Android Application Install-time Permission Validation and Run-time Malicious Pattern Detection

Zhongmin Ma

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

November 4, 2013
Arlington, Virginia

Keywords: Android, Security, Android Permission Label, Machine Learning
Neural Network, Action, Intent, Intent Log, Hidden Markov Model

Android Application Install-time Permission Validation and Run-time Malicious Pattern Detection

Zhongmin Ma

(ABSTRACT)

The open source structure of Android applications introduces security vulnerabilities that can be readily exploited by third-party applications. We address certain vulnerabilities at both installation and runtime using machine learning. Effective classification techniques with neural networks can be used to verify the application categories on installation. We devise a novel application category verification methodology that involves machine learning the application permissions and estimating the likelihoods of different categories. To detect malicious patterns in runtime, we present a Hidden Markov Model (HMM) method to analyze the activity usage by tracking Intent log information. After applying our technique to nearly 1,700 popular third-party Android applications and malware, we report that a major portion of the category declarations were judged correctly. This demonstrates the effectiveness of neural network decision engines in validating Android application categories. The approach, using HMM to analyze the Intent log for the detection of malicious runtime behavior, is new. The test results show promise with a limited input dataset (69.7% accuracy). To improve the performance, further work will be carried out to: increase the dataset size by adding game applications, to optimize Baum-Welch algorithm parameters, and to balance the size of the Intent sequence. To better emulate the participant's usage, some popular applications can be selected in advance, and the remainder can be randomly chosen.

Acknowledgments

I would like to express my deepest gratitude to my advisor Dr. T. Charles Clancy for his guidance and constant support in helping me to conduct and complete this work. I would also like to thank my advisory committee, fellow students especially Chen Yang for his inspiration and continuous encouragement during our research.

Contents

1 Introduction **1**

1.1 Android Architecture 2

1.2 Overview of Android Security 3

1.3 Android Security Problems 5

1.4 Category/permission based security 6

1.5 Contribution 7

2 Review of Literature **8**

2.1 Static Analysis 8

2.2 Dynamic Analysis 10

3 Permission Anomaly Detection using Neural Network **12**

3.1 Overview of Neural Network 13

3.2 Neural Network Classification 14

3.3 Dataset 17

3.4 Experiment Results 21

4 Runtime Malicious Intent Detection using HMM **28**

4.1 Application Tag Components 28

4.2 Intent Log components 31

4.3 Hidden Markov Model 32

4.4 Dataset and Experiments 35

5 Conclusion **40**

Bibliography 42

Appendix A. Android Permission Constants **46**

Appendix B. Android Action Constants **53**

Appendix C. Matlab code for BW algorithm **59**

List of Figures

1.1 Android System Architecture 2

1.2 Permission labels of com.ebay.mobile app 5

3.1 A typical supervised learning model 15

3.2 Screen shot of apktool 18

3.3 Android permissions example 18

3.4 Permission request distribution of the dataset 19

3.5 Relative permission requests of the dataset 21

3.6 Neural Network Toolbox Training Process Diagram 22

3.7 Distribution of good and malicious apps 24

3.8 FPR / FNR and threshold 26

4.1 Activity and Service tags 29

4.2 Sample output of app Intent Log information 31

4.3 Pseudo code of BW algorithm 35

4.4 training and testing process diagram by using BW algorithm 38

List of Tables

3.1 Unique Requested Permissions and Most Appeared Apps in Each Category . 20

3.2 Accuracy of Distinct Training Algorithm 23

3.3 Performance Measurement 26

4.1 Performance Measurement using HMM 38

Chapter 1

Introduction

In 2012, the worldwide sales of mobile devices to end-users totaled 1.75 billion units. However, the traditional PC market share continues to decline and a long-term decline among worldwide computing devices is expected. A smartphone is a mobile device with a modern operation system (OS) which provides more advanced capabilities than a 'feature' phone. Lower price, application (app) popularity and cloud storage facility has made more and more users embrace the smartphone instead of the feature phone. Compared to a feature phone, a smartphone has more processing power and a larger display screen. Smartphone OS include Google's Android, Apple's iOS, Nokia's Symbian, RIM's BlackBerry and Microsoft's Windows Phone. Android is a Linux-based open source smartphone OS. Android OS has many features: it supports connectivity including Bluetooth, WIFI, WiMAX, CDMA, EVDO and LTE; it uses short message service/multimedia messaging service (SMS/MMS) for message and text; it also integrates web browser, camera, GPS, storage among other capabilities.

Currently, the smartphone OS market is dominated by the Android. In 2012, the number of smartphones based on Android OS was 497 million. Android smartphones continue to significantly increase their market share with an overall 122.5 million sales to end users in the third quarter of 2012. This accounted for 72.4% of worldwide smartphone sales [1].

The Android is an open source platform which allows the device manufacturers, wireless carrier, and developers to freely develop and distribute all kinds of apps. As an open source platform, the Android requires a robust architecture and enhanced security features to protect user data and system resources from malware attacks, and to provide app's isolation to reduce the probability of potential malware attacks.

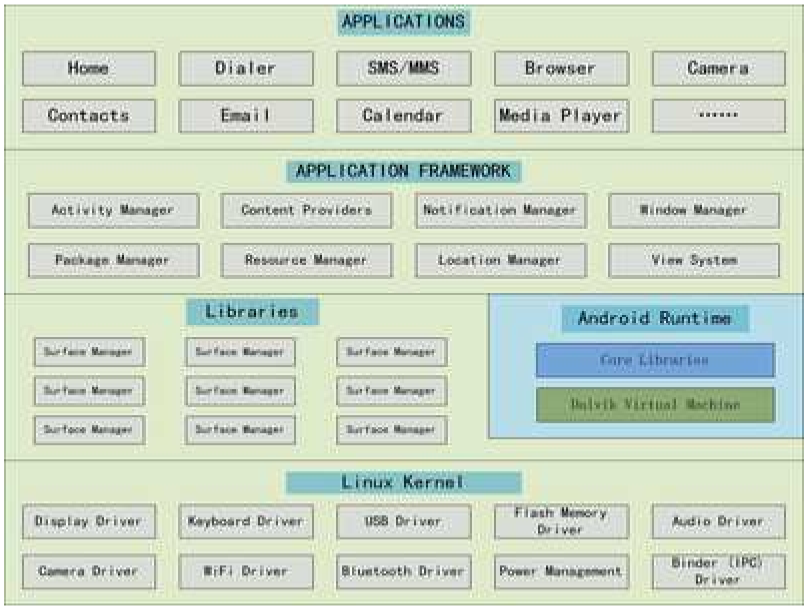


Figure 1.1: Android System Architecture

1.1 Android Architecture

The Android software stack as shown in Figure 1.1 consists of four layers: the Linux Kernel layer, Runtime layer, Framework layer and Application layer. The Linux Kernel layer is the first layer of the Android components. It is primarily responsible for interacting with the hardware which includes memory management, networking and security, driver management, and power management. The Runtime layer consists of two parts: libraries and virtual machine. Android runtime is the engine that powers the applications and the libraries. It forms the basis for the application framework [2]. All of the runtime programs including Media, SQLite, Webkit, and SSL are developed in C/C++. The Application framework layer and Application layer are normally developed in Java. To enable rich media functionality, Libraries are designed to unite all runtime programs and let Java developers use them through a Java API. Dalvik Virtual Machine (VM) is designed to allow apps to run on portable devices using low-level memory management. Dalvik is not a traditional Java VM. It is designed to ensure that multiple instances run efficiently on the Linux kernel [3]. The Application Framework provides lots of service functionalities including Activity Manager, Notification Manager, Window Manager, etc. With these services, developers have a rich API set to develop apps. The Applications layer is the top layer and is directly operated by the device user. Normal Android app would include the following components: Home, Phone, Contacts, SMS, Browser and Camera etc. In addition, most of the apps are written in Java and compiled to .dex file type which can executed by Dalvik VM.

1.2 Overview of Android Security

Since smartphone users are likely to store valuable and sensitive personal data such as messages, SMS, contacts, photos and videos on their smartphones, it is very important to ensure system security on the Android platform. To satisfy these security requirements, Android provides several key features. These key features include: Linux Kernel level security, application sandbox, application signing, application-defined, and user-granted permissions. Below we briefly discuss these important security features.

The Android platform uses the Linux kernel as its operating system core. As a proven stable and reliable kernel, Linux provides the data isolation mechanism to solve multi-user's data usage and the user-based permission model. Unlike the traditional Linux system, Android assigns a unique user ID and group IDs to each android app and thus lets different apps use their specific defined permissions.

Being a Linux-based OS, the Android creates an isolation boundary around each process [4]. This data isolation technique is called Sandboxing. It ensures that each app runs in its own Linux process instead of others. All components above the Linux Kernel, such as Applications, Application Framework, Libraries and Android runtime, are restricted within the Application Sandbox. Because Application Sandbox is based on Linux kernel, it makes this Sandbox very robust and difficult to be hacked or compromised. It enhances the security at the kernel-level. If a malicious app wants to interact with other apps without gaining the user's privileges, the Sandbox mechanism forbids this app's interactions on the process level. In addition, if data sharing is incurred between apps, it has to be explicitly declared. For example, one app cannot read or write directly from the device's contact list or messages without getting the requested communication approval. If two apps share the same user ID, they must be signed by the same author and declared explicitly [4, p.370-372].

The Android provides a securable Inter-Process Communication(IPC) mechanism to share data, reuse components, and pass relative data properties among the applications. These IPC mechanisms mainly include Intent and Binder. Intent mechanisms are used to communicate different activities in the same app or different apps. Intents specify activities by using *startActivity* calls or inform changes or events by using broadcast calls. Activities are used to communicate with users; Intents are used to start services, such as write email, play music, and online shopping. After the services are started, the related messages between applications or services can be sent through the broadcast mechanism to Intent. Then Intent receives those messages and calls the correspondent activity. Normally, Intent is restricted by manifest permissions. Binder is a top level abstraction of Intent. It is a remote procedure call mechanism designed for performing in-process and cross-process calls [3]. Binder is used to bridge Java and native language code running in different processes.

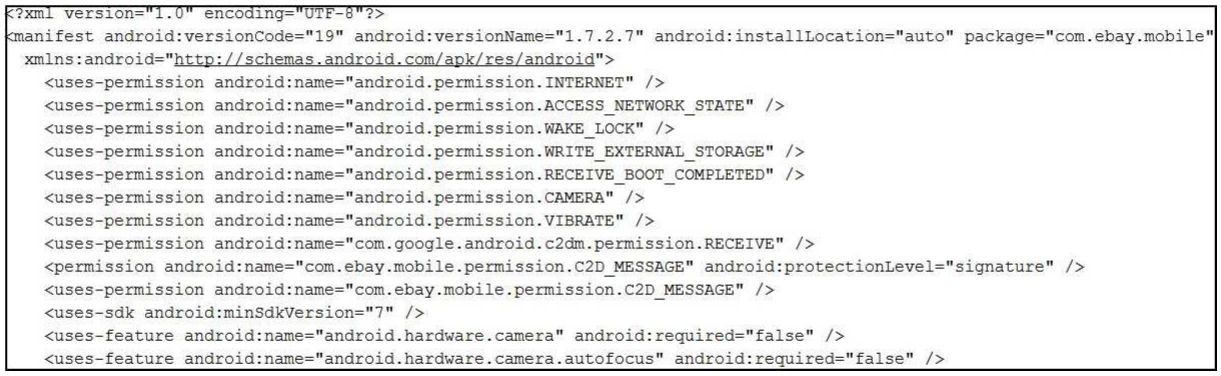
Before we analyze Android app security, we need to know how Android apps are constructed and how they are bound by using the Android manifest. Once we understand these app compositions and inner mechanisms, we can analyze the permissions and categories in our project at install time.

A standard app normally comprised six components: *Activity*, *Service*, *Content Provider*, *Intent*, *broadcast receiver*, and *Notification*. These components are configured by AndroidManifest.xml which is located in the root of an app's directory. The manifest file lets users define the app structure, metadata, and components. It uses Intent filter and permission to determine how the components interact with each other and how to communicate with other apps. We will describe the details of Intent filter in Chapter 4.

Permission is designed to minimize the impact of malicious apps. Apps need to get a grant or permission from users before they perform operations like using the camera, sending messages, accessing private data, dialing calls. If there are no permissions available, an app can not do anything. Starting Activities, sending/receiving broadcast Intents and invoking Binder all require manifest permissions. Permissions are managed in the AndroidManifest.xml file and the user agrees or denies the related configuration settings at install time. If some permission label is not granted in this xml file, any related execution or access will result in a permission failure.

Permission tags are mainly used to restrict the access level. Depending on the protection level, Android may perform a different action whether to grant the permission or not. There are four permission protection levels: normal, dangerous, signature and signatureOrSystem. The default value is 'normal'. It is a low-risk permission and the system automatically grants the permission at installation without being detrimental to the system. Dangerous is a high-risk permission that the system would grant the requesting app to access private user data or control over the device. Signature is that system which grants the permission only if the requesting app has been signed with the same certification as the app declaring the permission [5]. In Figure 1.2, the "com.ebay.mobile" app sets the customized permission protection level as 'signature'. If the certification matches, the system automatically grants the permission without getting the user's explicit approval. The signatureOrSystem level tells the system to grant the permission to apps with the same signature. It is only used in special cases where multiple vendors have apps built on a system image and need to share explicit features.

