mapping process and training the neural network, respectively.

As an example, consider training using 1-g. In our experiment, we implemented the permission-based detection and measured the execution time of each step. With $p = 90$ % and $n = 188$, the average time consists of 0.000343 s to dump permissions,

0.00012 s to map permissions, and 0.41 s to train neural networks. Hence, the total overhead of the training process is 0.613 s. Similarly, we investigated the overhead of system call-based detection. We note that in order to dump system calls associated with the execution of applications, we need to manually execute applications on real-world mobile devices and the execution times can be random, depending on the application. In our experiments, the overhead of mapping process is 0.00026 s and the total time for the training process is 0.194 s. It is worth noting that the computation overhead linearly increases with the number of applications. To make our system scale, one possible solution is to take advantage of powerful hardware for neuromorphic approaches to conduct threat analysis and detection.

7.6.2 Cloud Based Detection

We have developed an ANN-based malware detection system that detects unknown malware on mobile devices. Nonetheless, because a large number of mobile devices can be deployed in the system, those devices will generate big data associated with malware detection over time. Hence, mobile devices are characterized by limited storage capacity, constraint battery life time, and limited computational resources.

To address this issue, we shall investigate how to use the cloud computing infrastructure and algorithms to assist malware detection. Leveraging a cloud computing based service to store monitoring threat detection can expand resource and storage capacity and enhance the efficiency of threat analysis. By leveraging the cloud computing infrastructure, a monitoring agent can be deployed in the mobile device to collect permissions and system calls associated with applications then transmit such data sources to the remote cloud server. We can integrate our ANN-based detection and other detections schemes in the cloud server to help the human administer to defend against malware attacks.

7.7 Related Work

The detection of malware on a mobile platform can be categorized into static analysis, dynamic analysis, and permission analysis. These techniques have been investigated in the past by [12–16]. For example, Bose et al. [12] proposed a malware behavioral detection scheme on mobile handsets. Shamili et al. [13] presented a distributed Support Vector Machine (SVM) scheme to conduct malware detection, along with a statistical classification model. Deepak et al. [14] proposed a signature-based malware detection scheme. Schmidt et al. [15] conducted the static analysis of malware on the Android platform. To measure the effectiveness of different schemes on malware detection, Shabtai et al. [16] evaluated several classification and anomaly detection schemes and feature selection methods for mitigating malware on mobile devices.

IPR2023-01160
KASPERSKY EX1030 PAGE 17

Through permission analysis, malware detection can be conducted through the analysis of extracted security configurations and policy rules [10, 17–20]. For example, Aung et al. [18] developed a machine learning-based detection on the Android platform by monitoring permission related features and events. Huang et al. [20] conducted the permission-based detection for Android malware by using machine learning schemes such as AdaBoost, Naive Bayes, Decision Tree (C4.5), and Support Vector Machine. David et al. [19] presented a Self- Organizing Map (SOM) scheme to identify the permission-based security model using 1,100 android applications.

Neural networks can be used to learn and classify anomaly activities based on limited data sources [21]. There have been a number of research efforts on using neural networks to conduct threat detection [21–24]. For example, Mukkamala et al. [22] investigated schemes to conduct intrusion detection using neural networks and SVMs. Linda et al. [23] proposed a neural network-based approach to conduct intrusion detection for critical infrastructures. Golovko et al. [24] discussed the use of neural networks and artificial immune systems for carrying out malware and intrusion detection.

Different from existing research efforts, our detection system considers both permission and system calls as data sources. To learn the behavior of malware and benign applications, our system compares the performance of two classical neural networks: FNN and RNN. We have also shown that our implementation can detect unknown malware.

7.8 Conclusion

Malware attacks on smart mobile devices have been growing and posing security risks to mobile users. In this chapter, we developed an ANN-based malware detection system to automatically learn the behavior of applications and to detect unknown malware. In our developed system, we systematically compared the permission requests from application requests and system calls to capture the behavior of applications. Using real-world malware and benign applications, we conducted experiments on Android mobile devices. Our data shows the effectiveness of our developed detection system.

