BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Behavioral Malware Detection Approaches for Android

A Thesis submitted to the

DEPT. OF COMPUTER SCIENCE AND ENGINEERING

in partial fulfillment of the requirements for the
Degree of

BACHELOR OF SCIENCE IN ENGINEERING

By

Mehedee Zaman, Mohammad Rakib Amin, Tazrian Siddiqui
Dhaka, Bangladesh
2015

Behavioral Malware Detection Approaches for Android

This thesis is prepared by

Mehedee Zaman

Mohammad Rakib Amin

Tazrian Siddiqui

Under the supervision of

Dr. M Shohrab Hossain
Associate Professor,
Department of Computer Science and Engineering
Bangladesh University of Engineering & Technology
Dhaka 1000, Bangladesh

Acknowledgements

This work has been possible because of a number of individuals. First of all, We would like to express our sincere gratitude to our thesis supervisor Dr. M Shohrab Hossain for his expert advice, patience and continuous supervision throughout our Undergraduate Thesis, which made this work possible.

We are also thankful to all the faculty members of the Department of Computer Science at Bangladesh University of Engineering and Technology for providing their valuable comments on our work during different poster presentations.

The authors would like to acknowledge the technical support of Samsung Innovation Lab financed by HEQEP Project of University Grant Commission, for the android devices used in the research.

We would finally like to thank our parents and friends for providing moral support during the thesis work.

Table of Contents

List of Tables

List of Figures

Abstract

Android, the fastest growing mobile operating system began its journey with the release of the Android beta in November 2007. At the moment this mobile OS boasts of a staggering 1 Billion users, who without a careful monitoring of their in-device-security, are susceptible to malicious applications hacking into their personal data. There are plenty repositories that are full of free and cracked versions of premium applications and repackaged applications, majority of which are infected with malicious behaviors. People, being always eager to use free contents, are often deliberately putting themselves in danger of data-hacking and other harmful services through downloading these malwares.

Our goal, therefore, was to investigate the nature and identity of a malicious application and devise a detection procedure based on them. We concentrated on a behavioral analysis of malwares, focusing on their identifiable traits and data-flows in the Android system. Our first approach was network-based, we captured the outgoing data packets and analyzed their source and destination, thereby filtering any malicious domain servers or repositories. Our second step was to identify certain system calls and their frequency in a malicious application to establish a threshold that measures the acceptability of a random application. In both cases, we ran our experiments on 1260 malwares, acquired from Android Malware Genome Project, a malware database created by Y. Zhou et al. [6] and 227 non-malware benign applications. Both procedures are described in this dissertation along with their corresponding results. Our hope is the analysis given here will provide security professionals with more definitive and quantitative approaches in their investigations of mobile malwares on Android system.

Chapter 1

Introduction

1.1 Introduction

Smart phones and tablets are the most popular and widely used personal electronics devices today. According to a statistics, Android dominated the market of these devices with an 82.8% share in 2015 Q2 [1]. Due to Androids vast user base, open nature and relatively less restrictive application distribution system, it has always been an attractive platform for malwares. According to a recent report published jointly by Kaspersky Labs and INTERPOL [2], 20% of devices that uses their software were attacked at least once by malware. So, it is an extreme need to develop an efficient malware detection system in Android. So, it is an extreme need to develop an efficient malware detection system in Android.

Existing detection methods can be classified into two major categories: static (code analysis) and dynamic (runtime/behavioral analysis). The sneakiest malware are almost impossible to detect using static analysis, because they often obfuscate the malicious code using random keys. Some malware download the malicious code at runtime and remove it after execution [6]. In these cases, a code analysis for known malware signature cannot detect the malware.

These exists a few static and dynamic malware detection methods in the literature. Chandramohan et al. [5] has given a high-level overview of various detection methods. Zhou et al. [6] collected, classified and published a large collection of 1260 Android malware. We used malware samples from their collection to evaluate our detection method. Isohara et al. [10] demonstrated a system-call logging based method.

In this paper, we demonstrate two detection methods for finding malware in Android. The first one is based on network traffic analysis. The method we described will be effective against malware that communicates with known malicious remote servers. The second one is based on system call analysis.

1.2 Motivation and Problem Statement

Malware is a program which disrupts computer operation, gather sensitive information or gain access to private systems without users consent. With the ever increasing use of mobile devices, mobile malware pose a significant threat because these devices store contacts, bank account numbers, credit / debit numbers, private photos, messages and a lot of other sensitive information that can be leaked. The Kaspersky Lab study Financial Cyberthreats in 2014 reports that the number of financial malware attacks against Android users grew by 3.25 times in 2014. During 2014, Kaspersky Labs Android products blocked a total of 2,317,194 financial attacks against 775,887 users around the world. The lions share of these (2,217,979 attacks against 750,327 users) used Trojan-SMS malware, and the rest (99,215 attacks against 59,200 users) used Trojan-Banker malware [3]. Given the tremendous growth of Android malware, there is a pressing need for effective malware detection methods.

In this thesis, we defined our problem as follows- Given a random android application, the goal is to find a concrete and definitive method to test that app's behavior against our findings on 1260 malwares and 227 "good" applications, and classify that app as “good” (non-malware) or “bad” (malware). As there are negligible structural difference between a malware and a normal app, we concentrated on behavioral analysis of a given app. Therefore, our thesis is titled “Behavioral Malware Detection Approach for Android”.

1.3 Objectives

Our *proposed* objectives were as follows-

- Study of Malware, definition and evolution and understanding their behavior, thereby identifying their unique signature.
- Study of existing detection methods for malwares, understanding differences between different methods of detection, understanding limitations and discussing possibilities/impossibilities of implementation.
- Derive a detection method for malware in Android.
- Run simulation on test data.
- Document all Findings and Results.