App components require the permission label by adding the *android.permission.** attribute to explicitly spell it out. For example, if one app will need Internet to operate, the app's developer must add *android.permission.INTERNET* permission into the AndroidManifest.XML. The *Uses-permission* tags are the main part of the security model. They declare required permissions which are presented to the user at install time. Currently, the user cannot choose specific permissions; the user has to make a choice to either grant or deny all permissions.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<manifest android:versionCode="19" android:versionName="1.7.2.7" android:installLocation="auto" package="com.ebay.mobile"
  xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.VIBRATE" />
    <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
    <permission android:name="com.ebay.mobile.permission.C2D_MESSAGE" android:protectionLevel="signature" />
    <uses-permission android:name="com.ebay.mobile.permission.C2D_MESSAGE" />
    <uses-sdk android:minSdkVersion="7" />
    <uses-feature android:name="android.hardware.camera" android:required="false" />
    <uses-feature android:name="android.hardware.camera.autofocus" android:required="false" />
```

Figure 1.2: Permission labels of com.ebay.mobile app

Furthermore, an app's permissions cannot be changed once installed. How to minimize the potential insecure app flaws and detect the potential malicious apps at install time is very important.

1.3 Android Security Problems

Mobile malware evolves very quickly. An analysis by Gostev[6] concluded that mobile viruses evolve in only two years to cover the same ground as computer viruses over twenty years of evolution. According to the statistics of the Kaspersky Lab—one of the most famous anti-virus software providers—the number of Android Trojan attacks in the 2nd quarter of 2012 nearly tripled from the 1st quarter of 2012. Because the payload of today's smartphone malware is very similar to the computer malware, the malicious code is easy to write. As an open source platform, the Android is very vulnerable to the mobile malicious attack.

The other statistics show the test results of Android anti-virus app performance. Among a total of 42 anti-virus apps, only 7 anti-virus apps are able to catch over 90% of Android malware loaded on test devices [7].

When Android users install an app for the first time, users must grant or deny specific permissions as required by the app. But in real life, most users may have insufficient knowledge to distinguish whether the permissions of currently installing apps are benign, rather than intentionally malicious. Although some users can discern the potential malicious permissions, their devices may still be infected by some seemingly good apps. Furthermore, inexperienced developers may add redundant permissions, which make Android security problems worse. These developers may introduce vulnerabilities or side effects into their customer's Android system. In the April of 2011, one developer found a critical vulnerability in the Android version of the Skype. Skype stored the chat logs and other private information on the MicroSD card with improper permissions. This allowed anyone or any app to read those files, because they were accessible and completely unencrypted.

All of the above statistics and examples tell us that Android security has become a very severe problem. Because Android application programming interface(API) is open to the public, it is not very hard to break out of the Dalvik VM. The Android needs to maintain its security even if the VM is compromised. How to make manifest permissions to satisfy this requirement is a big challenge to the developers. They should specify manifest permissions more accurately and restrict the abuse of unnecessary permissions labels. For example, a game app should not include the *android.permission.READ_CONTACTS* permission label. Because this permission is used to read the user's address book, a good strategy is to forbid this label even if the game wants to let users send a high-score to remote servers. This can be done by using *android.permission.INTERNET*.

1.4 Category/permission based security

In any Android app store, apps are always divided into different categories, such as *Business*, *Books&Reference*, *Travel*, *Game*, and others. Each category has its specific functionalities, and different apps in the same category usually share several similar permissions, because permissions defined by the Android are responsible for specific functionalities. For instance, the *Books&Reference* category normally consists of all kinds of free or paid electronic(E)-book apps, online book shopping apps, book management apps, or bookrelated apps. If an app in the *Books&Reference* category requests *READ_CONTACTS* or *WRITE_CONTACTS* permission to read or write user contact information, this is very abnormal, since most of the good apps in this category do not need this kind of permission. Read or write permission usually let users send an E-mail, message, or call to some specific person. However, the category permission requests *READ_CONTACTS* or *WRITE_CONTACTS* from Communication apps should be fine.

Apps in different categories may request different permissions, while apps in the same category have very high possibilities of using more or less similar permissions. Large permission discrepancies between two apps suggest that these two apps should belong to different categories; otherwise, the apps should be carefully checked.

1.5 Contribution

Up to now, the Android permission policy has been very strict, and the Android does not have any runtime limitation or behavior censorship. Once installed, an app can run any activity including the malicious activity.

This thesis presents two-pronged approaches for the Android system security. During an app installation, we use neural network to cluster the category permissions and to validate the potential malicious permissions. Currently, many Android researchers study permission enforcements and malicious app detection by parsing code or API; yet, none of the published papers analyzed the malicious behaviors based on the category. We present a novel viewpoint in Android app installation security.

The other approach presented here is runtime suspicious behavior detection by using HMMs to track Intent log information. Currently, no related reference or published paper takes a similar approach to identity malicious activities by analyzing Intent activities information. We provide an original idea.

The remainder of this thesis is organized as follows: chapter 2 presents related work on the Android security both at install time and runtime, and chapter 3 explains the proposed methodology to validate the app categories by using a neural network approach. Some of the experiments and results come from previous research [8]. In Chapter 4 the thesis discusses runtime Intent activities detection based on HMMs. Finally some concluding remarks are given.

Chapter 2

Review of Literature

Our project is divided into two parts: permission anomaly detection at install time and malicious intent detection at runtime, which are also called the static detection and the dynamic detection, respectively. In this literature review, we first focus on the permission regulation at installation; then, we analyze a malicious pattern from intent activities between an app and its opponent at runtime.

2.1 Static Analysis

Enck et al.[9] developed the Kirin tool to check an app's permission requirements at install time. Kirin compares declared permissions with a set of security rules to detect potential malicious apps. Kirin ensures that an app satisfies Android security requirement at install time; if not, the app installation will terminate. The user does not make decisions to grant or deny the installation. Kirin creates several security rules and automatically validates an app's security state via the policy engine. As a well-known static framework, Kirin only provides an install time policy check instead of a dynamic security check. Moreover, it can not detect specific malicious apps that are capable of sending spam or creating premium SMS messages without accessing private information.

Barrera et al.[10] use self organizing maps (SOM), a neural network algorithm. As an unsupervised learning algorithm, SOM provides a 2-dimensional visualization of the high data dimension, thus reducing data size and displaying the similarities among data. At the start, they initialize neuron weights and present input sample vectors corresponding to an app. Based on SOM characteristics, they compute the minimum Euclidean distance of each sample vector with all the related neurons. The weight of a specific neuron with the shortest distance is the best matching unit. Next, the neighborhood of this matching unit is computed. As the value adjusts close to the input vector, the weights of the winning unit are adjusted as well as its neighbors. At the same time, the numbers of neighbors decrease over time. In this situation, the closer a neuron is to the best matching unit; the more its weights are adjusted. The farther away from the best matching unit, the less adjustment it learns. Finally the process is repeated for a large number of iterations. In their paper, they find that only a small number of permissions are frequently used by developers. This finding is in accordance with our research that the *INTERNET* permission is the most popular label. They hypothesize that this label is overused by the developers which permit ads send from remote servers. This methodology also proves that apps are clustered based on the requested permissions. Categories of Android Play Market are based on the semantic activity instead of the technical feature, because different neighbor categories can request similar sets of permissions. They also make observations about location-based permissions and specific category features. Location-based permission always require a pair of permissions: Fine and Coarse access location labels. The limitation of this paper is that the performance of SOM is largely depended on the validity parameters. However, these parameters are merely subjectively determined and not mathematically well-defined. The paper performs statistical analysis, but it does not address the detection or validation issues.

Some researchers focus on improving the Android security and access control models. Vidas et al.[11] present a tool that can extract permission specification by parsing the Android documentation. This aid tool can check unnecessary permissions by analyzing the source code to automatically compare the minimal set of permissions and required permissions. Since the Android documentation undergoes updates periodically, this specification must be modified upon the future changes. Similar tools have been designed by Felt et al.[12] to address permission re-delegation and present IPC inspection.

Currently, the introduction of the permissions label on the Android official website is too abstract and short of sufficient explanations and definitions. Felt et al.[13], via Internet survey and research studies, found that current permission warnings do not guide users in making correct decisions to grant or deny apps.

Stowaway [14] is composed of a static analysis tool to determine what kind of API is called for by an app request. It presents a permission map to identify the required permissions. Stowaway analyzed 940 apps and found that one-third of these were over-privileged. Among the over-privileged apps, approximately 56% apps had more than an extra permission; though, a low degree of per-application over-privilege suggested that the developers are not intentionally adding unneeded permissions. Developer errors are responsible for these unnecessary permissions. For example, some developers cannot distinguish the differences between *ACCESS_WIFI_STATE* and *ACCESS_NETWORK_STATE*, so both are requested, although one is undesired. Other reasons for over-privilege may include: inaccurate source code and information is copied and pasted into apps by developers; deprecated permissions are still used; testing labels like *ACCESS_MOCK_LOCATION* is used during the simulation stage but remains in the released apps. In addition, developers sometimes intentionally request extra permissions to send ads; this also conforms to Barrera's hypothesis.

Felt et al. [15] performed case studies on app permissions of the Google Chrome extension system and the Android OS. They surveyed 100 paid apps and 856 free apps from Google Play Market. They estimated that about 93% of free and 82% of paid apps requested a dangerous permission at least. It requires that users grant the dangerous permissions without paying enough attention to the possible consequences. However, the above works [11, 12, 13, 15, 14] did not try to detect or categorize malicious apps.

Sarma et al. [16] investigated the feasibility of using both the requested permissions in an app and what permissions are requested by other apps in the same category to evaluate the risks and benefits of an app.

Enck et al. [17] decompiled and analyzed the source of apps to detect further leaks and usage of data. They described Android apps that often unnecessarily access user's personal data by studying 1,100 popular Android free apps with the help of tools and observations.

Au et al. [18] surveyed permission systems of several popular smartphone OS and categorized them by the amount of control they give to users, the amount of information they convey to users and the level of interactivity they require from users. In addition, they discuss the problems linked with extracting permission-based information from Android apps.

2.2 Dynamic Analysis

Ongtang presents Saint [19], a modified infrastructure that manages install time and runtime permission assignments, as dictated by app provider policy. Saint provides the necessary utility for apps to assert and control the security on an Android system. As a policy enforcement framework, Saint can check the static policy at install time and dynamic policy at runtime. Saint creates an AppPolicy Provider to store install time and runtime policies. Upon install time, Saint Installer retrieves the requested permissions and compares them with the rules of AppPolicy Provider. If the rules match the requested permissions, the installation proceeds; otherwise, the installation cancels. Saint's runtime policy regulates the interactions between a caller app and a callee app. For the runtime enforcement policy, a caller app sends an IPC and a callee app receives this IPC. The IPC is allowed to proceed only if the conditions of both the caller and the callee are satisfied. Saint Mediator is responsible for retrieving policy information from apps; later it sends these messages to Saint AppPolicy Provider. AppPolicy Provider compares these permissions and conditions and, if the result matches, the IPC can continue; otherwise, the IPC is blocked. However, the actual context cannot grant access control because defining context policy in Saint is not possible. Furthermore, Saint only provides policy enforcement; no feasible evaluation and experiments support this framework.

Since the Android does not permit users to manually grant requested permissions, users must either accept all permissions or deny all of them. Apex [20] is a policy enforcement framework which lets users grant permissions individually or impose constraints on the resources during install time. Apex also allows users to specify runtime constraints on the use of sensitive data by apps. However, users may unwarily grant dangerous permissions, which may violate a larger security goal. In addition, there have been no evaluations of, nor or experiments on Apex for app analysis, and Apex is a framework.

SCanDroid [21], a modular automated reasoning analyzer, allows the incremental checking of apps at runtime. This analyzer extracts requested permissions from an app and checks consistency between permissions, as well as data flows through this app. SCanDroid is the first program analysis tool; however, it is not tested on Android apps. The other limitation is that the Android app security certification requires the availability of either the Java source code or the compiled JVM byte code of apps.