References

1. What is Android? http://android.pk/android.html.
2. Smartphones account for half of all mobile phones, dominate new phone purchases in the us. http://www.nielsen.com/us/en/newswire/2012/smartphones-account-for-half-of-all-mobile-phones-dominate-new-phone-purchases-inhtml.
3. A. Nere, A. Hashmi, M. Lipasti, and G. Tononi: Bridging the Semantic Gap: Emulating Biological Neuronal Behaviors with Simple Digital Neurons. In Proceedings of IEEE 19th International Symposium on High Perfor- mance Computer Architecture (HPCA), (2013).

4. D. J. Montana and L. Davis: Training Feedforward Neural Networks Using Ge- netic. In Proceedings of International Joint Conference on Artificial Intelligence Algorithms, (1989).
5. X. Yu, M. O. Efe, and O. Kaynak: A General Backpropagation Algorithm for Feedforward Neural Networks Learning. In IEEE Transactions on Neural Net- works, vol. 13, pp. 251-254 (2002).
6. G. Arulampalam and A. Bouzerdoum: A Generalized Feedforward Neural Network Architecture for Classification and Regression. In Journal of Neural Networks, vol. 16, pp. 561-568 (2003).
7. J. Y. F. Yam and T. W. S. Chow: A Weight Initialization Method for Improving Training Speed in Feedforward Neural Network. In Neurocomputing, vol. 30, pp. 219-232 (2000).
8. S. Kak: On Training Feedforward Neural Networks. In Pramana-Journal of Physics, vol. 40, pp. 35-42 (1993).
9. A. D. Schmidt, R. Bye, H. G. Schmidt, J. H. Clausen, O. Kiraz, K. Yuksel, S. A. Camtepe, and S. Albayrak: Static Analysis of Executables for Collaborative Malware Detection on Android. In Proceedings of the IEEE International Conference on Communications (ICC), (2009).
10. M. Grace, Y. Zhou, Z. Wang, and X. Jiang: Systematic Detection of Capability Leaks in Stock Android Smartphones. In Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS), (2012).
11. I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani: Crowdroid: Behavior-based Mal- ware Detection System for Android. In Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, (2011).
12. A. Bose, X. Hu, K. G. Shin, and T. Park: Behavioral Detection of Malware on Mobile Handsets. In Proceedings of the 6th ACM International Conference on Mobile Systems, Applications, and Services, (2008).
13. A. S. Shamili, C. Bauckhage, and T. Alpcan: Malware Detection on Mobile Devices using Distributed Machine Learning. In Proceedings of 20th IEEE International Conference on Pattern Recognition (ICPR), (2010).
14. D. Venugopal and G. Hu: Efficient Signature based Malware Detection on Mobile Devices. In Journal of Mobile Information Systems, vol. 4, no. 1, pp. 33- 49 (2008).
15. A. D. Schmidt, R. Bye, H. G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak: Static Analysis of Executables for Collaborative Malware Detection on Android. In Proceedings of IEEE International Conference on Communications (ICC), (2009).
16. A. Shabtai: Malware Detection on Mobile Devices. In Proceedings of the 11th IEEE International Conference on Mobile Data Management (MDM), pp. (2010).
17. A. Dinaburg, P. Royal, M. Sharif, and W. Lee: Ether: Malware Analysis via Hardware Virtualization Extensions. In Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS), (2008).
18. Z. Aung and W. Zaw: Permission-Based Android Malware Detection. In International Journal of Scientific and Technology Research, vol. 2 (2013).
19. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji: A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS), (2010).
20. C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu: Performance Evaluation on Permission-based Detection for Android Malware. In Springer Berlin Heidelberg, pp. 111-120 (2013).
21. J. Cannady: Artificial Neural Networks for Misuse Detection. In Proceedings of National Information Systems Security Conference, (1998).
22. S. Mukkamala, G. Janoski, and A. Sung: Intrusion Detection Using Neural Networks and Support Vector Machines. In Proceedings of IEEE International Joint Conference on Neural Networks, (2002).
23. O. Linda, T. Vollmer, and M. Manic: Neural Network based Intrusion Detection System for Critical Infrastructures. In Proceedings of IEEE International Joint Conference on Neural Networks, (2009).
24. V. Golovko, S. Bezobrazov, P. Kachurka, and L. Vaitsekhovich: Neural Network and Artificial Immune Systems for Malware and Network Intrusion Detection. In Advances in Machine Learning II. Springer, pp. 485-513 (2010).