Our *performed* objectives are as follows-

- Initial study of Malware, definition and evolution and understanding their behavior has been performed, a summary of which is illustrated in Chapter 2.
- Study of existing detection methods for malwares has been conducted.
- A detection method has been proposed, consisting of two approaches Network-based (Chapter 3) and System-call based (Chapter 4 & 5), along with their results (Chapter 6).
- The experimental findings are listed and evaluated using various performance metrics in Chapter 6.

1.4 Contributions

- Our network traffic analysis is based on logging the URLs of all remote locations that are contacted by applications for a specific period of time. Given, we have a database of known malicious domains; the applications that contact any of those malicious domains can be flagged as malware.

- On the other hand, in our system call analysis, if we can log all system calls made by an application, we can try to use it on known malwares to find patterns in sequence of system calls. These signatures can be used to detect new applications infected by known malwares. For logging system calls, we use strace, a standard unix tool.

- We described our detection method in a detailed step-by-step manner, mentioning all the necessary tools and techniques used. Also, we briefly explained the purpose behind each step. This paper can be used as a technical guideline by researchers, who are trying to develop network traffic-based or system call based malware detection applications.

1.5 Organization of the Thesis

From the next chapter this literature takes more focused demonstration of our work. The organization is as follows -

In the second chapter of our paper, we analyze some literature that are related to our works. We give a summary of them along with their positive outcome and limitations.

In the third chapter, we start to demonstrate our malware detection method by illustrating network traffic analysis, our first approach.

In the fourth chapter, we will discuss the overall strategy of our second approach for malware detection which uses system call traces of applications to predict malicious activity.

In the fifth chapter, we will go into details on the experiment we conducted to validate our model given in the previous chapter.

In the sixth chapter, we summarize the result obtained from our given model with illustrations.

Finally, in the seventh chapter of this thesis, we draw conclusion and discuss possible future applications of this work.

There are two appendices included in the thesis. Appendix A provides a glossary to some technical words used in this literature, Appendix B serves all the example codes used at different stages of our thesis.

Chapter 2

Literature Survey

In this chapter we have summarized all the related works. At first, we briefly discussed a survey on mobile malwares and their generic detection methods [4]. Then another survey with detection methods divided into two categoriesm static and dynamic [5], followed by a time-line analysis and characterization of 1260 malware dataset [6] are summarized. We then proceed on to more focused approaches like network analysis [7], aggregated system call events [8] and finally system call patterns in repackaged applications [9]. Our problem domain identification and detection approaches are indebted to these mentioned researches. Each research are presented in following subsection, and in the next subsection, a comparison-based literature is provided, posing all differences between all these related works and our approach.

2.1 Briefly described related works

2.1.1 Mobile Malware Evolution, Detection and Defense [4]

- The Research briefly summarizes the history of mobile malware, specifics of mobile security when compared with computer security, various attack vector and attack models, various detection techniques for specific mobile devices, the defense mechanisms to control mobile malware.

- *Strength of the work:*

  1. Works as a baseline for related research.

  2. Clearly distinguishes among various malicious applications (e.g grey-wares etc.)

  3. Defines various detection approaches.

  4. Brings up newer models of malware posting like mal-advertising etc. to focus

- *Limitations:* It doesn't specify a definitive approach, rather generic.

2.1.2 Detection of Mobile Malware in the Wild [5]

- In this paper a survey of techniques that are used to detect mobile malware in the wild is presented and the limitations of current techniques are discussed.

- *Strength of the work:*

  1. Successfully classifies some malwares according to their behavior.

  2. Clearly outlines static and dynamic analysis.

  3. Suggests cloud based detection, resulting in off-device analysis for battery-life efficiency.

- *Limitations:* Suggests that permission analysis can be used for pre-screening, where in the android system, without giving permission, an application cannot be installed to check its behavior.

2.1.3 Dissecting Android Malware: Characterization and Evolution [6]

- There are three goals and contributions of this paper. First, this paper presents the first large collection of 1260 Android malware samples (that we used in our research, see Acknowledgement) in 49 different malware families, which covers the majority of existing Android malware, ranging from August 2010 to October 2011. Second, based on the collected malware samples, it performs a timeline analysis of their discovery and characterize them based on their detailed behavior breakdown, including the installation, activation, and payloads. Third, it performs an evolution-based study of representative Android malware, which shows that they are rapidly evolving and existing anti-malware solutions are seriously lagging behind.

- *Strength of the work:*
  1. Successful creation of data tables for malwares clearly stating their repository (official market/free repository); classification based on installation,activation, malicious payloads, permission issues; evolution of specific malwares through time (since they're discovered)
  2. Evaluation of performance for 4 known antivirus on Android phone.

- *Limitations:* Permission comparison among different apps (malwares and non-malwares) which is inadequate to form definitive outcome, as we have learnt in our research.

2.1.4 Detecting Android Malware on Network Level [7]

- The papers analysis of packet traces focuses on finding information leakage in HTTP traces and identifying connection attempts to command-and-control servers. Conversions containing International Mobile Equipment Identity number (IMEI), phone number or credit card information were tracked. If no abnormalities are detected so far, the packet dump is compared manually to a dump generated by the uninfected VM template image to determine whether the sample was not detected or simply inactive.

- *Strength of the work:*

  1. Blacklisting DNS servers for possible malicious repository.

  2. String matching on HTTP header flags, GET, POST requests for possible malicious data transfer.

  3. The researchers were able to observe client-side communication using mock DNS and HTTP server responses. In total, 18 samples were investigated, generating traces compared against the patterns of identifying information. Of those, 8 samples were detected, 2 evaded detection and 8 samples failed to execute in the virtualized environment.

- *Limitations:*

  1. Addresses cannot be black-listed until a malicious application is identified and the connections it makes are analyzed.