TaintDroid [22] is a framework that performs a dynamic analysis. It tracks sensitive data-API return values via apps. Results from a TaintDroid monitoring analysis of 30 apps showed that 20 of these apps may incorrectly use sensitive information. Half of the sample apps may report the user's location, while seven apps gain information for the developers, such as the SIM serial, device identification, and phone numbers. These findings remind us that protecting personal data during Android app installation and usage is very beneficial. However, TaintDroid only tracks data flows like explicit flows; it cannot track control flows. In addition, the dataset is very limited.

At present, analyzing the Android security with machine learning methods focuses on the context of spam detection, which is out of Android's scope and anomaly detection in network traffic flows. We detect Android malicious patterns based on Intent log at runtime and detection of permissions anomalies at install time.

Chapter 3

Permission Anomaly Detection using Neural Network

Android has become the most popular mobile platform, and its third-party app store is a big target for cybercriminals. Google Play Market [23] is the biggest online Android app store; alternative Android app stores include Samsung Apps, Amazon Appstore, Verizon Wireless, Appia, Exent, and AppBackr. In our project, we selected Google Play Market as our data resource. Attackers can exploit the vulnerabilities of inexperienced app developers or pose the malicious apps on online stores. Although online stores frequently reject the attacks and remove malicious apps, many android devices have already installed malware and are infected. These infected devices may send SMS/MMS messages, make a phone call to high-rate numbers, steal private data from storage, or incur unwanted service charges on users' accounts. To defend against malware attacks, anti-virus actions must be taken at install time to check whether apps have malicious granted permissions. Runtime malicious activities detection and prevention is the second line of defense.

Online app stores classify apps into different categories based on their functionalities. A system-defined permission represents its unique functionality. Therefore, apps in the same category may have, more or less, the same permissions. Large permission discrepancies among different apps suggest that these should (with very high possibility) belong to different categories, because apps in the same category always have similar usages and functionalities. Good apps in the same category may request the same permissions to implement their similar functionalities. Malicious apps have different characteristics from normal apps in the same category, e.g., they may use very low frequency safe permissions or request dangerous permissions which other good apps in the same category do not request. Well-designed apps must have some subtle relationships with their associated category; on the contrary, malicious apps may have different relationships with their associated category. This is a very complicated problem which cannot be expressed as a series of well-defined steps. Because of these subtle conundrums, it is impossible to know the exact result in advance. We need technology to extract the hidden relationships between an app and its category, then to identify which app is malicious and which is not. Neural network (NN), a machine learning method, is very suitable for our project since it can cluster malicious patterns and good patterns by analysis of these discrepancies. NN can be trained by a large size of apps and can be self-adjusted from inner app-category differences.

Machine learning is used for the construction and study of systems that can be trained and learned from the data. In this thesis, the dataset is created by parsing app permission labels, and the experiment results confirm that NN is a robust tool to properly identify apps whose permissions are inconsistent with their category, and are perhaps malicious.

3.1 Overview of Neural Network

There is no commonly accepted definition of a NN. When we talk about a NN, we are referring to an “artificial neural network”. Although the studies of NN were performed on the biological human brain, much of the inspiration for NN catered to the needs of producing artificial systems, i.e., for making intelligent computations, as does the human brain. Generally, the actual NN refers to the mathematical models instead of the more complex biological neural network. Like Nigrin states: “A neural network is a circuit composed of a very large number of simple processing elements that are neutrally based. Each element operates only on local information. Furthermore, each element operates asynchronously; thus, there is no overall system clock.”

NN is a network of many simple units; each unit connects with the others. A unit usually carries numeric data and is capable of carrying a function. NN has different usages: classification, pattern recognition, and prediction. Classification is designed to analyze input training samples and classifies them into groups. Pattern recognition is the most used process for a NN. According to a training input, NN tries to find a recognized pattern which can differentiate and match some pattern for which the sample data has been trained. Pattern recognition is mainly used in feature extraction, image matching, and speech recognition. Prediction is an extrapolation based on historical data. It usually makes claims about the future—for example, energy consumption, earthquake prediction, stock market price and currency-based on the current and past state.

How does a NN work? The strategy of a NN is based on the input/output data or input data only. A NN sets suitable values for connection weight and threshold, allowing the activation function to perform well and allowing optimal results output. The output of a neuron is a function of the weighted sum of the inputs, plus a bias [26].

To better illustrate a NN, a single neuron example is provided. We denote m input values as $[x_1, x_2, x_3, \ldots, x_m]$, each of the input values has its own synapse (connection) to a neuron, and each has a weigh: $[w_1, w_2, w_3, \ldots, w_m]$. In a NN, the predicted input of this neuron is $v = x_1w_1+x_2w_2+x_3w_3+\ldots+x_mw_m$. A bias value should be added to shift the activation function to the left or the right. If there is no bias, users have to train a threshold for each output besides train the weights value. Since a standard NN includes many outputs, it is impractical to setup a NN learning algorithm like Back Propagation algorithm without a bias. Therefore, the sum of input value for the neuron is: $v = x_1w_1 + x_2w_2 + x_3w_3 + \ldots + x_mw_m + bias$.

The output of the neuron is some function of the input values: $y = f(v)$. In a NN, this function is normally called an activation function. An activation function is normally hidden in a NN. A linear activation function and a sigmoid activation function is the most commonly used functions. For a linear activation function, a neuron will become a linear model with bias parameter α. As in a binary output result, $if\ v > \alpha, f(v) = 1; otherwise, f(v) = 0$.

The standard NN is made up of neurons and their interconnections. Each connection is assigned with a connection weight. The weights and the bias are called adjustable parameters and play a key role in determining the output of NNs. The main goal of a NN is to allow the network to exhibit the desired performance by adjusting these parameters.

3.2 Neural Network Classification

Generally, the common structure of NNs normally includes three layers: an input layer, an output layer, and one or more hidden layers. NN includes three main categories based on the learning algorithm—supervised, unsupervised, and reinforcement—and based on the network topology, a NN is divided into two classifications: feedforward and recurrent.

The core part in a NN is the learning process. Learning implies to get knowledge of by studying, experiencing, or teaching. Learning lets allows NNs to adjust the parameters of neurons so that the networks can finish a specific task. Supervised learning is the most popular learning algorithm. Supervised learning provides a NN with a training pair (the inputs and desired outputs). The main idea of a supervised training is that a NN has knownis aware of the requested outputs and automatically modifies the coefficients, using certain criteria [27, 28], to let obtain a the computed result that is as close as possible to the desired output by using some criteria.

Figure 3.1 shows that the input and target output have been provided by the system. Supervised learning algorithms will learn and modify the threshold and weights by the feedback of the error vector.

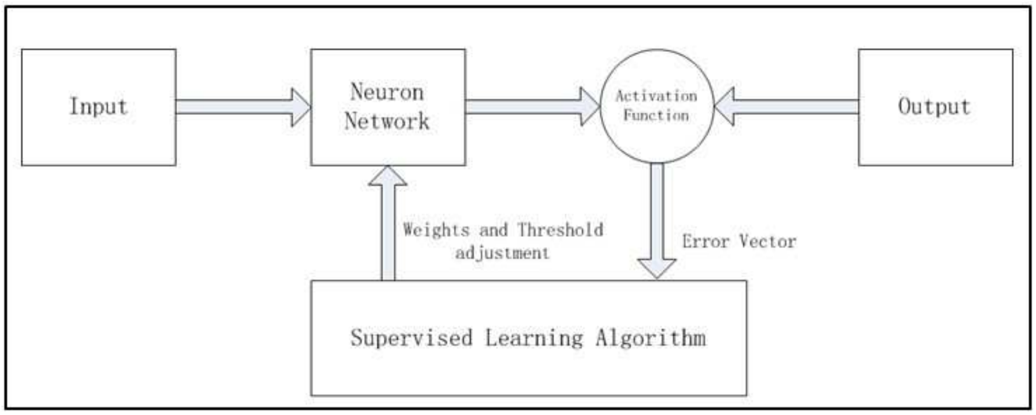

Input
Neuron Network
Activation Function
Output
Weights and Threshold adjustment
Error Vector
Supervised Learning Algorithm

Figure 3.1: A typical supervised learning model

Unsupervised learning is also called self-organization due to the lack of feedback. In this learning algorithm, all similar input patterns are grouped as clusters. Compared to the supervised learning paradigm, no previous induced information can be referenced. The system must discover patterns and regularities from the input data and develop its own representation of the output. If a new input pattern is not found, then a new cluster is created.

The reinforcement learning algorithm uses both the supervised and unsupervised training data, and can therefore be considered an intermediate form of the supervised and unsupervised learning. Reinforcement learning takes action on, and receives a response from, the environment. The learning system grades its action as a reward or penalty based on the environmental response and accordingly adjusts its parameters. Generally, the feedback in this algorithm is only evaluative, not instructive.

The above three learning algorithms represent different approaches. The outputs of all depend on the space of interconnected neurons. Supervised learning adjusts the weight with the help of an error response signal, whereas unsupervised learning exploits the information associated with the distribution of input patterns and neuronal updating. Reinforcement learning functions through sample and epoch iteration with its environment, if less information is available about the critic information [29].

Feedforward NN is a network featuring the flow of connections is only in one direction, forward. Traditional feed-forward networks consist of three parts: an input layer, a hidden layer, and an output layer. In this simple and stable network, information flows from the input layer to the hidden layer, then to the output layer, with no cycle connection. On the contrary, recurrent NN, also called a feedback NN, is a network allowing connections to form a cycle. Recurrent NN is dynamic, since the internal states change constantly before reaching equilibrium. Although recurrent NN is very powerful, the feedback training is very difficult and performance may be problematic, since dynamic features bring non-stable output results and unexpected oscillation and chaos. In practice, most NNs use the feedforward, rather than the recurrent, structure.

Feedforward neural network falls into two categories: a single-layer perceptron (neuron) and a multi-layer perceptron. Feedforward NN includes three features: first, perceptrons with the same layer cannot contact each other; second, each connection has only one direction, excluding backward, i.e., each perceptron is connected to its next layer; third, hidden layers cannot be seen by the outside. A single layer feedforward NN has a single layer of weights, i.e., the training process only uses a single weights adjustment. This NN is mainly used in linear separable problems.

Multilayer Feedforward (MLF) network is the most common NN. In this model, the outputs of one layer act as the inputs to its next layer [30], and back propagation (BP) learning is the preferred training algorithm. The BP algorithm is the best known example of a NN training algorithm. In practice, about 80%–90% NNs use BP as their algorithm. It calculates derivative flow backwards through the network. First, it is an easy algorithm to understand. In BP, the calculated gradient vector of the error surface moves with very short steps along with the line of steepest descent from the current point. A series of such moves can reveal a minimum of some sort [31]. The performance largely depends on the step size, or learning rate, of the moves. Learning rate is application dependent; it can be selected by experiments at various times. A large learning rate reduces the execution time, but it inevitably results in bad convergence or moves in the wrong direction. A small learning rate in the opposite direction can result in the maximum decrease of the local error function. However, a very small learning rate means a very large number of iterations, which drastically slow execution time, to attain convergence.

NN is used for learning in feedforward networks belonging to the domain of supervised learning. The number of neurons in the hidden layers is important for the overall NN performance. Increasing the number of neurons leads to an increase in weights; therefore, the NN execution time can increase greatly. In practice, adding new hidden layers to improve the accuracy, which also helps in the number of weights and computational time reduction, is better. However, a NN with more than three hidden layers is uncommon.

There are two disadvantages to BP learning. BP learning often takes a long time to converge; some sigmoid functions need thousands of epochs. Generalization is not guaranteed in Bayesian regulation, due to overfitting problems.

The number of hidden layers, and neurons in each hidden layer, should be carefully set. Too few neurons will result in underfitting; however, too many neurons may result in overfitting and a longer training time. Underfitting normally happens with too few neurons, leading to large bias and a large expected generalization error. Overfitting occurs when the learning performs well on the used data during the training, though badly on the new data. The model captures not only the regularities, but also the peculiarities, in the training dataset. Overfitting the network may allow memorization of the training data, making the network less able to generalize the unseen data [33]. Overfitting problems create noise data and thus result in a large expected generalization error. Since a NN with more weight models a more complicated function, the result is more prone to overfitting.

In our project, we start with a small number of neurons and adjust the number according to a NN performance measure. To avoid underfitting and overfitting, during the training time, the system parameter was adjusted and the training error naturally dropped, in order to minimize a true error function. Therefore the selection error was reduced as well, and if this parameter arises during training, it implies the NN overfit the training data. If this were to happen, we would reduce the number of hidden layers and neurons.

3.3 Dataset

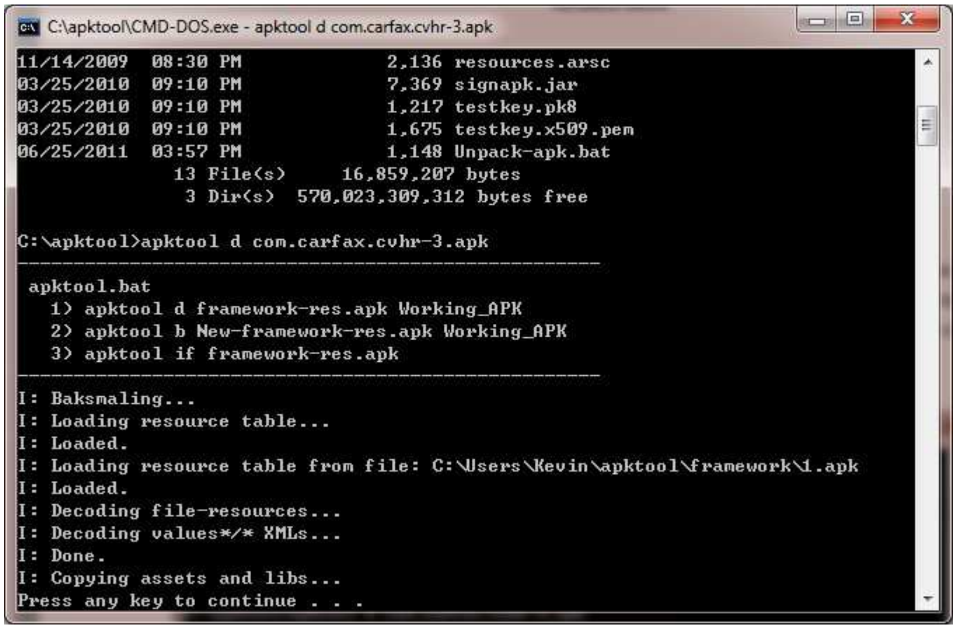
Google Play Market is the most popular online app store, allowing users to discover, install, and buy apps for Android smartphone. Our dataset was extracted from Google Play Market in March 2012. The dataset includes top 50 free apps of 34 different categories. It is divided by two parts: application (26 categories) and game (8 categories). The downloaded apps were saved in the Android Application Package (APK) file type. We were not able to gain direct access to the AndroidManifest.xml, due to the digital signing protection.

To read manifest.xml, a very popular decompile tool called apktool was used. It is a tool for reverse engineering and decoding third party apps to their original deployment form [34]. Figure 3.2 is the screenshot of how to use apktool to decompile app.

Generally, AndroidManifest.xml is located in the root directory of each app. This xml file contains configuration settings of the Android apps. It includes the specific application package name; components of the application, like activities, services, and content providers; and it declares the required permissions provided by developers.

Figure 3.3 is a screenshot example of an application's permission label access control. During install time, apps will request permissions to the device user. In this example, there are five requested permissions at install time: *INTERNET*, *CAMERA*, *VIBRATE*, *FLASHLIGHT*, and *READ_PHONE_STATE*.

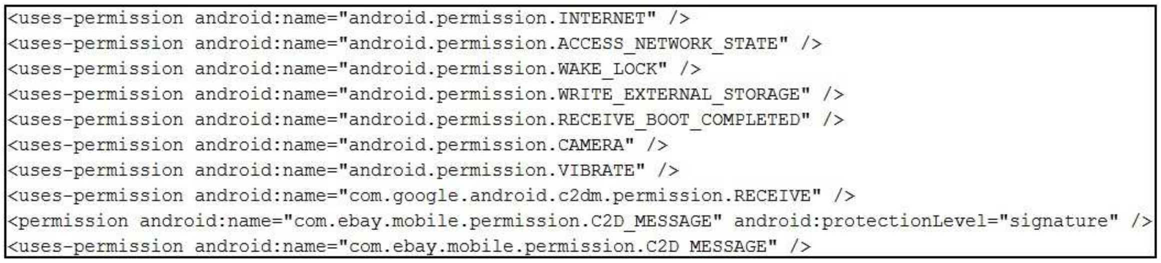
Until March of 2012, the Android consisted of 124 permission labels. For reference purposes, Appendix A lists all of the permission labels and their descriptions. In fact, apps can define their own permissions if they intend for other apps to have programmatic access to them



```
11/14/2009  08:30 PM             2,136 resources.arsc
03/25/2010  09:10 PM             7,369 signapk.jar
03/25/2010  09:10 PM             1,217 testkey.pk8
03/25/2010  09:10 PM             1,675 testkey.x509.pem
06/25/2011  03:57 PM             1,148 Unpack-apk.bat
              13 File(s)     16,859,207 bytes
               3 Dir(s)  570,023,309,312 bytes free

C:\apktool>apktool d com.carfax.cvhr-3.apk
---------------------------------------------------
 apktool.bat
   1) apktool d framework-res.apk Working_APK
   2) apktool b New-framework-res.apk Working_APK
   3) apktool if framework-res.apk
---------------------------------------------------
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Users\Kevin\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
Press any key to continue . . .
```

Figure 3.2: Screen shot of apktool



```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
<permission android:name="com.ebay.mobile.permission.C2D_MESSAGE" android:protectionLevel="signature" />
<uses-permission android:name="com.ebay.mobile.permission.C2D_MESSAGE" />
```

Figure 3.3: Android permissions example

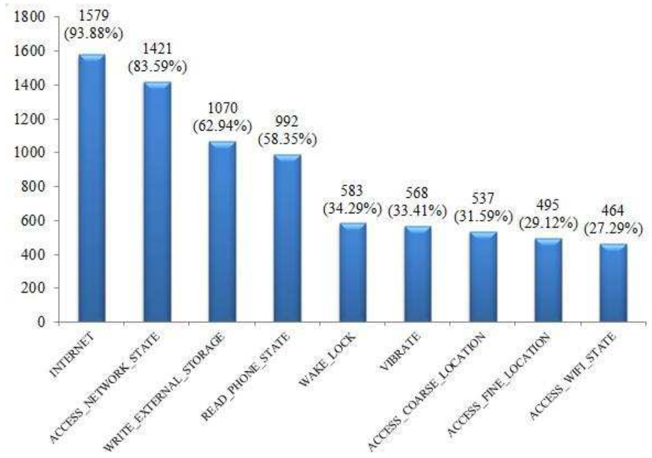


Figure 3.4: Permission request distribution of the dataset

[35]. But these self–defined permissions are not frequently used and have no general rules. In our project, we focus on more generalized permissions which are defined and provided by Google.

The dataset is depicted with binary vectors of their permission labels. In the vectors, '1' shows that current permission is requested, and '0' indicates that permission is not requested. To better evaluate the dataset, statistical analysis of the request permission labels is provided in Table 3.1 . In this table, we recorded all of the categories (34), as well as the number of unique and most frequently appeared permissions in each category.

In the dataset, there are 11,365 permission labels requested and an average of 6.69 permissions requested for each app. The Game Widgets category requested the least (17) distinct permissions, while the App Widgets category requested the most (75) distinct permissions. Furthermore, the NQ Mobile Security and Antivirus app, which belongs to the App Productivity category, requested the most (40) requested permissions overall.

Figure 3.4 and Figure 3.5 show the most used permissions, permission request distribution, and relative permission requests. According to the current data set, nearly 93.88% apps requested *INTERNET* permission, while 1,421 of the 1,700 apps requested *ACCESS_NETWO RK_STATE* permission. Most requested permissions are listed in the descending order: *INTERNET* , *ACCESS_NETWORK_STATE*, *WRITE_EXTERNAL_STORAGE* , *READ_PH ONE_STATE*, *WAKE_LOCK*, *VIBRATE*, *ACCESS_COARSE_LOCATION*, *ACCESS_FIN E_LOCATION* and *ACCESS_WIFI_STATE*. These nine permission labels comprised 68% of the total permission labels; the other 115 permissions comprised the remaining 32% of total permission request.

Table 3.1: Unique Requested Permissions and Most Appeared Apps in Each Category

Type	Category	Unique Permissions	Most Permissions Per app
Apps	Books&Reference	34	13
	Business	57	29
	Comics	19	**8**
	Communication	65	36
	Education	34	11
	Entertainment	25	12
	Finance	36	15
	Health & Fitness	36	18
	Libraries & Demo	46	34
	Lifestyle	47	25
	Live Wallpaper	38	21
	Media & Video	39	14
	Medical	28	16
	Music & Audio	50	19
	News & Magazines	35	14
	Personalization	33	18
	Photography	37	15
	Productivity	71	**40**
	Shopping	33	18
	Social	57	19
	Sports	37	16
	Tools	61	32
	Transportation	34	16
	Travel& Local	37	21
	Weather	38	18
	Widgets	**75**	39
Apps	Arcade & Action	24	15
	Brain & Puzzle	22	9
	Cards & Casino	24	15
	Casual	41	15
	Live Wallpaper	25	13
	Racing	29	13
	Sports Games	30	17
	Widgets & Fitness	**17**	12

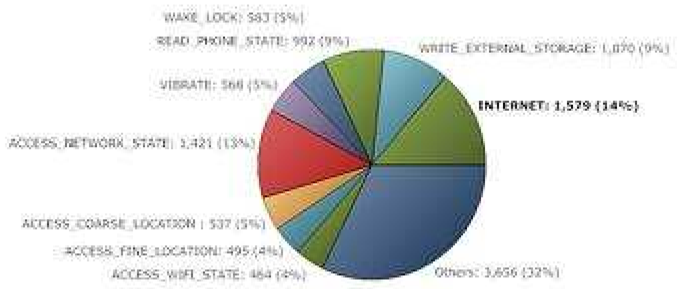


Figure 3.5: Relative permission requests of the dataset

3.4 Experiment Results

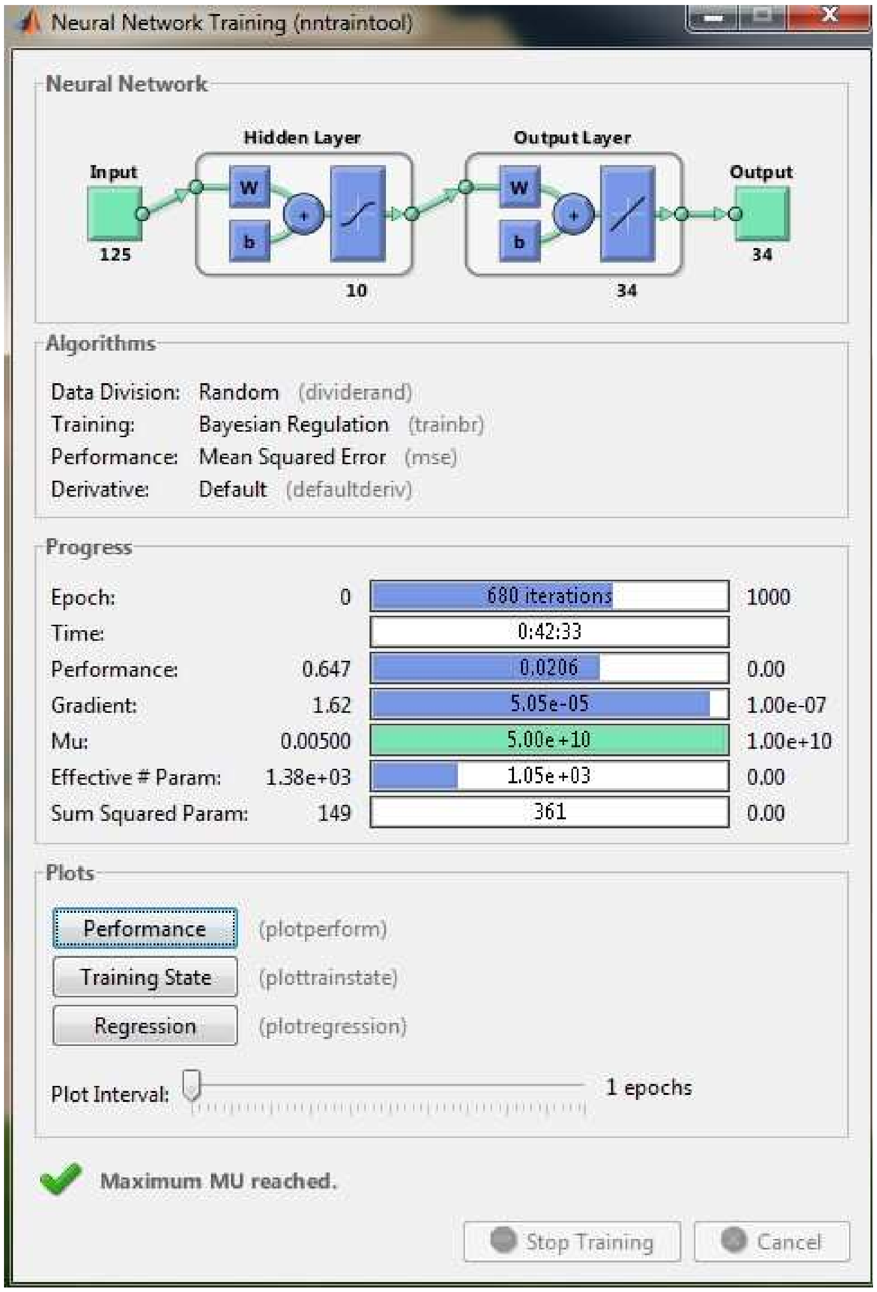
Since we have set up a dataset, we can create, configure, train, and validate the network. In our project, a MLF network is selected for simplicity, a relatively short training time, and efficiency in distinguishing an app's category.

The real dataset is divided into three subsets: training (70%), validation (10%), and testing (20%). The training subset is a set ofconsists of examples for learning where the output value is known; the test subset is used to assess the performance of a classifier and to estimate the error rate; and the validation subset is a setconsists of examples to tune the architecture of a classifier and to calculate the error. Since the test subset is not used in the training stage, the error on the test subset can provide an unbiased estimate of the generalization error.

As for the structure, the hidden layer includes 10 neurons and a sigmoid transfer function, which is depicted in Figure 3.6. The number of neurons is determined by the recommendation of the pattern recognition applications [37] and the concrete practical performance.

In this training process, NN inputs are vectors containing permission labels of apps (124 permission labels) plus one for the correct category. The value of category in the dataset is from 1 to 34. Because because we have 1,700 apps, the dataset size is 1700×125. The output is the 34 dimensional binary vectors: '1' means the permission is included in the training set, and '0' indicates otherwise. Before training a feedforward network, the NN initializes the weights and biases. Once these adjustable parameters are initialized, the network is ready for training. The process of training a NN involves tuning the values of these parameters to optimize network performance. Currently, the optimization methods of MLF use either the gradient of the network performance or the Jacobian of the network errors [27], two methods that are calculated by BP algorithm.

The NN toolbox provides a variety of training algorithms, and these algorithms either use



Algorithms

Data Division:	Random	(dividerand)
Training:	Bayesian Regulation	(trainbr)
Performance:	Mean Squared Error	(mse)
Derivative:	Default	(defaultderiv)

Progress

Epoch:	0	680 iterations	1000
Time:		0:42:33	
Performance:	0.647	0.0206	0.00
Gradient:	1.62	5.05e-05	1.00e-07
Mu:	0.00500	5.00e+10	1.00e+10
Effective # Param:	1.38e+03	1.05e+03	0.00
Sum Squared Param:	149	361	0.00

Plots

Performance	(plotperform)
Training State	(plottrainstate)
Regression	(plotregression)

Plot Interval: 1 epochs

✓ Maximum MU reached.

Figure 3.6: Neural Network Toolbox Training Process Diagram

Table 3.2: Accuracy of Distinct Training Algorithm

Function	Training Algorithm	Accuracy
trainbfg	BFGS Quasi-Newton Back-propagation	0.471
trainbr	Bayesian Regularization	0.606
traincgb	Powell-Beale	0.550
traincgp	Polak-Ribiere	0.341
traingda	Adaptive Learning	0.509
traingdm	Momentum Back-propagation	0.477
traingdx	Adaptive Learning	0.527
trainscg	Scaled Gradient Descent	0.585
trainlm	Levenberg-Marquardt	0.479

gradient-based or Jacobian-based optimization methods. We selected all suitable algorithms for training based on our needs. Different algorithms have different advantages. Table 3.2 shows the accuracy for all selected algorithms: conjugate gradient methods, Levenberg-Margquardt, gradient descent methods, etc. In order to better tune our classifiers among the series type of popular training algorithms, we run different algorithms twice, using our dataset with neuron number 10. In addition, to help us choose the appropriate algorithm, each one in the NN toolbox supports network performance plots and status information.

The Bayesian regularization algorithm results in the highest accuracy (60.6%) and was selected as the experiment's algorithm. Accuracy is the ratio of the number of correct classifications to the total number of test data. It plays a key role in evaluating the performance measure. The efficiency of the Bayesian regularization algorithm is relatively better in constructing and using this model.

The Bayesian regulation algorithm uses the Jacobian for the calculation and has better generalization capabilities than other algorithms. Moreover, it produces accurate results due to good generalization power. In addition, during our test stage, it performed well with noise and missing data. Furthermore, due to its simplicity, it does not require a deep understanding of the NN.

During the training, the output layer produced 34 dimensional binary output vectors (neurons). Each of the output vectors revealed the likelihood that the app belonged to each of the 34 categories. We map the output vector into a binary vector such that the highest likelihood in the output vector is set as '1', while other likelihoods are mapped to '0'. The unity valued element of the binary output vector is the NN's prediction about the concrete category of an app.

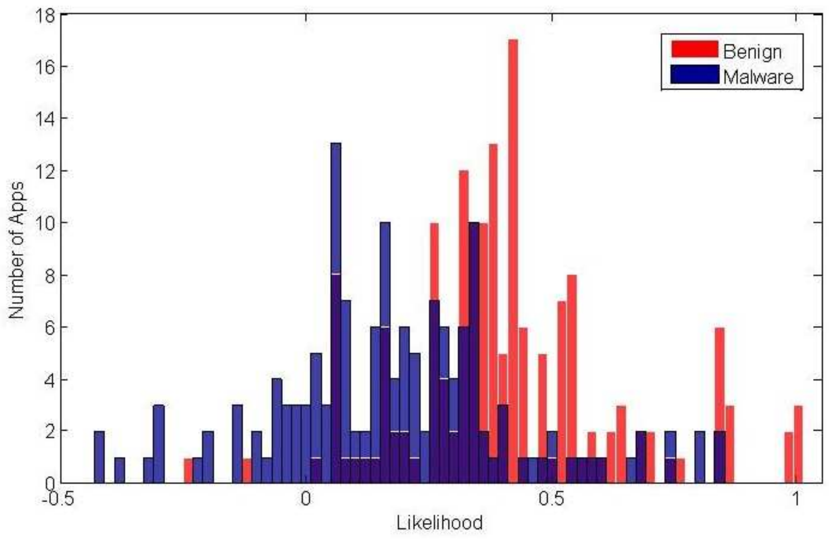


Figure 3.7: Distribution of good and malicious apps

Once a NN is trained, it performs a permission–category clustering when given a testing set. The test dataset contains some good apps and some malware apps. In order to generate malware apps, we randomly permutated the permissions of 50% of the test dataset without changing their current category fields. Therefore, permissions of a genuine app are changed, while its claimed category is unaltered. The test dataset is formed by merging the untouched and manipulated test data. The vector elements attribute categories to apps by estimation, based on the app's permissions revealed during the training phase. Since we know the real categories in the untouched portion of the malware-included dataset, and assuming a manipulated app may not precisely represent its claimed category, we compare the response components with a threshold. If a NN's prediction for an app's category is greater than the threshold, the app is classified as belonging to that category; otherwise, the result is negative.

The trained NN plot displays the empirical distribution of the normal and abnormal apps. Blue curves represent the malicious apps and the red curves represent the benign apps. As shown in Figure 3.7, our approach classifies benign apps and malicious apps relatively well. The best performance can be achieved if an optimal point is used as a threshold for the NN optimal likelihood.

In data mining and pattern recognition, precision, recall, F-score, and accuracy are four important measurements to check the performance. Precision is the fraction of retrieved instances that are relevant, while recall is the faction of relevant instances that are retrieved [38]. Precision is the number of correctly classified positive examples divided by the total number of examples that are classified as positive. In addition, recall is the number of correctly classified positive examples divided by the total number of actual positive examples in the test dataset.

TP (true positive) is the number of correct classification of the positive examples; FP (false positive) is the number of incorrect classification of the negative examples; FN (false negative) is the number of incorrect classification of positive examples; and TN (true negative) is the number of correct classification of negative examples. So, in most cases,

$$Precision = TP/(TP + FP); \quad Recall = TP/(TP + FN). \tag{3.1}$$

A high recall implies that most of the correctly classified results were indeed received. A high precision returns more correct results, as opposed to incorrect ones. F-score is a weighted average (harmonic mean) of the precision and recall, and accuracy is the proportion of true results in a dataset. Accuracy represents the fraction of desirable results.

In our project, TP and FP are predicted outcomes for the benign apps, whereas TN and FP are predicted for malicious apps. The threshold is the minimum value of FPs and FNs; as shown in Figure 3.8, as the threshold increases, the rate of FP decreases and the rate of FN increases. As the threshold decreases, the likelihood that an app is malicious increases; however, if the threshold increases, the accuracy of benign prediction improves. Therefore, we selected the intersection of FP rate (FPR) and FN rate (FNR), which is 0.32 as our threshold to discriminate normal and abnormal apps. The performance will be ensured due to least biased errors.

After we manually choose the threshold, the simulations must be repeated ten times for performance evaluation. In each experiment, we obtain TP, FP, FN, TN, which are then used to calculate the accuracy. Finally, we calculate the average accuracy.

Table 3.3 shows that the final accuracy is approximately 65.1%. The evaluation would be better if we could compare our performance with other experimented results. However, there is no published paper using an approach similar to ours, so we cannot judge the accuracy. Currently, the majority of papers provide specific frameworks to enhance permission policy or filter malicious policy rules. In addition, few published studies include detailed experiment results.

We evaluated these results and sought to improve better performance. We found three possible reasons accounting for the relatively low accuracy observed.

The dataset properties may be responsible for the observed low accuracy. Our dataset is made up of the free downloaded apps from Google Play Market. Among free apps of Google Play Market, some higher-quality apps do not include unnecessary permission labels; how-

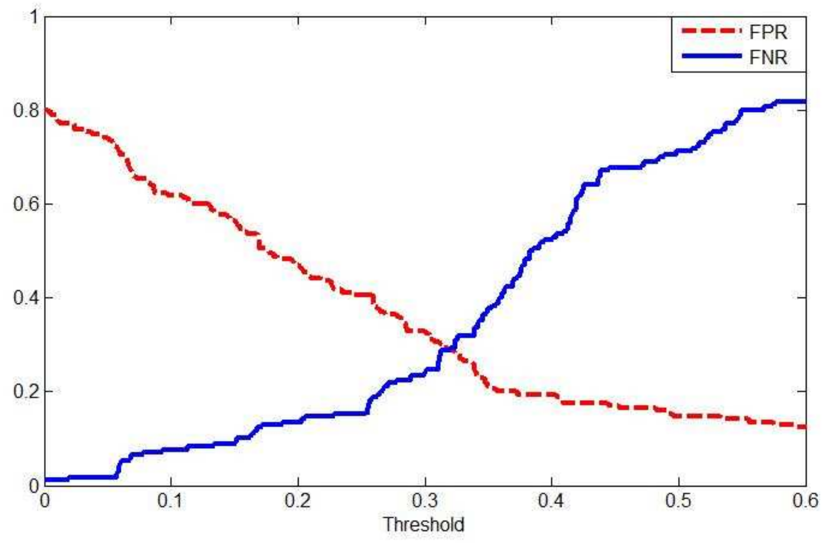


Figure 3.8: FPR / FNR and threshold

Table 3.3: Performance Measurement

No.	TP	FP	FN	TN	Recall	Precision	F	Accuracy
1	116	54	55	115	0.682	0.678	0.680	0.679
2	115	55	55	115	0.676	0.676	0.676	0.676
3	106	64	64	106	0.624	0.624	0.624	0.624
4	109	61	61	109	0.641	0.641	0.641	0.641
5	116	54	54	116	0.682	0.682	0.681	0.682
6	105	65	66	104	0.618	0.614	0.616	0.615
7	114	56	57	113	0.671	0.667	0.667	0.668
8	102	68	69	101	0.600	0.596	0.598	0.597
9	105	65	65	105	0.618	0.618	0.618	0.618
10	121	49	49	121	0.712	0.712	0.712	0.712
AVG					0.652	0.651	0.652	0.651

ever, some apps include some unnecessary permissions added by inexperienced developers. Furthermore, though some very popular and freely downloadable enterprise apps do not access the internet, developers routinely add *INTERNET*, *ACCESS_NETWORK_STATE*, and other permission labels, so that their ads will appear on the screen during usage. As we discussed above, the average number of permissions per app was only 6.69, and nearly one-third of the apps [14] tested included one or more unneeded permissions. In this situation, some permission-overusing apps were combined with genuinely non-overusing apps, thus skewing our performance and reducing the accuracy of the results.

The second reason is that if we randomly change the permissions as malicious app, then we identify the results, in most cases it categories very well. But sometimes it seems too arbitrary. We analyze two possible explanations: firstly, the modification of permissions from malicious to benign, and vice versa, is easily identifiable. Secondly, these exchanged permissions may be non-malicious frequently used permissions, although the likelihood of this scenario is relatively low. The errors produced by second scenario are less than the first; thus, our malicious dataset may be underfitting, leading to lower experimental accuracy.

Also, accuracy is largely dependent on the machine learning algorithm. If the size of the training set increases, the task performance improves consequently.

Considering these data, we concluded that increasing the NN number and layer may improve the performance.

Chapter 4

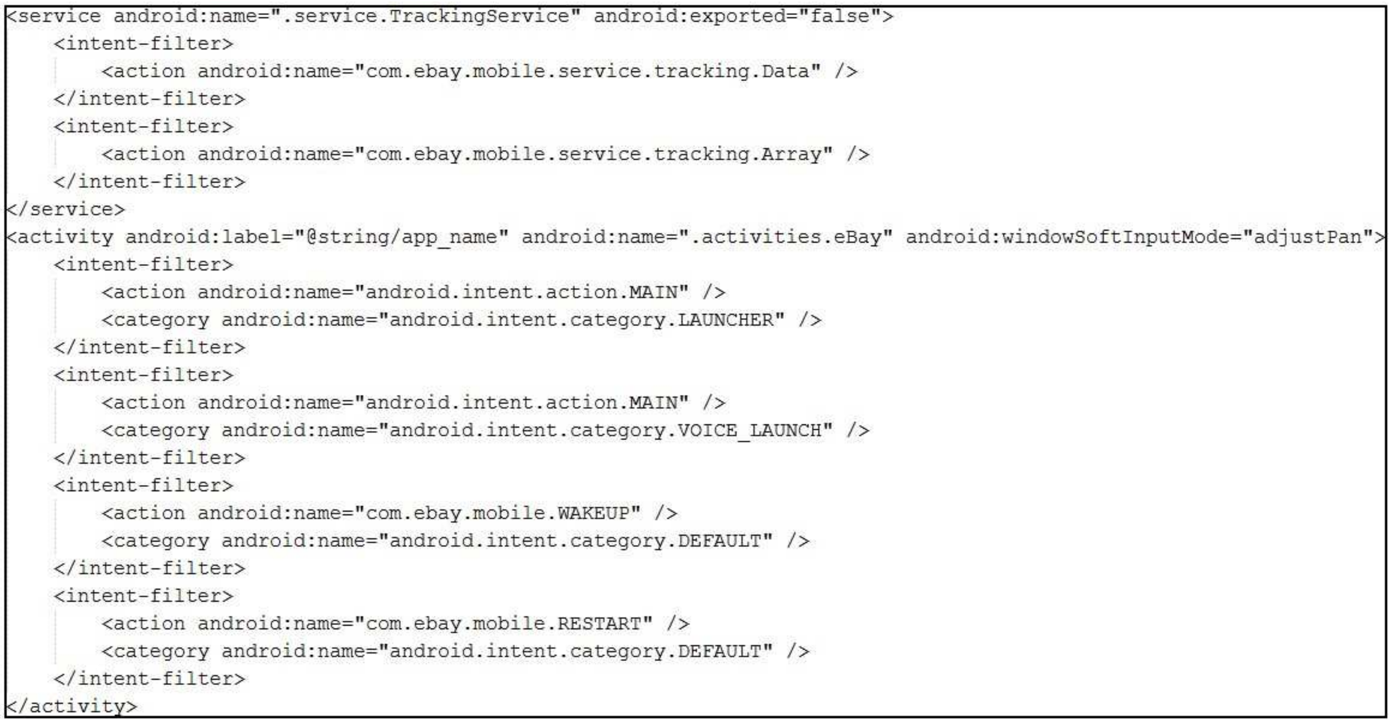
Runtime Malicious Intent Detection using HMM

The Android has a permission security check, which allows or denies the requested Activity Intent action. If the same permissions have been listed in the AndroidManifest.xml, then these activities can be allowed; otherwise, the system will deny the action. Most Intents always combine with permission labels, the keys to ensure security. At install time, we can use category detection to differentiate anomaly apps from good apps. Intent is the "intention" to perform an action, like glue connecting all kinds of activities. At runtime, we can detect malicious patterns based on Intent activities. Since most of the system and Intent action messages are stored in the log file, we can analyze the Intent log files during runtime to recognize malicious patterns.

4.1 Application Tag Components

A quick overview of an app's components is a very beneficial way to understand the relationships between Intent and other functions. In Chapter 1, we described two tags of manifest: use-permission and permission in detail. In this chapter we explore the inside of an app and introduce application tags. Application tags include Activity, Service, Broadcast Receiver, and Content Provider tags.

Activity is the presentation layer for an app, which uses Views to form a graphical user interface (GUI) to display front-end information [39]. A smartphone has limited screen size, and only one or two apps are visibly operated by the user. However, as we know, the Android system can run smoothly on multiple threads. How does one pass functionalities and