  2. Use of Android x86 virtual machines also introduces several limitations. An Android x86 VM is not a cellphone, it does not support text messages, and has a non-standard IMEI and IMSI. Requests of IMEI and IMSI return the null value.

2.1.5 Detection of Malicious Android Mobile Applications Based on Aggregated System Call Events [8]

- The research suggests a method to distinguish Android-based malicious apps based on the system call event pattern internally activated after running suspicious malicious applications. It analyzed the malicious system call event pattern selected from Android Malware Genome Project. The actual system call patterns are extracted from the normal and malicious apps on Android-based mobile devices. And then, feature events were aggregated to calculate a similarity analysis between normal and malicious event set.

- *Strength of the work:*

  1. Found pattern of normal and malicious system call events.
  2. classifies apps based on malicious behavior based on system call events.
  3. makes activity pattern comparison between normal and malicious apps.
  4. Established Quantifiable similarity among malware examples.

- *Limitations:*

  Identifies 17 system call events which do not occur in normal application, are found in malicious applications. Therefore, any given apps could be suspected as malicious mobile application if the 17 kinds of system call events above have occurred simultaneously in the application. But in our research we found out that there is at most 2/3 system calls that can correspond to such decision.

2.1.6 Identifying android malicious repackaged applications by thread-grained system call sequences [9]

- Based on Malicious Repackaged Applications (MRAs), this work proposes a mechanism SCSdroid (System Call Sequence Droid), which adopts the thread-grained system call sequences activated by applications. The concept is that even if MRAs can be camouflaged as benign applications, their malicious behavior would still appear in the system call sequences. SCSdroid extracts the truly malicious common subsequences from the system call sequences of MRAs belonging to the same family. Therefore, these extracted common subsequences can be used to identify any evaluated application without requiring the original benign application.

- *Strength of the work:* SCSdroid achieved up to 95.97% detection accuracy, i.e., 143 correct detections among 149 applications.
- *Limitations:* No significant limitations.

2.2 Difference between our approach and existing ones

The aforementioned researches focuses on various malware detection attempts. Here follows a comparison of those approaches outlining difference between ours and those.

- The First two research [4] and [5], were targeted on defining malwares and classification of their detection methods, where our work avoids such attempts and was targeted on refining detection methods and implementing them.
- The research work by Y. Zhou et al. [6] was focused on detecting malwares by identifying specific system calls (They devised 17 such), whereas our work proposes system call frequencies to detect malicious behavior in an application.
- The work that follows after [7] represents first definitive approach among we have analyzed with its focus on network level analysis. Our work uses a similar but more specific approach like packet capture and filtering destination for black-listed domains.
- The last two works put their concentration on aggregated system call events [8] and system call patterns in repackaged applications [9], respectively. In contrast, our work tries to form a concrete and generalized approach by establishing a threshold for acceptance based on "system call frequency".

In short, there are notable differences among the presented works before and in the next chapter, our first approach, the network based analysis is illustrated outlining our work to its full proportion.

Chapter 3

Detection using network traffic analysis

In this chapter, we will discuss about malware detection using Network traffic analysis. This chapter contains 2 sections. Section 3.1 describes the outline of our strategy, and in section 3.2, we describe the procedure in details.