```
<service android:name=".service.TrackingService" android:exported="false">
    <intent-filter>
        <action android:name="com.ebay.mobile.service.tracking.Data" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.ebay.mobile.service.tracking.Array" />
    </intent-filter>
</service>
<activity android:label="@string/app_name" android:name=".activities.eBay" android:windowSoftInputMode="adjustPan">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.VOICE_LAUNCH" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.ebay.mobile.WAKEUP" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.ebay.mobile.RESTART" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Figure 4.1: Activity and Service tags

operations without GUI? Service is the answer. Service works in the background without a GUI; it responds to events, updates Intent and Content Provider, and triggers notifications, among other functions. For instance, Service can free the low-priority unused Activity tags and provide threads to apps, which can handle events even if they are inactive. Figure 4.1 shows typical tags of Activity and Service. Activity tags are required for every Activity and are used to specify the class name. Service supports intent-filter subtags to allow late runtime binding.

Receiver tags are abbreviations of a Broadcast Receiver. As a global event listener, if a Receiver obtains one action name with *com.ebay.mobile.RESTART*, the following Intent will execute. From this example, the eBay app will restart.

Provider tags, abbreviations of Content Provider, are used to manage databases and share data within or between apps.

We have discussed Activity, Service, Content Provider and Broadcast Receiver above. Intent is used to combine these to implement various functionalities and actions. It is a message-passing component used to start, stop, and transit activities. Intent carries an asynchronous message, which is used by Android components to perform concrete works. It is a signal to notify the Android system that some event has occurred. In short, Intent is a message passed between activities.

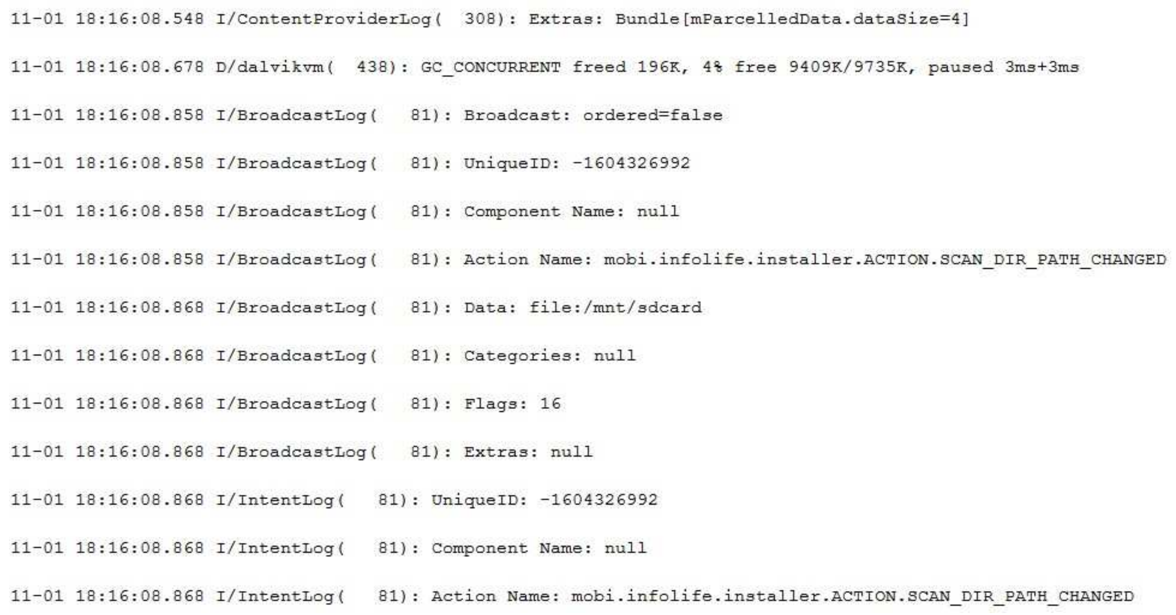
To simplify this, we consider an example of a real app (the Domino Pizza app) to describe the relationship between activity and Intent. A user of the Domino Pizza online via Android app [40] browses the menu screen to choose and purchase pizza. In fact, each pizza icon (activity) is registered by an event listener. When the user clicks the “Customize” button and chooses, say, Mexican Green Wave Pizza on the screen, this click action triggers some method to produce Intent (message) to the system. Then the other screen (activity) will appear. Then the user can choose the size of the pizza, crust and quantity, and other options. After the user clicks the “Next” button, a new Intent is sent to the system, and another popup screen (new activity) is displayed on the screen. Of course, these Intent messages will be stored in the Intent log files.

The Intent contains information that can be passed to the Receive component. If a user sends Intent to the Android, he/she should point out which type of component should be sent. If a user starts an activity or service, he/she uses the *startActivity* or *startService* (Intent) method. After the above activity finishes, the *onActivityResult* method is called. The user or developer can use this method to perform concrete functions.

Intents are divided into two categories: Activity Intent and Broadcast Intent. Activity Intent is used to call Activities outside of an app, since only Activity can solve the Intent. For instance, *Action_Call* performs a call to someone specified by the data. If a user wants to call someone, *Action_Call* must initiate for a phone call to be made. Broadcast Intent is responsible for solving multiple activity requests. For example, if low memory conditions occur due to excess use of resources, *Action_Device_Storage_Low* will begin to manage the memory. At the same time, the Android broadcasts this notification to all other activities. As a result, the affected activities can react accordingly and idle resources can be freed. When the memory condition is restored to a normal level, *Action_Device_Storage_OK* propagates this message to all of the activities.

As Figure 3.3 shows that the *com.ebay.mobile* grants the *CAMERA* permission and the current action is to press the camera button. This calls *Action_Camera_Button*, this action broadcasts at camera initiation message to the Android system. Next, the Android will check the corresponding permission which is the *CAMERA* label. After that Android will assign the permission to *Action_Camera_Button* activity; of course, the Android will permit Intent to launch this activity.

Notification allows the apps to alert users without using an Activity. An Activity uses Intent to let other services or activities communicate with one another. Since Intent broadcasts messages to the system, the other matched app or service can intercept the messages by creating or registering a Broadcast Receiver. A Broadcast Receiver allows those notified apps or services to fire Intent or to trigger events. It can warn users of, or require user attention to,



```
11-01 18:16:08.548 I/ContentProviderLog(  308): Extras: Bundle[mParcelledData.dataSize=4]

11-01 18:16:08.678 D/dalvikvm(  438): GC_CONCURRENT freed 196K, 4% free 9409K/9735K, paused 3ms+3ms

11-01 18:16:08.858 I/BroadcastLog(   81): Broadcast: ordered=false

11-01 18:16:08.858 I/BroadcastLog(   81): UniqueID: -1604326992

11-01 18:16:08.858 I/BroadcastLog(   81): Component Name: null

11-01 18:16:08.858 I/BroadcastLog(   81): Action Name: mobi.infolife.installer.ACTION.SCAN_DIR_PATH_CHANGED

11-01 18:16:08.868 I/BroadcastLog(   81): Data: file:/mnt/sdcard

11-01 18:16:08.868 I/BroadcastLog(   81): Categories: null

11-01 18:16:08.868 I/BroadcastLog(   81): Flags: 16

11-01 18:16:08.868 I/BroadcastLog(   81): Extras: null

11-01 18:16:08.868 I/IntentLog(   81): UniqueID: -1604326992

11-01 18:16:08.868 I/IntentLog(   81): Component Name: null

11-01 18:16:08.868 I/IntentLog(   81): Action Name: mobi.infolife.installer.ACTION.SCAN_DIR_PATH_CHANGED
```

Figure 4.2: Sample output of app Intent Log information

events by creating a new status bar icon, vibrating, sounding alerts, or displaying information on the screen.

4.2 Intent Log components

The Android has a logging facility (log driver) to allow system record logging information of apps and system components. Intent log is used to record all of the Intents. Intent contains information of interest to the requested component and to the Android system [41]. Appendix B lists and describes Intent action names and descriptions. In most cases, Intent fulfills the described functionality. As shown in Figure 4.2, Intent log information consists of user's IDs (UIDs), components, action names, data, categories, flags and extras.

First, *Component* name is recorded in the Intent log. This name is optional. If the Android can find a target component in an app package, the Intent is delivered to some corresponding instance; otherwise, the Android uses other methods to locate this component. Therefore, it can be set as *NULL*, or any other specific name.

Second, available information is *Action* name. At present, Google has defined 140 actions. The system allows a developer to create customized actions. Those actions can be mapped into the AndroidManifest.xml. For our example, *com.ebay.mobile* app defines its own action strings.

Third, *Data* can be concluded from a uniform resource identifier (URI), a string of characters used to identify a web resource. A URI can tell a system and content provider about the data location in a device. Obtaining the information about a data type is beneficial because components capable of displaying pictures should not be called upon to play an audio file [41].

Fourth, Figure 4.1 contains *Category* information. The Intent category is used to describe a target in an Intent-filter. Intent falls into two categories: Explicit Intent and Implicit Intent. Explicit Intent defines a component which is explicitly defined by the Android. Explicit intent has specified a component, whereas implicit Intent has not specified a component. If the system can not ascertain to which category a specific service belongs, the Implicit Intent category is assigned. In the Android, *android.intent.category.LAUNCHER* decides which app or functionality can be listed. *Category* is more like a constraint, managing which activities should be used or not.

Fifth, Intent uses a set of *Flags* to control the behavior of how new activities are invoked [4]. *Flags* are mainly used to control Android launch models.

The last attribute is *Extras*, a bundle of any additional information. It is used to provide extra information about a specific component. For instance, if a user sends an email, the email should include extra data such as subject and body, carbon copy receiver, among other data.

4.3 Hidden Markov Model

Hidden Markov Model (HMM) is a statistical model for classifying a sequence of data. In a Markov model, the states are directly visible to the observer, and the parameter is the state transition probability. For example, the colors of a traffic light are red, green, and yellow. The signals are made up of sequence of traffic lights; each state (yellow color, red color, green color) is only dependent on the previous state. However, in an HMM, the state is not visible (hidden) and each state has a probability distribution over the possible output tokens [42]. This hidden state characteristic is the difference between a HMM and a simple Markov model.

HMM is mainly used in Pattern Recognition like speech, gesture recognition and Bioinformatics. An HMM can be represented as a triple series (π, A, B), where π is the vector of initial state probabilities, A is the state transition matrix, and B is the confusion matrix. Each probability in the state transition matrix and confusion matrix is time-independent.

Multiple series of (π, A, B) represents HMMs with a collection of different hidden states and a sequence of observations. Rabiner [43] presented an HMM description and three basic problems, rendering HMM useful in real-world applications. These three common applications are evaluation, decoding, and learning. For evaluation, given a sequence of observations and a model, the problem is solved by using a forward-backward algorithm, which evaluates the probability of efficient computation. For decoding problems, the Viterbi algorithm, which estimates corresponding optimal state sequences, is used. The learning problem is the most important, and the expectation-maximization (EM) algorithm is most often used. In order to better describe the observations, it optimizes the parameters.

The EM algorithm, introduced by Dempster et al.[44] in 1977, is a very general method, and is primarily used to solve two problems: missing data values due to limitations of the observation process, and maximum likelihood estimation (MLE) problems. Given a set of observed feature vectors in a HMM, using EM algorithm to find the parameters of the MLE is called Baum-Welch algorithm. Baum-Welch (BW) algorithm is the most often used EM algorithm. This algorithm exploits the forward-backward algorithm and computes MLE and posterior mode estimation for the probability parameters in an HMM with known training data [28].

The BW mechanism includes two steps: the E-step and the M-step. The former uses a forward-backward algorithm to estimate the probability of the state sequence when given the observation sequence, while the latter uses the maximum likelihood algorithm to fit new parameters based on the completed data.

Below is the concrete explanation of the BW algorithm. First set $\lambda = (\pi, A, B)$. At the random initial condition, BW algorithm updates λ by repeating iterations until *convergence*. Each of iterations executes E-step and M-step [45] separately. The detail is: given each s and t, it computes $P(S_t = s \mid o_1, o_2, \ldots, o_T)$; then given each s , s' and t, it computes $P(S_t = s, S_(t+1) = s' \mid o_1, o_2, \ldots, o_T)$;

At forward procedure, we initial

$$\alpha_1(i) = \pi_i b_i(O_1), 1 \leqslant i \leqslant N \tag{4.1}$$

Then we calculate recursively as

$$\alpha_{i+1}(j) = b_j(O_{T+1})[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\,],\ 1 \leqslant t \leqslant T-1,\ 1 \leqslant j \leqslant N. \tag{4.2}$$

Finally the iterations terminate when

$$P(O \mid \lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{4.3}$$

At backward procedure, the feature is similar to the forward procedure, but recursive backward instead of forward.

First, we initialize

$$\beta_T(i) = 1,\ 1 \leqslant i \leqslant N \tag{4.4}$$

$$\beta_i(t) = \sum_{j=1}^{N} b_j(O_{t+1})\alpha_{ij}\beta_{i+1}(j) \tag{4.5}$$

Backward processes stop when

$$P(O \mid \lambda) = \sum_{i=1}^{N} \beta_T(i) \tag{4.6}$$

Using α and β, we compute the following variables, we denote $\xi_t(ij)$ represents state $= S_i$ at time t and the probability of state $= S_j$ at time $= t+1$; we denotes $\gamma_i(t)$ as the probability of state $= S_i$ at time t.

$$\xi_t(i,j) = P(Q_t = i,\ Qi+1 = j \mid O, \lambda) = \frac{\alpha_t(i)\beta_i(t)\alpha_{ij}b_j(O_{t+1})}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)\beta_{i+1}(j)\alpha_{ij}b_j(O_{t+1})} \tag{4.7}$$

$$\gamma_i(t) = \sum_{j=1}^{N} \xi_t(i,j) \tag{4.8}$$

After we get ξ and γ, using the auxiliary quantity, an estimated version can be set as below.

$$\pi^* = \gamma_i(t) \tag{4.9}$$

$$\alpha^*_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)} \tag{4.10}$$

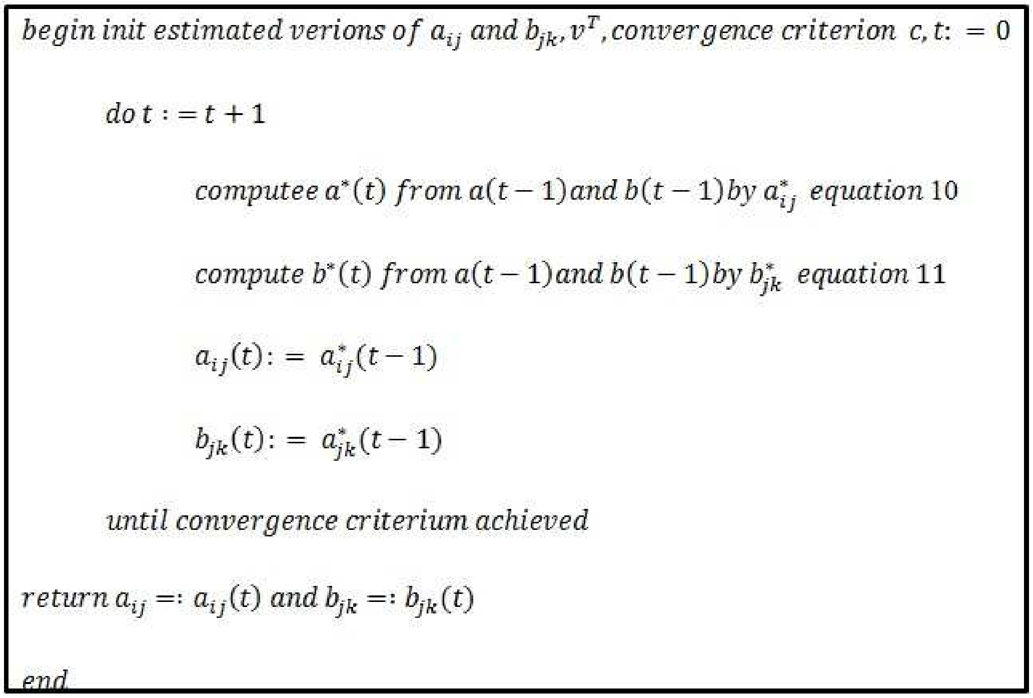
when $(O_t = v_k)$ then

$$b^*_{jk} = \frac{\sum_{t=1}^{T}\sum_l \gamma_{jl}(t)}{\sum_{t=1}^{T}\sum_l \gamma_{jl}(t)} \tag{4.11}$$

The computations iterate very quickly and will stop when the *converge* is less than threshold ϵ.

$$|\log P(O \mid \lambda) - \log P(O \mid \lambda_0)| < \epsilon \tag{4.12}$$

For better understanding a BW algorithm, we list a pseudo code in Figure 4.3. $\alpha^*(t)$ and $b^*(t)$ are the estimated version of the transition and confusion probabilities at time t.



```
begin init estimated verions of $a_{ij}$ and $b_{jk}, v^T$, convergence criterion $c, t: = 0$
    do $t := t + 1$
        computee $a^*(t)$ from $a(t-1)$ and $b(t-1)$ by $a^*_{ij}$ equation 10
        compute $b^*(t)$ from $a(t-1)$ and $b(t-1)$ by $b^*_{jk}$ equation 11
        $a_{ij}(t) := a^*_{ij}(t-1)$
        $b_{jk}(t) := a^*_{jk}(t-1)$
    until convergence criterium achieved
return $a_{ij} =: a_{ij}(t)$ and $b_{jk} =: b_{jk}(t)$
end
```

Figure 4.3: Pseudo code of BW algorithm

HMM can be used to model a system for which a set of observations is provided, and for which the underlying system state is unknown. For Android system calls, we have measured Intent messages from training databases for an HMM, which models Android app behaviors. The reason for choosing BW in our project is that BW is suitable for addressing our problem: how to perform maximum likelihood learning with unknown state information. In addition, the Matlab toolbox recommends the use of the BW algorithm to adjust maximum likelihood parameter estimation.

4.4 Dataset and Experiments

As we discussed above, each Intent log activity includes seven contents: *Unique ID*, *Component* name, *Action* name, *Categories*, *Data*, *flags*, and *extras*. The total usages (activities) of each app can be extracted by parsing the Intent log file. To generate the real usage of Android apps, we may use either Android automated testing tools such as Robotium [46], or manually run apps in the Android emulation environment. In our project, we emulate 529 app usages which are nearly equally distributed in 24 categories.

At the beginning, we generated Intent logs via Robotium. Unfortunately, there were several factors that discouraged us to continue to use Robotium. Generally, different apps categories perform different functionalities. Testing codes are quite different even for apps in the same category, so the testing code reuse is very limited. Another reason is that large number of apps makes it prohibitively time consuming to thoroughly test an app. It is very reasonable to neglect some testing parts if we use automated testing tools without emulating an app completely in advance. Furthermore, half of the apps possess more than ten functionalities, thus writing test code is very slow as compared to manual emulation.

Therefore, we installed all the apps into an Android emulator and ran the apps. Then we intercepted the Intent messages through the Android development tools (ADT) of Eclipse [47]. Eclipse has integrated full-fledged Android environments, including a logging system, whereas smartphone does not provide the mechanism; therefore we used Eclipse as an emulator, as opposed to the Android phone itself.

In our dataset, we use 529 apps: 479 benign apps, and 50 actual malicious apps. The apps were classified into 24 app subcategories (Table 3.1). We omitted two categories (Tools and Wallpaper) because they are not well categorized. To reduce the generalized bias, we excluded the duplicate apps by the same developers which were classified into multiple categories, and chose roughly an equal number of apps from each category.

According to our statistics, for each app, there were, on average, 150 activities which were emulated and saved. Each activity's data contains information from 7 components, plus one unique ID. The current dataset is limited; precision of Machine Learning results increases with dataset size.

Moreover, in the real environment, Android users generally use different apps simultaneously. For instance, a user may run an audio app while playing a card game app. Concurrently, several other apps may be running in the background, e.g., synchronizing data, downloading data, logging events, or surfing the web. A user may use 15-25 apps under normal circumstances.

Based on the above analysis, we used an alternative strategy. We randomly chose 20 apps and mixed their Intent activities (average 150 per app) into one sequence, and sorted activities by the order of their timestamp. i.e.: for each app, activity with newer timestamps was sorted behind activity of older timestamps. Different apps appeared randomly in the Intent log. No other order rules were used among different apps. We generated a long sequence which consisted of an average of 3000 (20 apps$\times$150 activities per app) Intent activities. This sequence is similar to an Android user's normal activity. Using this method, we randomly generated 500 sequences which represented 500 normal Android user sessions.

For the malicious apps dataset, we combined benign and malware apps together. The most popular Android malware—Trojan—always hides inside apps, which run as normal. Trojan usually hibernates and can be triggered only if the Android user performs specific functions. Based on this fact, we randomly choose 15 apps from the benign set and 5 malware as a base dataset; the mixed dataset will emulate the infected app usages as practically as possible.

Besides system-defined actions, nearly 770 developer-defined actions are contained in our dataset. The frequency of these actions varies greatly. From our observations, more than 90% of the actions appear no more than 6 times; almost all other actions appear more than 60 times, and the frequencies from 7 to 50 are very rare. As we know, the number of system-defined actions is 140, and these occupy no more than 8% of the total actions. Therefore, we added 10 additional action slots to these self-defined actions to reduce training errors or other bias. Eight action slots were occupied by high frequency actions; each slot was put into five high-frequency actions. Other relative low frequency actions were assigned two slots. Finally, our actions allocated 150 slots. Since action slots represent observations of HMM, we set 150 as the observation number.

Training and test data were typically 80% and 20% of the dataset, respectively. A training dataset is categorized by two subsets: 400 benign sequences for the benign subset, and 400 for the mixed- malicious subset. Of course, the test dataset included 100 benign sequences for the benign test subset and 100 for the mixed-malicious test subset. The training process can produce two HMMs: one benign pattern, and one malicious pattern. The output of HMMs is a product of each action's probability, and each action's probability is based on its previous states. For example, if the size of the sequence is 1,500, then this sequence can be denoted as $[O_1, O_2, O_3, \ldots, O_{1500}]$ and each observation of O can results in any action label (the total is 150). We put each test data into these two HMMs. The result depends on the probabilities of benign HMM and the probability of malicious HMM. We utilized the winner-take-all mechanism: if the probability of benign is greater than the probability of malicious, the action is categorized as benign; otherwise, as the action is categorized as malicious. The processing diagram is shown in Figure 4.4.

For the learning algorithm, we used the BW algorithm provided by Matlab toolbox. To evaluate the performance of this methodology, we chose precision, recall, F-measure, and accuracy as performance measures.

Before we simulated the experiment, our learning BW algorithm required observation set O and underlying state set Q. Since each action fulfills specific functionalities, we set the action name in our data to represent observation O and Q to represent hidden states of HMM; the output is the probability pattern.

In our simulation, we measured the performance 10 times and results are depicted in Table 4.1 . The average accuracy was 69.7%.

If we increase the number of state parameter Q, the accuracy will also increase. However,

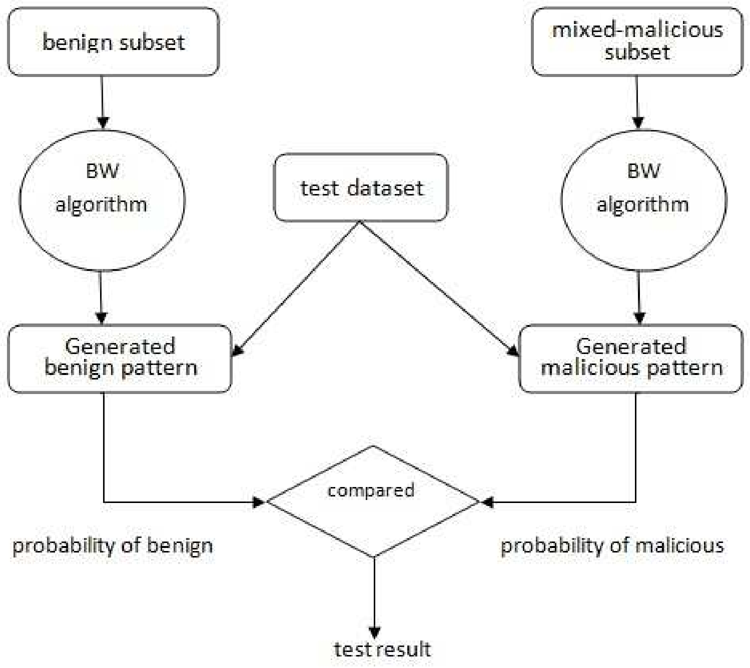


Figure 4.4: training and testing process diagram by using BW algorithm

Table 4.1: Performance Measurement using HMM

No.	TP	FP	FN	TN	Recall	Precision	F	Accuracy
1	78	43	22	57	0.780	0.645	0.706	0.675
2	80	42	20	58	0.800	0.656	0.721	0.690
3	86	41	14	59	0.860	0.677	0.758	0.725
4	78	49	22	51	0.780	0.614	0.687	0.645
5	84	38	16	62	0.840	0.689	0.757	0.730
6	82	40	18	60	0.820	0.672	0.739	0.710
7	82	33	18	67	0.820	0.713	0.763	0.745
8	88	51	12	49	0.880	0.633	0.736	0.685
9	78	43	22	57	0.780	0.645	0.706	0.675
10	88	51	12	49	0.880	0.633	0.736	0.685
AVG					0.880	0.633	0.736	0.685

this incurs an extremely slow execution. Another improved method is to balance the sequence size. Because the current sequence is roughly equal to information from 3,000 Intent activities, the size of the sequence sometimes varies greatly. For this reason, accuracy is reduced. Another possible way to improve the accuracy is to increase the input dataset size.

Chapter 5

Conclusion

In our project, we detect malicious patterns at install time and runtime. At install time, a total of 1,700 apps from 34 categories reveal an increasing trend in app permission request traits. We have introduced a NN approach to validate Android app categories using their permission requests by estimating the app category likelihoods. Our experiments show that Bayesian regularization training, with a category-containing dataset alongside permissions, renders greater accuracy. Furthermore, we leverage a threshold-sweeping approach to jointly minimize FP and FN by intersecting curves for FP and FN rates.

The methodology performance measurement creates the possibility for leveraging NNs for detection of category misrepresentation. The current study assumes permission manipulation, without altering a category, may misrepresent a category; this is not always true. The average number of permissions per app is only 6.69, and about 1/3 of the apps includes one or more unneeded permissions. In this situation, some permission-overusing apps were combined with genuinely non-overusing apps, thus skewing our performance and reducing the accuracy of the results. Another reason is that if we randomly change the permission labels as malicious app, then we identify the results, but sometimes it may be arbitrary. In the first case, if permissions are modified from dangerous to non-dangerous and vice versa, we can identify this very well. In the second case, sometimes these exchanged permissions are all benign, oft-used permissions. The errors produced by the second case are comparably smaller than those in the first case. Thus, our malicious dataset may be underfitting, reducing the experimental accuracy. Furthermore, the accuracy largely depends on the machine learning algorithm. If the size of the training set increases, the task performance subsequently improves. In addition, increasing the NN number and layer may improve the performance.

At runtime, we generated an activities dataset by tracking the output of the Intent log. Each dataset consisted of roughly 150 Intent activities. Our dataset has 529 emulated app usages. To emulate the real usage and generate sufficient data without bias, we randomly chose 20 apps, combined these usages, ordered them by time stamp. Thus, we generated a 3000-activity sequence as a single dataset. For a malicious dataset, we randomly chose 15 benign apps and 5 malicious apps to generate a 3000-activity sequence dataset. Each dataset represents a one-time real usage. We iterated and generated 1,000 one-time usage data sequences.