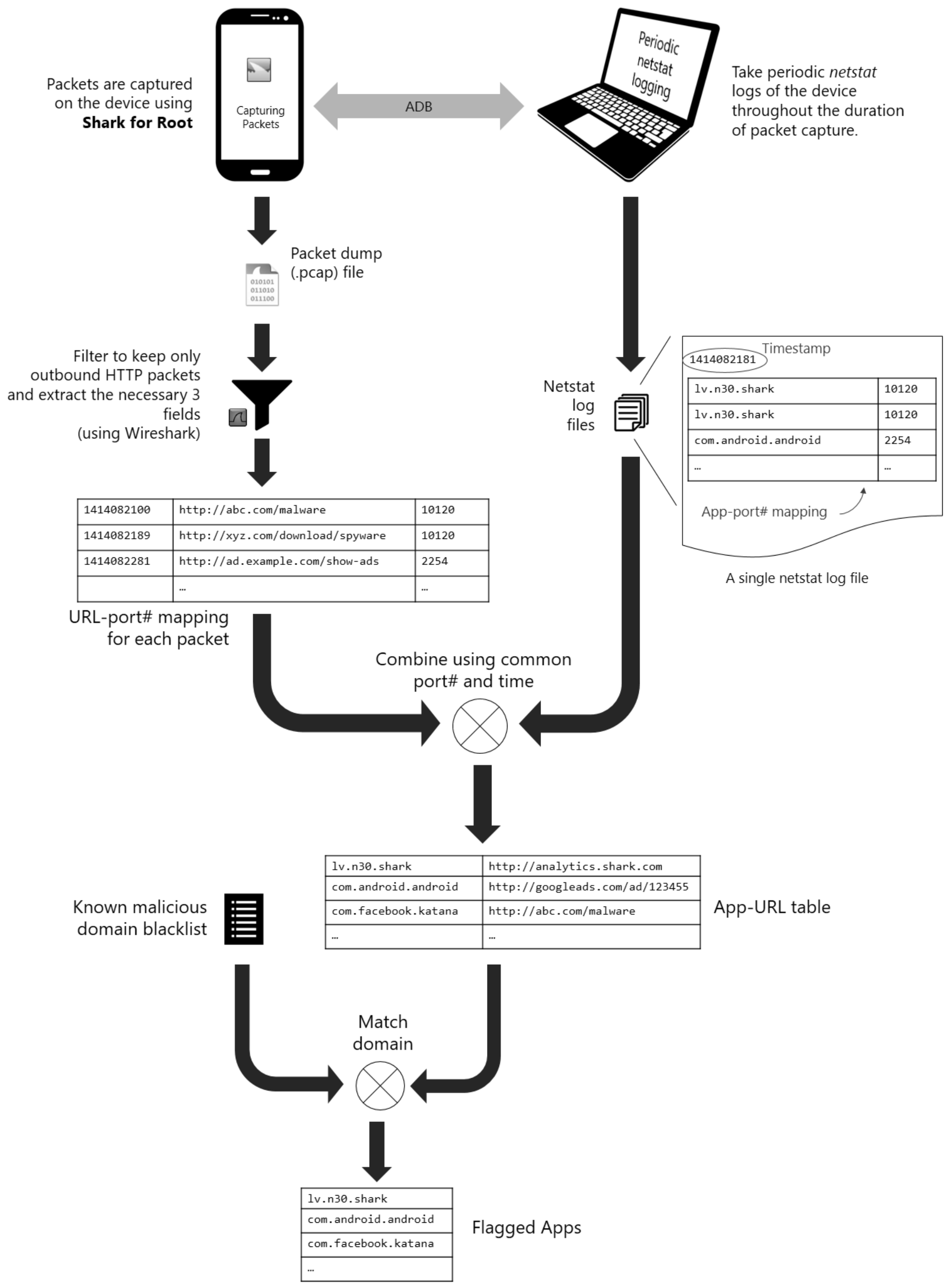
3.1 Strategy of Malware Detection

In this section, we described an outline of our strategy in brief. The whole process in shown as a flowchart in figure 3.1.

At first, we created log of URLs that are contacted by applications for a specific period of time. Then we tried to match each entry (URL) of the log with a list of known malicious domains. If a match is found, the application that contacted the malicious domain is a malware itself or has been affected by one.

3.1.1 Creating the App-URL table

App-URL table is a history/log of all attempts made by all applications to communicate with remote servers over HTTP. The table consists of **(url, app)** entries. Each HTTP request maps to a single entry, where **url** is the URL which is contacted, and


Packets are captured on the device using **Shark for Root**
Capturing Packets
ADB
Periodic netstat logging
Take periodic *netstat* logs of the device throughout the duration of packet capture.
Packet dump (.pcap) file
010101 011010 011100
Filter to keep only outbound HTTP packets and extract the necessary 3 fields (using Wireshark)
Netstat log files
Timestamp
App-port# mapping
A single netstat log file
URL-port# mapping for each packet
Combine using common port# and time
App-URL table
Known malicious domain blacklist
Match domain
Flagged Apps

1414082181

lv.n30.shark	10120
lv.n30.shark	10120
com.android.android	2254
…	…

1414082100	http://abc.com/malware	10120
1414082189	http://xyz.com/download/spyware	10120
1414082281	http://ad.example.com/show-ads	2254
	…	…

lv.n30.shark	http://analytics.shark.com
com.android.android	http://googleads.com/ad/123455
com.facebook.katana	http://abc.com/malware
…	…

lv.n30.shark
com.android.android
com.facebook.katana
…

Figure 3.1: Flowchart of detection strategy using network traffic analysis

app is the application that originated the HTTP request.

This process is further subdivided into four tasks:

3.1.1.1 Packet dumping

We have recorded all incoming and outgoing network packets to/from the android device for specific duration of time. This creates a packet dump file that contains information of which port number (of the mobile device) is accessing which URL.

3.1.1.2 Netstat Logging

To relate port numbers with applications, we periodically executed *netstat* [11] command throughout the duration of packet dumping and saved the outputs. Netstat gives information of which port number is being used by which application when the command is executed.

3.1.1.3 Extracting necessary information from packet dump

We do not take all packets into consideration. We are only interested in HTTP packets (and only requests, not responses). So we have filtered out all other packets from the packet dump we generated at the first step. We took only three fields from each packet: time, originating port and full request URI. This gives a time-sequenced log of port numbers and URIs that a port tried to connect to.

3.1.1.4 Aggregating packet dump and netstat logs

We have so far obtained two separate mappings: *application vs. port number* from netstat logs, and *port number vs. URL* from packet dump. We aggregate these two maps to create a time-sequenced log of applications and the URLs each application tried to contact (The App-URL table).

3.1.2 Matching the URLs with Domain-blacklists

We search the URLs in the App-URL table for known malicious domains. If an application tries to connect to a rogue domain (URL), we flag it as a malware. We can also enrich our blacklist by adding other domains contacted by a flagged application.

These steps are discussed in detail in the following section.

3.2 Details of Malware Detection steps

Our first step is to create an App-URL table. In this table, each row of the table indicates an attempt to make an HTTP connection by any application. We store the time, the application's unique identifier (package name), and the URL which was contacted.

3.2.1 Creating the App-URL table

3.2.1.1 Packet dumping

We need to use a software for recording all incoming or outgoing traffic (packets) of the android device. This can be done using *Wireshark* [12] in a computer which is connected to the same local network of the android device.

Alternatively, we can use a similar application in the mobile device. We have used *Shark for Root* [13] for this purpose. A rooted device is not required for this step. Non-rooted devices can use other applications, such as *tPacketCapture*, which captures packets by creating a VPN and directing all traffic through the VPN. We captured packets for a specific amount of time. This step produces a packet dump (*.pcap*) file.

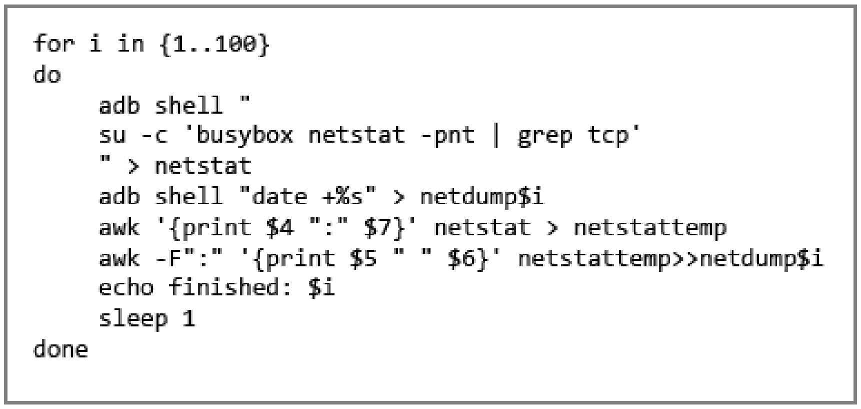
3.2.1.2 Netstat Logging

The packet dump does not directly detect which packet is originated from/destined for which mobile application. The system differentiates packets of different applications by port numbers (source port for outgoing packets or destination port for incoming packets). Hence, we need to know which ports were being used by which applications when the packet was captured. We used the UNIX tool *netstat* [11] to get the mapping between applications and port numbers at a specific time.

Since the packets are recorded for some duration of time and netstat gives the *port number vs. application* mapping for an instance of time (just when the command is executed), a single netstat output will not suffice. Therefore, we executed netstat periodically, while the packets were being recorded.

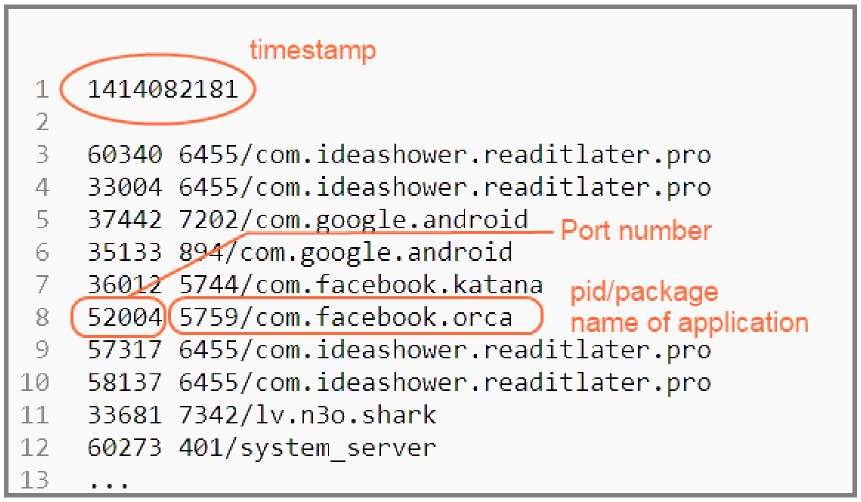
We used *ADB* to communicate with the android device. To access the interactive shell of the device, *adb shell* was used. In our experiment, we connected the android device with a UNIX computer. Then we executed the shell script shown in Fig. 3.2 in the computer.

This script calls netstat 100 times, with 1 second interval in between. It filters just the necessary information (port numbers and corresponding pid/package



```
for i in {1..100}
do
     adb shell "
     su -c 'busybox netstat -pnt | grep tcp'
     " > netstat
     adb shell "date +%s" > netdump$i
     awk '{print $4 ":" $7}' netstat > netstattemp
     awk -F":" '{print $5 " " $6}' netstattemp>>netdump$i
     echo finished: $i
     sleep 1
done
```

Figure 3.2: Shell script used for netstat logging.



```
1414082181

60340 6455/com.ideashower.readitlater.pro
33004 6455/com.ideashower.readitlater.pro
37442 7202/com.google.android
35133 894/com.google.android
36012 5744/com.facebook.katana
52004 5759/com.facebook.orca
57317 6455/com.ideashower.readitlater.pro
58137 6455/com.ideashower.readitlater.pro
33681 7342/lv.n3o.shark
60273 401/system_server
...
```

Figure 3.3: A single netdump file

names) from each netstat output, and saves them in separate files, along with the timestamp when the dump was taken. So after executing this script, we had 100 files (namely `netdump1, netdump2, ... netdump100`). A single netdump file is shown in Fig. 3.3.

This step requires a rooted android device. Because, being a stripped down variant of linux, Android does not come with the *netstat* executable by default. So we used *Busybox*, a tool that allows execution of all standard UNIX commands in android. Busybox cannot be installed without super user permissions.

3.2.1.3 Extracting necessary information from packet dump

Packet dump (.pcap) contains comprehensive meta information about all packets, along with their contents. However, we are only interested in HTTP packets and only three fields of each packet. Pcap filtering can be accomplished by many different ways among which we used Wireshark.

We opened the pcap file in Wireshark. Then the following display filter was applied on the dump:

```
http && ip.src == X.X.X.X
```

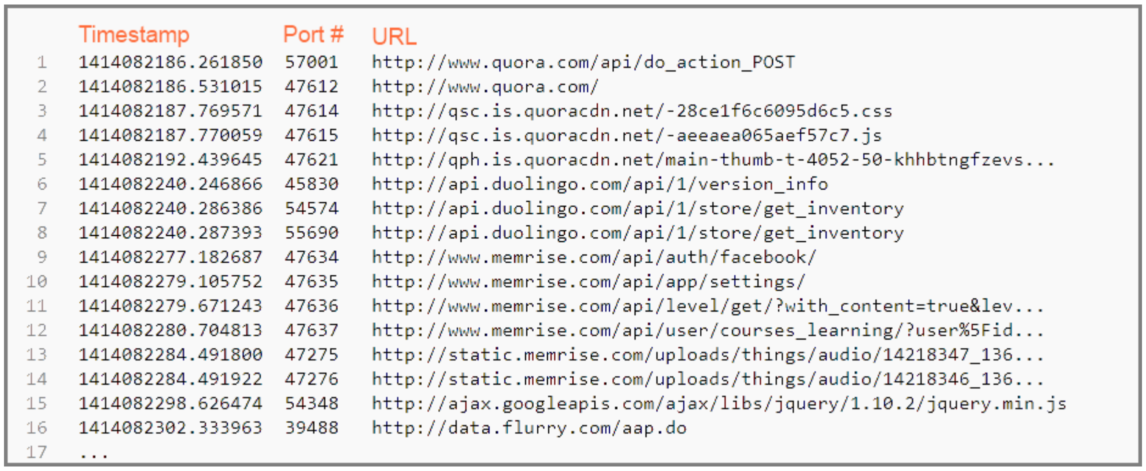
Here, `X.X.X.X` is the IP address of the device. This was used to filter out the http responses. For now, we are only interested in requests.
We kept only the following columns in Wireshark:

- Time (in Seconds since epoch format)
- Src Port
- Full Request URI

Then we exported the displayed packets summary in a plain text file. In our experiment, we named the file `filtered.txt` (shown in Fig. 3.4).