Based on the data's features and our requirements, we used HMM to train the data and test it. There were two important adjustable parameters: hidden states number and observation number. The observation number is fixed, since it is the size of the *Action* name labels. The number of Action labels is 150, which consists of system-defined Action labels (140) and 10 developer-defined Action labels.

The average accuracy is approximately 69.7% when we set the hidden states at $Q = 50$. If we increase this parameter, the performance of accuracy can increase lightly. However, this incurs an extremely slow execution.

In our project, we present two novel methods to detect malicious apps. Currently, there is no previous research on this topic. Some studies focus on permission policy enforcement or permission validation, while no published studies have examined the relationship between apps and their assigned category. Other studies analyze Intent activities with the help of the Kernel level mechanism. We analyze Intent activities above the Kernel layer and use HMM, a novel mechanism. To some extent, our biggest contribution is providing two original solutions to analyze Android security problems.

Our future work will focus on three aspects. First, we must emulate Game apps to generate game activities. Our current dataset does not include this type of data. Second, we will focus on how to better emulate the user's functions. Most Android users run apps in *Communication*, *Media & Video*, *Game*, *shopping*, and *Photography*; they do not often use apps of *Medical*, *Libraries & Demo*, *Comics*. Therefore, we may not randomly choose 20 apps from all categories; rather, we may assign some Intent activities in advance and then randomly choose. For example, we may pick 3 apps from *Communication*, 4 apps from *Game*, 2 apps from *Social* and 1 app from *Music & Video*, while the remaining apps will be randomly chosen from the rest of categories. Third, we need to balance the size of each dataset. Since the size of data varies greatly, the results are not very stable. The lowest accuracy is 64.5% and highest accuracy is 74.5%. We can create a threshold, such that if a data is too long or too short, we will discard it and generate a new one. In this manner, the output of performance may stabilize and accuracy may improve.

Bibliography

[1] "Gartner says worldwide sales of mobile phones of third quarter of 2012." `http://www.gartner.com/newsroom/id/2408515`, November 2012.

[2] G.Sheran, *Android apps security.* 1st edition ed., 2012.

[3] "Android security overview." `http://source.android.com/tech/security/`, May 2012.

[4] K. Satya and M. Dave, *Pro android 4.* Apress Berkely, 2012.

[5] "Android permission api guides." `http://developer.android.com/guide/topics/manifest/permission-element.html`, May 2012.

[6] "Mobile malware evolution." `http://www.securelist.com/en/analysis?pubid=200119916`, September 2009.

[7] "Only 7 have malware detection rate among 40 android anti-virus apps tested." `http://www.droid-life.com/2012/03/07/`, March 2012.

[8] M. Ghorbanzadeh, Z. Ma, T. C. Clancy, R. McGwier, and Y. Chen, "A neural network approach to category validation of android applications," in *Proceedings of the 2013 International Conference on Computing, Networking and Communications (ICNC)*, ICNC '13, (Washington, DC, USA), pp. 740–744, IEEE Computer Society, 2013.

[9] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 235–245, ACM, 2009.

[10] 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 '10, (New York, NY, USA), pp. 73–84, ACM, 2010.

[11] T. Vidas, N. Christin, and L. Cranor, "Curbing Android permission creep," in *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, (Oakland, CA), May 2011.

[12] A. P. Felt, S. Hanna, E. Chin, H. J. Wang, and E. Moshchuk, "Permission re-delegation: Attacks and defenses," in *In 20th Usenix Security Symposium*, 2011.

[13] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, (New York, NY, USA), pp. 3:1–3:14, ACM, 2012.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011.

[15] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2011.

[16] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, (New York, NY, USA), pp. 13–22, ACM, 2012.

[17] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011.

[18] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short paper: a look at smartphone permission models," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, (New York, NY, USA), pp. 63–68, ACM, 2011.

[19] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, (Washington, DC, USA), pp. 340–349, IEEE Computer Society, 2009.

[20] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, (New York, NY, USA), pp. 328–332, ACM, 2010.

[21] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[22] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[23] "Google play market." `https://play.google.com/store`, March 2012.

[24] "Matlab machine learning." `http://www.mathworks.com/discovery/machine-learning.html`, January 2013.

[25] A. Nigrin, *Neural Networks for Pattern Recognition.* MIT Press, 1993.

[26] "A basic introduction to feedforward backpropagation neural networks." `http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html`, November 2012.

[27] H. Demuth and M. Beale, "Neural network toolbox users guide," The MathWorks, Inc, 2000.

[28] "Baum-welch algorithm." `http://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm`, Oct 2012.

[29] P. Donmez, G. Lebanon, and K. Balasubramanian, "Unsupervised supervised learning i: Estimating classification and regression errors without labels," *J. Mach. Learn. Res.*, vol. 99, pp. 1323–1351, August 2010.

[30] "Introduction to neural networks." `http://itee.uq.edu.au/~cogs2010/cmc/chapters/Introduction/`, January 2013.

[31] T. O. Ayodele, "Types of machine learning algorithms." `http://www.intechopen.com/books/new-advances-in-machine-learning/types-of-machine-learning-algorithms`, November 2012.

[32] "Stanford machine learing." `http://www.holehouse.org/mlclass/07_Regularization.html`, November 2012.

[33] V. K. D. Svozil and J. Pospichal, "Chemon," tech. rep., Intell Lab Syste, 1997.

[34] "Android apktool." `http://code.google.com/p/android-apktool/downloads/list`, January 2012.

[35] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," 2011.

[36] S. Grossberg, ed., *Neural networks and natural intelligence.* Cambridge, MA, USA: Massachusetts Institute of Technology, 1988.

[37] D. MacKay, *Bayesian Interpolation*, vol. 4. Neural Computation, 1992.

[38] "Precision and recall." `http://en.wikipedia.org/wiki/Precision_and_recall`, May 2012.

[39] M. Reto, *Professional android application development.* Wrox, Nov 24,2008.

[40] "Android tutorials for beginners." `http://www.edureka.in/blog/android-tutorials-intent-component/`, January 2013.

[41] "Android developers api guides." `http://developer.android.com/guide/components/intents-filters.html`, August 2012.

[42] "Hidden markov model." `http://en.wikipedia.org/wiki/Hidden_Markov_model`, Oct 2012.

[43] L. R. Rabiner, "Readings in speech recognition," ch. A tutorial on hidden Markov models and selected applications in speech recognition, pp. 267–296, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[44] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, vol. 39, no. 1, pp. 1–38, 1977.

[45] J. Bilmes, "A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models," tech. rep., 1998.

[46] "Android robotium." `https://code.google.com/p/robotium/`, August 2012.

[47] "Adt plugin." `http://developer.android.com/tools/sdk/eclipse-adt.html`, January 2012.

[48] K. Murphy, "Hmm matlab toolbox." `http://www.mathworks.com/matlabcentral/newsreader/view_thread/306248`, April 1998.

Appendix A.

Android Permission Constants

For reference purposes, this appendix provides a complete list of Android permission constants.

Permission Name	Description
ACCESS_CHECKIN_PROPERTIES	Allows read/write access to the "properties" table in the checkin database, to change values that get uploaded.
ACCESS_COARSE_LOCATION	Allows an app to access approximate location derived from network location sources such as cell towers and Wi-Fi.
ACCESS_FINE_LOCATION	Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
ACCESS_LOCATION_EXTRA_COMMANDS	Allows an application to access extra location provider commands.
ACCESS_MOCK_LOCATION	Allows an application to create mock location providers for testing.
ACCESS_NETWORK_STATE	Allows applications to access information about networks.
ACCESS_SURFACE_FLINGER	Allows an application to use SurfaceFlinger's low level features.
ACCESS_WIFI_STATE	Allows applications to access information about Wi-Fi networks.
ACCOUNT_MANAGER	Allows applications to call into AccountAuthenticators.

ADD_VOICEMAIL	Allows an application to add voicemails into the system.
AUTHENTICATE_ACCOUNTS	Allows an application to act as an AccountAuthenticator for the AccountManager.
BATTERY_STATS	Allows an application to collect battery statistics.
BIND_APPWIDGET	Allows an application to tell the AppWidget service which application can access AppWidget's data.
BIND_DEVICE_ADMIN	Must be required by device administration receiver, to ensure that only the system can interact with it.
BIND_INPUT_METHOD	Must be required by an InputMethodService, to ensure that only the system can bind to it.
BIND_REMOTEVIEWS	Must be required by a RemoteViewsService, to ensure that only the system can bind to it.
BIND_TEXT_SERVICE	Must be required by a TextService (e.g.)
BIND_VPN_SERVICE	Must be required by an VpnService, to ensure that only the system can bind to it.
BIND_WALLPAPER	Must be required by a WallpaperService, to ensure that only the system can bind to it.
BLUETOOTH	Allows applications to connect to paired bluetooth devices.
BLUETOOTH_ADMIN	Allows applications to discover and pair bluetooth devices.
BRICK	Required to be able to disable the device (very dangerous!).
BROADCAST_PACKAGE_REMOVED	Allows an application to broadcast a notification that an application package has been removed.
BROADCAST_SMS	Allows an application to broadcast an SMS receipt notification.
BROADCAST_STICKY	Allows an application to broadcast sticky intents.
BROADCAST_WAP_PUSH	Allows an application to broadcast a WAP PUSH receipt notification.
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed.
CALL_PRIVILEGED	Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.

CAMERA	Required to be able to access the camera device.
CHANGE_COMPONENT_ENABLED_STATE	Allows an application to change whether an application component (other than its own) is enabled or not.
CHANGE_CONFIGURATION	Allows an application to modify the current configuration, such as locale.
CHANGE_NETWORK_STATE	Allows applications to change network connectivity state.
CHANGE_WIFI_MULTICAST_STATE	Allows applications to enter Wi-Fi Multicast mode.
CHANGE_WIFI_STATE	Allows applications to change Wi-Fi connectivity state.
CLEAR_APP_CACHE	Allows an application to clear the caches of all installed applications on the device.
CLEAR_APP_USER_DATA	Allows an application to clear user data.
CONTROL_LOCATION_UPDATES	Allows an application to create mock location providers for testing.
DELETE_CACHE_FILES	Allows an application to delete cache files.
DELETE_PACKAGES	Allows an application to delete packages.
DEVICE_POWER	Allows low-level access to power management.
DIAGNOSTIC	Allows applications to RW to diagnostic resources.
DISABLE_KEYGUARD	Allows applications to disable the keyguard.
DUMP	Allows an application to retrieve state dump information from system services.
EXPAND_STATUS_BAR	Allows an application to expand or collapse the status bar.
FACTORY_TEST	Run as a manufacturer test application, running as the root user.
FLASHLIGHT	Allows access to the flashlight.
FORCE_BACK	Allows an application to force a BACK operation on whatever is the top activity.
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service.
GET_PACKAGE_SIZE	Allows an application to find out the space used by any package.
GET_TASKS	Allows an application to get information about the currently or recently running tasks.

GLOBAL_SEARCH	This permission can be used on content providers to allow the global search system to access their data.
HARDWARE_TEST	Allows access to hardware peripherals.
INJECT_EVENTS	Allows an application to inject user events (keys, touch, trackball) into the event stream and deliver them to ANY window.
INSTALL_LOCATION_PROVIDER	Allows an application to install a location provider into the Location Manager.
INSTALL_PACKAGES	Allows an application to install packages.
INTERNAL_SYSTEM_WINDOW	Allows an application to open windows that are for use by parts of the system user interface.
INTERNET	Allows applications to open network sockets.
KILL_BACKGROUND_PROCESSES	Allows an application to callkillBackgroundProcesses(String)
MANAGE_ACCOUNTS	Allows an application to manage the list of accounts in the AccountManager.
MANAGE_APP_TOKENS	Allows an application to manage (create, destroy, Z-order) application tokens in the window manager.
MASTER_CLEAR	Not for use by third-party applications.
MODIFY_AUDIO_SETTINGS	Allows an application to modify global audio settings.
MODIFY_PHONE_STATE	Allows modification of the telephony state - power on, mmi, etc.
MOUNT_FORMAT_FILESYSTEMS	Allows formatting file systems for removable storage.
MOUNT_UNMOUNT_FILESYSTEMS	Allows mounting and unmounting file systems for removable storage.
NFC	Allows applications to perform I/O operations over NFC.
PERSISTENT_ACTIVITY	*This constant was deprecated in API level 9. This functionality will be removed in the future; please do not use. Allow an application to make its activities persistent.*
PROCESS_OUTGOING_CALLS	Allows an application to monitor, modify, or abort outgoing calls.
READ_CALENDAR	Allows an application to read the user's calendar data.

READ_CONTACTS	Allows an application to read the user's contacts data.
READ_FRAME_BUFFER	Allows an application to take screen shots and more generally get access to the frame buffer data.
READ_HISTORY_BOOKMARKS	Allows an application to read (but not write) the user's browsing history and bookmarks.
READ_INPUT_STATE	*This constant was deprecated in API level 16. The API that used this permission has been removed.*
READ_LOGS	Allows an application to read the low-level system log files.
READ_PHONE_STATE	Allows read only access to phone state.
READ_PROFILE	Allows an application to read the user's personal profile data.
READ_SMS	Allows an application to read SMS messages.
READ_SOCIAL_STREAM	Allows an application to read from the user's social stream.
READ_SYNC_SETTINGS	Allows applications to read the sync settings.
READ_SYNC_STATS	Allows applications to read the sync stats.
REBOOT	Required to be able to reboot the device.
RECEIVE_BOOT_COMPLETED	Allows an application to receive theACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.
RECEIVE_MMS	Allows an application to monitor incoming MMS messages, to record or perform processing on them.
RECEIVE_SMS	Allows an application to monitor incoming SMS messages, to record or perform processing on them.
RECEIVE_WAP_PUSH	Allows an application to monitor incoming WAP push messages.
RECORD_AUDIO	Allows an application to record audio.
REORDER_TASKS	Allows an application to change the Z-order of tasks.
RESTART_PACKAGES	*This constant was deprecated in API level 8. TherestartPackage(String) API is no longer supported.*
SEND_SMS	Allows an application to send SMS messages.

SET_ACTIVITY_WATCHER	Allows an application to watch and control how activities are started globally in the system.
SET_ALARM	Allows an application to broadcast an Intent to set an alarm for the user.
SET_ALWAYS_FINISH	Allows an application to control whether activities are immediately finished when put in the background.
SET_ANIMATION_SCALE	Modify the global animation scaling factor.
SET_DEBUG_APP	Configure an application for debugging.
SET_ORIENTATION	Allows low-level access to setting the orientation (actually rotation) of the screen.
SET_POINTER_SPEED	Allows low-level access to setting the pointer speed.
SET_PREFERRED_APPLICATIONS	*This constant was deprecated in API level 7. No longer useful, see addPackageToPreferred(String) for details.*
SET_PROCESS_LIMIT	Allows an application to set the maximum number of (not needed) application processes that can be running.
SET_TIME	Allows applications to set the system time.
SET_TIME_ZONE	Allows applications to set the system time zone.
SET_WALLPAPER	Allows applications to set the wallpaper.
SET_WALLPAPER_HINTS	Allows applications to set the wallpaper hints.
SIGNAL_PERSISTENT_PROCESSES	Allow an application to request that a signal be sent to all persistent processes.
STATUS_BAR	Allows an application to open, close, or disable the status bar and its icons.
SUBSCRIBED_FEEDS_READ	Allows an application to allow access the subscribed feeds ContentProvider.
SUBSCRIBED_FEEDS_WRITE	Allows an application to allow write the subscribed feeds ContentProvider.
SYSTEM_ALERT_WINDOW	Allows an application to open windows using the type TYPE_SYSTEM_ALERT, shown on top of all other applications.
UPDATE_DEVICE_STATS	Allows an application to update device statistics.
USE_CREDENTIALS	Allows an application to request authtokens from the AccountManager.
USE_SIP	Allows an application to use SIP service.
VIBRATE	Allows access to the vibrator.