3.2.1.4 Aggregating packet dump and netstat logs

Before this step, we had 100 files containing netstat outputs (*port-application* mapping at specific times). And we had a file filtered.txt, which contains the *port-URL* mapping for all HTTP request packets. We have written a script which processes



```
Timestamp          Port #  URL
1414082186.261850  57001   http://www.quora.com/api/do_action_POST
1414082186.531015  47612   http://www.quora.com/
1414082187.769571  47614   http://qsc.is.quoracdn.net/-28ce1f6c6095d6c5.css
1414082187.770059  47615   http://qsc.is.quoracdn.net/-aeeaea065aef57c7.js
1414082192.439645  47621   http://qph.is.quoracdn.net/main-thumb-t-4052-50-khhbtngfzevs...
1414082240.246866  45830   http://api.duolingo.com/api/1/version_info
1414082240.286386  54574   http://api.duolingo.com/api/1/store/get_inventory
1414082240.287393  55690   http://api.duolingo.com/api/1/store/get_inventory
1414082277.182687  47634   http://www.memrise.com/api/auth/facebook/
1414082279.105752  47635   http://www.memrise.com/api/app/settings/
1414082279.671243  47636   http://www.memrise.com/api/level/get/?with_content=true&lev...
1414082280.704813  47637   http://www.memrise.com/api/user/courses_learning/?user%5Fid...
1414082284.491800  47275   http://static.memrise.com/uploads/things/audio/14218347_136...
1414082284.491922  47276   http://static.memrise.com/uploads/things/audio/14218346_136...
1414082298.626474  54348   http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js
1414082302.333963  39488   http://data.flurry.com/aap.do
...
```

Figure 3.4: Extracted information from the packet dump in `filtered.txt` file

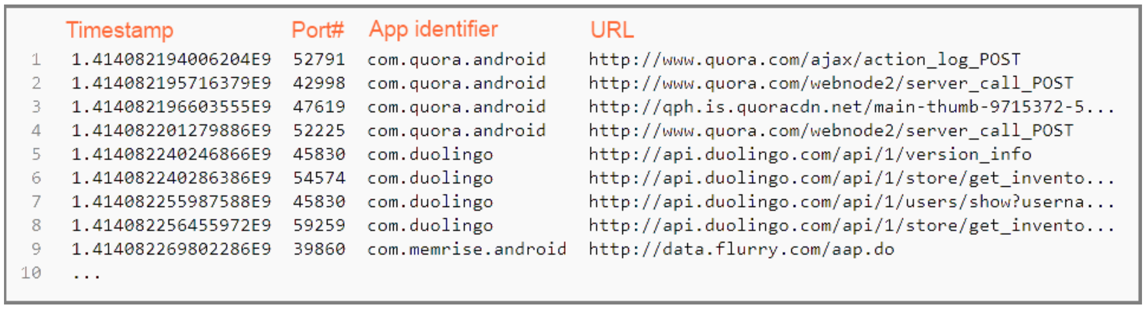
all these files to produce the final App-URL table.

Since netdump files contains *port-app* mappings for specific moments (1 second apart), a packet's time will not necessarily match exactly with any of these moments. To assign such a packet to an application, we have made some assumptions.

Let t be the timestamp of a packet. Let t_1, t_2, $t_3, ..., t_{100}$ are the timestamps of the netstat outputs (they are stored in corresponding netdump files). Of course $t_1 < t_1 < t_3 < ... < t_{100}$. If $t < t_1$ or $t > t_{100}$, we discard the packet. We only consider packets with t such that $t_1 \leq t \leq t_{100}$.

Now for each of these packets, there is an i such that $t_i \leq t$ and $t_{i+1} > t$. We assign a packet to an application using the following rules:

1. If the same application A was using the packet's port at both t_i and t_{i+1}, then application A is the sender of the packet.
2. If application A was using the port at t_i, and the port was not in use at t_{i+1}, application A originated the packet.



Timestamp	Port#	App identifier	URL
1.414082194006204E9	52791	com.quora.android	http://www.quora.com/ajax/action_log_POST
1.414082195716379E9	42998	com.quora.android	http://www.quora.com/webnode2/server_call_POST
1.414082196603555E9	47619	com.quora.android	http://qph.is.quoracdn.net/main-thumb-9715372-5...
1.414082201279886E9	52225	com.quora.android	http://www.quora.com/webnode2/server_call_POST
1.414082240246866E9	45830	com.duolingo	http://api.duolingo.com/api/1/version_info
1.414082240286386E9	54574	com.duolingo	http://api.duolingo.com/api/1/store/get_invento...
1.414082255987588E9	45830	com.duolingo	http://api.duolingo.com/api/1/users/show?userna...
1.414082256455972E9	59259	com.duolingo	http://api.duolingo.com/api/1/store/get_invento...
1.414082269802286E9	39860	com.memrise.android	http://data.flurry.com/aap.do
...			