WAKE_LOCK	Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming.
WRITE_APN_SETTINGS	Allows applications to write the apn settings
WRITE_CALENDAR	Allows an application to write (but not read) the user's calendar data.
WRITE_CONTACTS	Allows an application to write (but not read) the user's contacts data.
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.
WRITE_GSERVICES	Allows an application to modify the Google service map.
WRITE_HISTORY_BOOKMARKS	Allows an application to write (but not read) the user's browsing history and bookmarks.
WRITE_PROFILE	Allows an application to write (but not read) the user's personal profile data.
WRITE_SECURE_SETTINGS	Allows an application to read or write the secure system settings.
WRITE_SETTINGS	Allows an application to read or write the system settings.
WRITE_SMS	Allows an application to write SMS messages.
WRITE_SOCIAL_STREAM	Allows an application to write (but not read) the user's social stream data.
WRITE_SYNC_SETTINGS	Allows applications to write the sync settings.

Appendix B.

Android Action Constants

For reference purposes, this appendix provides a complete list of Android Action constants.

Action Name	Description
ACTION_AIRPLANE_MODE_CHANGED	Broadcast Action: The user has switched the phone into or out of Airplane Mode.
ACTION_ALL_APPS	Activity Action: List all available applications. Input: Nothing.
ACTION_ANSWER	Activity Action: Handle an incoming phone call.
ACTION_APP_ERROR	Activity Action: The user pressed the “Report” button in the crash/ANR dialog.
ACTION_ASSIST	Activity Action: Perform assist action.
ACTION_ATTACH_DATA	Used to indicate that some piece of data should be attached to some other place.
ACTION_BATTERY_CHANGED	Broadcast Action: This is a sticky broadcast containing the charging state, level, and other information about the battery.
ACTION_BATTERY_LOW	Broadcast Action: Indicates low battery condition on the device.
ACTION_BATTERY_OKAY	Broadcast Action: Indicates the battery is now okay after being low.
ACTION_BOOT_COMPLETED	Broadcast Action: This is broadcast once, after the system has finished booting.
ACTION_BUG_REPORT	Activity Action: Show activity for reporting a bug.
ACTION_CALL	Activity Action: Perform a call to someone specified by the data.

ACTION_CALL_BUTTON	Activity Action: The user pressed the “call” button to go to the dialer or other appropriate UI for placing a call.
ACTION_CAMERA_BUTTON	Broadcast Action: The “Camera Button” was pressed.
ACTION_CHOOSER	Activity Action: Display an activity chooser, allowing the user to pick what they want to before proceeding.
ACTION_CLOSE_SYSTEM_DIALOGS	Broadcast Action: This is broadcast when a user action should request a temporary system dialog to dismiss.
ACTION_CONFIGURATION_CHANGED	Broadcast Action: The current device Configuration (orientation, locale, etc) has changed.
ACTION_CREATE_SHORTCUT	Activity Action: Creates a shortcut.
ACTION_DATE_CHANGED	Broadcast Action: The date has changed.
ACTION_DEFAULT	A synonym for ACTION VIEW, the “standard” action that is performed on a piece of data.
ACTION_DELETE	Activity Action: Delete the given data from its container.
ACTION_DEVICE_STORAGE_LOW	Broadcast Action: A sticky broadcast that indicates low memory condition on the device This is a protected intent that can only be sent by the system.
ACTION_DEVICE_STORAGE_OK	Broadcast Action: Indicates low memory condition on the device no longer exists This is a protected intent that can only be sent by the system.
ACTION_DIAL	Activity Action: Dial a number as specified by the data.
ACTION_DOCK_EVENT	Broadcast Action: A sticky broadcast for changes in the physical docking state of the device.
ACTION_DREAMING_STARTED	Broadcast Action: Sent after the system starts dreaming.
ACTION_DREAMING_STOPPED	Broadcast Action: Sent after the system stops dreaming.
ACTION_EDIT	Activity Action: Provide explicit editable access to the given data.
ACTION_EXTERNAL_APPLICATIONS_AVAILABLE	Broadcast Action: Resources for a set of packages (which were previously unavailable) are currently available since the media on which they exist is available.

ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE	Broadcast Action: Resources for a set of packages are currently unavailable since the media on which they exist is unavailable.
ACTION_FACTORY_TEST	Activity Action: Main entry point for factory tests.
ACTION_GET_CONTENT	Activity Action: Allow the user to select a particular kind of data and return it.
ACTION_GTALK_SERVICE_CONNECTED	Broadcast Action: A GTalk connection has been established.
ACTION_GTALK_SERVICE_DISCONNECTED	Broadcast Action: A GTalk connection has been disconnected.
ACTION_HEADSET_PLUG	Broadcast Action: Wired Headset plugged in or unplugged.
ACTION_INPUT_METHOD_CHANGED	Broadcast Action: An input method has been changed.
ACTION_INSERT	Activity Action: Insert an empty item into the given container.
ACTION_INSERT_OR_EDIT	Activity Action: Pick an existing item, or insert a new item, and then edit it.
ACTION_INSTALL_PACKAGE	Activity Action: Launch application installer.
ACTION_LOCALE_CHANGED	Broadcast Action: The current device's locale has changed.
ACTION_MAIN	Activity Action: Start as a main entry point, does not expect to receive data.
ACTION_MANAGE_NETWORK_USAGE	Activity Action: Show settings for managing network data usage of a specific application.
ACTION_MANAGE_PACKAGE_STORAGE	Broadcast Action: Indicates low memory condition notification acknowledged by user and package management should be started.
ACTION_MEDIA_BAD_REMOVAL	Broadcast Action: External media was removed from SD card slot, but mount point was not unmounted.
ACTION_MEDIA_BUTTON	Broadcast Action: The "Media Button" was pressed.
ACTION_MEDIA_CHECKING	Broadcast Action: External media is present, and being disk-checked The path to the mount point for the checking media is contained in the Intent.mData field.
ACTION_MEDIA_EJECT	Broadcast Action: User has expressed the desire to remove the external storage media.

ACTION_MEDIA_MOUNTED	Broadcast Action: External media is present and mounted at its mount point.
ACTION_MEDIA_NOFS	Broadcast Action: External media is present, but is using an incompatible fs (or is blank) The path to the mount point for the checking media is contained in the Intent.mData field.
ACTION_MEDIA_REMOVED	Broadcast Action: External media has been removed.
ACTION_MEDIA_SCANNER_FINISHED	Broadcast Action: The media scanner has finished scanning a directory.
ACTION_MEDIA_SCANNER_SCAN_FILE	Broadcast Action: Request the media scanner to scan a file and add it to the media database.
ACTION_MEDIA_SCANNER_STARTED	Broadcast Action: The media scanner has started scanning a directory.
ACTION_MEDIA_SHARED	Broadcast Action: External media is unmounted because it is being shared via USB mass storage.
ACTION_MEDIA_UNMOUNTABLE	Broadcast Action: External media is present but cannot be mounted.
ACTION_MEDIA_UNMOUNTED	Broadcast Action: External media is present, but not mounted at its mount point.
ACTION_MY_PACKAGE_REPLACED	Broadcast Action: A new version of your application has been installed over an existing one.
ACTION_NEW_OUTGOING_CALL	Broadcast Action: An outgoing call is about to be placed.
ACTION_PACKAGE_ADDED	Broadcast Action: A new application package has been installed on the device.
ACTION_PACKAGE_CHANGED	Broadcast Action: An existing application package has been changed.
ACTION_PACKAGE_DATA_CLEARED	Broadcast Action: The user has cleared the data of a package.
ACTION_PACKAGE_FIRST_LAUNCH	Broadcast Action: Sent to the installer package of an application when that application is first launched (that is the first time it is moved out of the stopped state).
ACTION_PACKAGE_FULLY_REMOVED	Broadcast Action: An existing application package has been completely removed from the device.
ACTION_PACKAGE_INSTALL	*This constant was deprecated in API level 14. This constant has never been used.*
ACTION_PACKAGE_NEEDS_VERIFICATION	Broadcast Action: Sent to the system package verifier when a package needs to be verified.

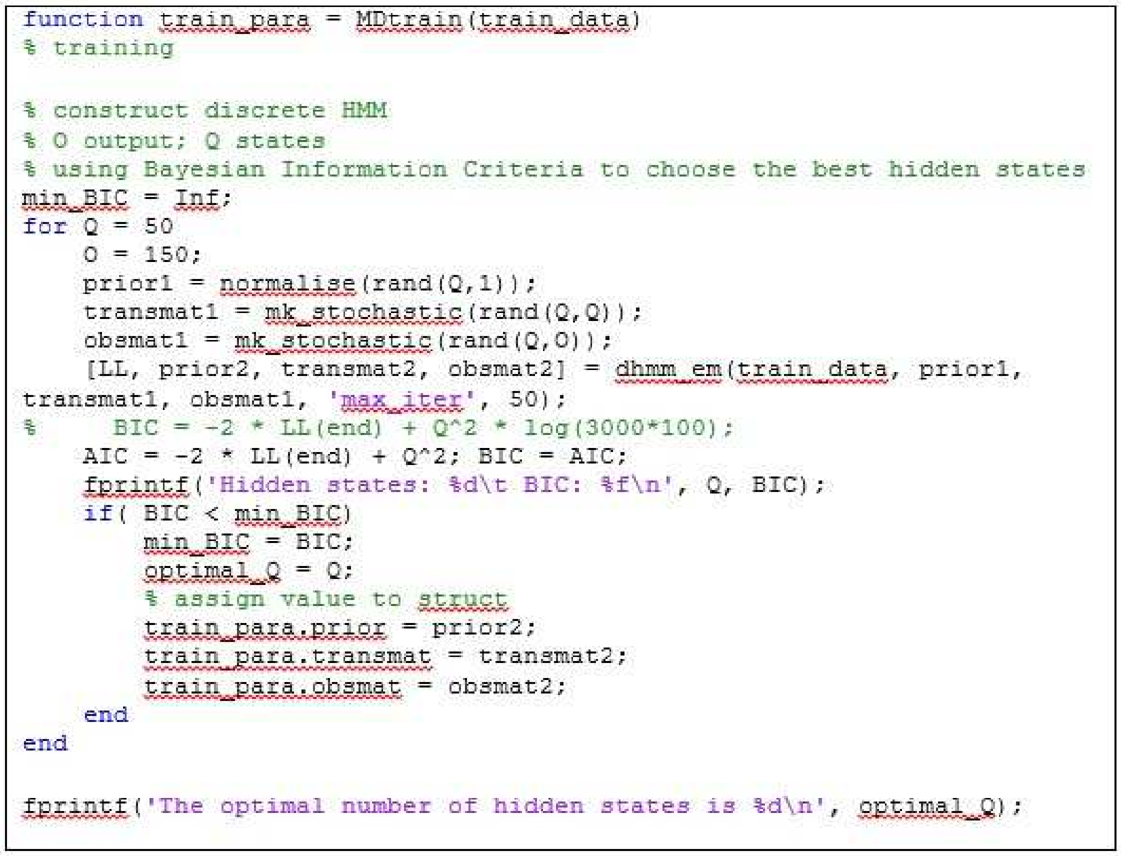
ACTION_PACKAGE_REMOVED	Broadcast Action: An existing application package has been removed from the device.
ACTION_PACKAGE_REPLACED	Broadcast Action: A new version of an application package has been installed, replacing an existing version that was previously installed.
ACTION_PACKAGE_RESTARTED	Broadcast Action: The user has restarted a package, and all of its processes have been killed.
ACTION_PACKAGE_VERIFIED	Broadcast Action: Sent to the system package verifier when a package is verified.
ACTION_PASTE	Activity Action: Create a new item in the given container, initializing it from the current contents of the clipboard.
ACTION_PICK	Activity Action: Pick an item from the data, returning what was selected.
ACTION_PICK_ACTIVITY	Activity Action: Pick an activity given an intent, returning the class selected.
ACTION_POWER_CONNECTED	Broadcast Action: External power has been connected to the device.
ACTION_POWER_DISCONNECTED	Broadcast Action: External power has been removed from the device.
ACTION_POWER_USAGE_SUMMARY	Activity Action: Show power usage information to the user.
ACTION_PROVIDER_CHANGED	Broadcast Action: Some content providers have parts of their namespace where they publish new events or items that the user may be especially interested in.
ACTION_QUICK_CLOCK	Sent when the user taps on the clock widget in the system's "quick settings" area.
ACTION_REBOOT	Broadcast Action: Have the device reboot.
ACTION_RUN	Activity Action: Run the data, whatever that means.
ACTION_SCREEN_OFF	Broadcast Action: Sent after the screen turns off.
ACTION_SCREEN_ON	Broadcast Action: Sent after the screen turns on.
ACTION_SEARCH	Activity Action: Perform a search.
ACTION_SEARCH_LONG_PRESS	Activity Action: Start action associated with long pressing on the search key.
ACTION_SEND	Activity Action: Deliver some data to someone else.
ACTION_SENDTO	Activity Action: Send a message to someone specified by the data.

ACTION_SEND_MULTIPLE	Activity Action: Deliver multiple data to someone else.
ACTION_SET_WALLPAPER	Activity Action: Show settings for choosing wallpaper Input: Nothing.
ACTION_SHUTDOWN	Broadcast Action: Device is shutting down.
ACTION_SYNC	Activity Action: Perform a data synchronization.
ACTION_SYSTEM_TUTORIAL	Activity Action: Start the platform-defined tutorial Input: getStringExtra (SearchManager.QUERY) is the text to search for.
ACTION_TIMEZONE_CHANGED	Broadcast Action: The timezone has changed.
ACTION_TIME_CHANGED	Broadcast Action: The time was set.
ACTION_TIME_TICK	Broadcast Action: The current time has changed.
ACTION_UID_REMOVED	Broadcast Action: A user ID has been removed from the system.
ACTION_UMS_CONNECTED	*This constant was deprecated in API level 14. replaced by android.os.storage.StorageEventListener*
ACTION_UMS_DISCONNECTED	*This constant was deprecated in API level 14. replaced by android.os.storage.StorageEventListener*
ACTION_UNINSTALL_PACKAGE	Activity Action: Launch application uninstaller.
ACTION_USER_BACKGROUND	Sent when a user switch is happening, causing the process's user to be sent to the background.
ACTION_USER_FOREGROUND	Sent when a user switch is happening, causing the process's user to be brought to the foreground.
ACTION_USER_INITIALIZE	Sent the first time a user is starting, to allow system apps to perform one time initialization.
ACTION_USER_PRESENT	Broadcast Action: Sent when the user is present after device wakes up (e.g when the keyguard is gone).
ACTION_VIEW	Activity Action: Display the data to the user.
ACTION_VOICE_COMMAND	Activity Action: Start Voice Command.
ACTION_WALLPAPER_CHANGED	*This constant was deprecated in API level 16. Modern applications should use WindowManager. LayoutParams.FLAG SHOW WALLPAPER to have the wallpaper shown behind their UI, rather than watching for this broadcast and rendering the wallpaper on their own.*
ACTION_WEB_SEARCH	Activity Action: Perform a web search.

Appendix C.

Matlab code for BW algorithm

For reference purposes, this appendix provides Matlab code for BW algorithm; we reference the original code from Kevin Murphy's toolbox [48].



```
function train_para = MDtrain(train_data)
% training

% construct discrete HMM
% O output; Q states
% using Bayesian Information Criteria to choose the best hidden states
min_BIC = Inf;
for Q = 50
    O = 150;
    prior1 = normalise(rand(Q,1));
    transmat1 = mk_stochastic(rand(Q,Q));
    obsmat1 = mk_stochastic(rand(Q,O));
    [LL, prior2, transmat2, obsmat2] = dhmm_em(train_data, prior1,
transmat1, obsmat1, 'max_iter', 50);
%     BIC = -2 * LL(end) + Q^2 * log(3000*100);
    AIC = -2 * LL(end) + Q^2; BIC = AIC;
    fprintf('Hidden states: %d\t BIC: %f\n', Q, BIC);
    if( BIC < min_BIC)
        min_BIC = BIC;
        optimal_Q = Q;
        % assign value to struct
        train_para.prior = prior2;
        train_para.transmat = transmat2;
        train_para.obsmat = obsmat2;
    end
end

fprintf('The optimal number of hidden states is %d\n', optimal_Q);
```