Figure 3.5: Final output: App vs. URL table

3. If the port was not in use at t_i, and application A was holding it at t_{i+1}, application A originated the packet.

4. If the port was being used by application A at t_i and application B at t_{i+1} then,

   if $t - t_i \leq t_{i+1} - t$, application A originated the packet. Otherwise application B originated it.

5. If no application was using the port at either t_i or t_{i+1}, We discard the packet.

Case 5 indicates that after t_i, some application opened the port, sent some packet(s) and then released the port before t_{i+1}. So this packet has gone untraced. We can lessen the frequency of such occurrences by decreasing the interval between t_i and t_{i+1}.

So for every packet (except the ones of case 5), we know the app which originated it. And `filtered.txt` contains full request URI of all packets. So we now know the URL specified in the packet was contacted by this application. We have logged these (Application, URL) entries for each packet and the App-URL table is ready. A sample table is shown in Fig. 3.5.

3.2.2 Matching the URLs with Domain-blacklists

When the App-URL table is ready, the table can be sent to a central server. The server can search the table for already known malicious domains, and notify the android device of any rogue application which might be trying to connect to a blacklisted domain. The server can also enhance its blacklist by adding new domains that are contacted by a malicious application.

3.3 Results

To validate this approach, we analyzed a total of 28 applications using this method (14 known malwares and 14 known non-malwares). Among them, all of the 14 actual non-malware were correctly classified as non malwares (**True negatives**). None of the actual non-malware were mistaken as malware (**False positives**). However, 9 of the actual 14 malwares were correctly flagged as malwares (**True positives**), while the other 5 were mistakenly classified as non-malwares (**False negatives**). This result is shown in the form of a **Confusion Matrix** in table 3.1.

9 True positives	0 False positives
5 False negatives	14 True negatives

Table 3.1: Confusion Matrix obtained from experimental detection using network analysis

We calculated the following metrics to evaluate the performance of our detection approach.

1. **Accuracy**

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN} = 82.14\%$$

2. **Recall**

$$\rho = \frac{TP}{TP + FN} = 60.8\%$$

3. **Specificity**

$$\sigma = \frac{TN}{TN + FP} = 100\%$$

4. **Precision**

$$\phi = \frac{TP}{TP + FP} = 100\%$$

5. **F-measure**

$$F = 2 \times \frac{\phi \times \rho}{\phi + \rho} = 75.6$$

where TP, TN, FP and FN are number of true positives, true negatives, false positives and false negatives respectively.

Chapter 4

System call based detection

In this chapter, the overall strategy of our second approach for malware detection is discussed, which uses system call traces of applications to predict malicious activity. The next chapter (chapter 5) discusses the implementation details of this model. The results achieved in the experiment using this model is detailed and explained in chapter 6.

The central idea is to run an application for a specific amount of time. During its execution, the details about the system calls it makes to the operating system are recorded. We developed a machine learning model/classifier that can detect malware based on its system call trace. The syscall records of known malwares and known non-malwares are used to train the classifier. There are two phases: training and classification. How the known malware and non-malware traces will be used to train the classifier is described in section 4.1. Section 4.2 decribes how the model will classify an unknown app using its syscall trace.

4.1 Training

We used a set of applications consisting both known malwares and known non-malwares as the training dataset. We collected system call traces of all applications of the training dataset (all of the applications are run for a specific amount of time). The system call trace of a single application is a list of system calls the application used during execution. For example: **{recv, semget, msgget, . . . }**; where **recv**, **semget**, **msgget** are system calls.

After collecting system call traces, We aggregate this traces to create two binary relation matrices M_{mal} and M_{nmal}. M_{mal} shows relation between system calls and malware applications, Where M_{nmal} shows relationship between system calls and non-malwares. M_{mal} and M_{nmal} matrices are defined as follows:

$$M_{mal}(i,j) = \begin{cases} 1 & \text{if } i^{th} \text{ malware uses } j^{th} \text{ syscall} \\ 0 & \text{otherwise} \end{cases}$$

$$M_{nmal}(i,j) = \begin{cases} 1 & \text{if } i^{th} \text{ non-malware uses } j^{th} \text{ syscall} \\ 0 & \text{otherwise} \end{cases}$$

Then we calculate the Goodness Rating of j^{th} syscall, γ_j as follows,

$$\gamma_j = \frac{1}{N_{nmal}} \sum_{i=1}^{N_{nmal}} M_{nmal}(i,j) - \frac{1}{N_{mal}} \sum_{i=1}^{N_{mal}} M_{mal}(i,j)$$

where N_{nmal} and N_{mal} are number of non-malware and malware samples.

4.2 Classification

To classify an unknown application as *malware* or *non-malware*, first, we execute the application for the same time duration duration we used with each training application. We collect the system call trace of that application during that execution, same as before. But this time, we also record the frequency of each syscall used by the application during execution. So now, the syscall trace of an application during classification phase can be expressed as a list of pairs of syscalls and their frequencies. For example: **{(recv,1032), (semget, 143), . . . }** is a trace of an application which called the **recv** routine 1032 times, **semget** 143 times and so on.

Then we define the Goodness Rating of that application, γ_{app} as follows,

$$\gamma_{app} = \sum_{s \in S_{app}} \gamma_s \times f_s$$

Where, S_{app} is the set of system calls used by *app*, γ_s is the Goodness rating of syscall s and f_s is the frequency of syscall s in *app*.

If we assume that malwares uses similar system calls which are distinctive from those used by non-malwares; It is logical to assume that a malware will use more syscalls those has lower goodness ratings and less syscalls having higher goodness ratings. The opposite can be said for non-malware applications. So this will result in higher goodness ratings of non-malware applications and lower goodness rating for malware applications.

Now for classification, we check if the goodness rating of the application under inspection exceeds some threshold. If so, we classify it as non-malware. Otherwise we flag it as malware.

$$\begin{cases} \text{app is a malware} & \text{if } \gamma_{app} > \tau \\ \text{app is not a malware} & \text{otherwise} \end{cases}$$

Where, τ is a threshold. Theoretically, the threshold should be zero. But it actually depends on the experiment and the training data used. We used this approach to classify apps in our validation dataset and calculated **accuracy** (α), **recall** (ρ), **specificity** (σ), **precision** (ϕ) and **f-measure** (F). These metrics are defined in section 3.3.

Chapter 5

Experiment on System call based classifier

In this chapter, we went into details on the experiment we conducted to validate our model.

5.1 Preparing experiment

We collected system call traces of all applications using a single device. The reason behind this is we intended to provide identical environments for all applications to execute in. The device was reset to factory default configuration and the device needed to be *rooted*. We used standard linux utility *strace* to trace system call of applications. We also used *timeout* command to run every application for a fixed duration of time. Although *strace* and *timeout* are standard linux utilities, they are not included in standard Android builds. So we had to collect the source code of this tools and cross-compile them for the CPU architecture of the device on which the experiment was run. The compilation task required *Android NDK*. After the binaries are created for our desired CPU architecture (in our case **ARMv7**), they are put in the **/system/xbin** of our device, so that they can be accessed by a shell script run through *ADB*. Copying any binary into **/system** requires superuser permission, that is one of the reasons why we needed to *root* our device at the first place. Additionally, necessary drivers and Android SDK are required on the host machine, where the script would run.

5.2 Experiment

We planned to collect system call traces of a total of 453 malwares and 227 non-malwares. The number of apps used in validation and training is shown in table 5.1.

	Training	Validation	Total
Malware	403	50	453
Non-malware	177	50	227

Table 5.1: Number of apps used in the experiment

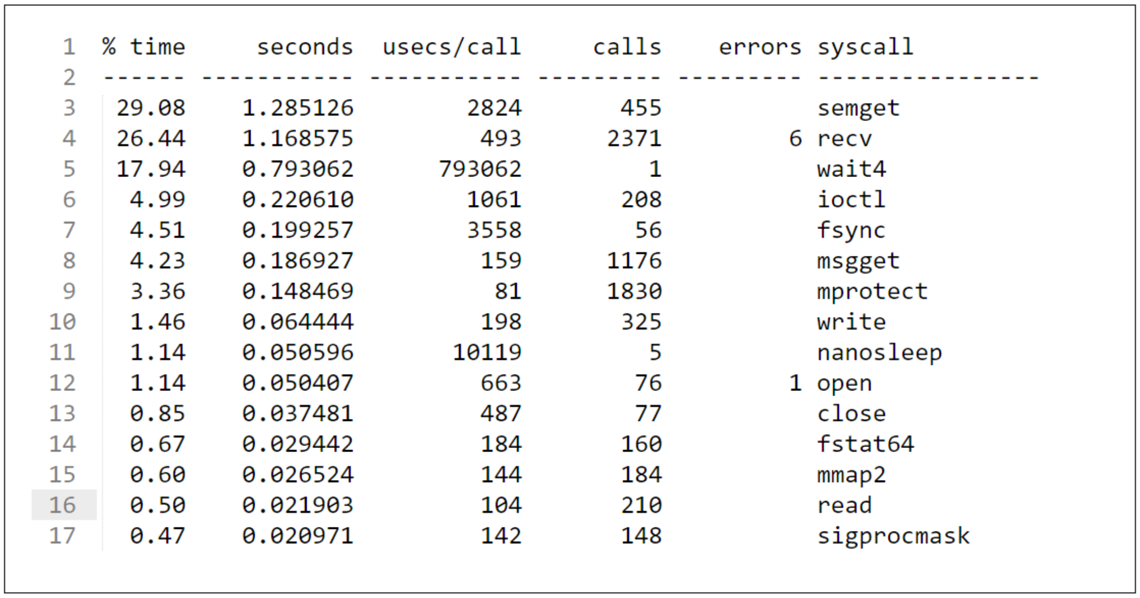
We wrote a batch script that automates the whole process. The script executes commands in the device using *ADB*. The workflow of the script is outlined in Algorithm 1.

Algorithm 1 Syscall trace collect script

```
 1: procedure COLLECT-ALL-SYSCALL-TRACE(directory)
 2:     for each apk file in directory do
 3:         pckgname ← get package name from that apk using aapt
 4:         Install the apk in the device.
 5:         Launch the app
 6:         pid ← ps(pckgname)
 7:         stracelogs[pckgname] ← output of strace(pid) with 20 seconds timeout
 8:         Force close the app
 9:         Uninstall the app
10:     end for
11:     return stracelogs
12: end procedure
```

The exact script is given in Appendix B (See Listing B.4).

We have two directories, one containing 453 malwares apks and another containing 227 non-malware apks. The malware samples are collected from *Android Malware Genome Project*. The non-malwares are directly downloaded from Google Play Store. We run the script twice. Once given the directory of malwares, and again for directory of non-malwares. After the execution, we are left with 453 malware trace files and 227 non-malware trace files. A sample single trace file is shown in figure 5.1.



```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 29.08    1.285126        2824       455           semget
 26.44    1.168575         493      2371         6 recv
 17.94    0.793062      793062         1           wait4
  4.99    0.220610        1061       208           ioctl
  4.51    0.199257        3558        56           fsync
  4.23    0.186927         159      1176           msgget
  3.36    0.148469          81      1830           mprotect
  1.46    0.064444         198       325           write
  1.14    0.050596       10119         5           nanosleep
  1.14    0.050407         663        76         1 open
  0.85    0.037481         487        77           close
  0.67    0.029442         184       160           fstat64
  0.60    0.026524         144       184           mmap2
  0.50    0.021903         104       210           read
  0.47    0.020971         142       148           sigprocmask
```

Figure 5.1: System call trace of an application

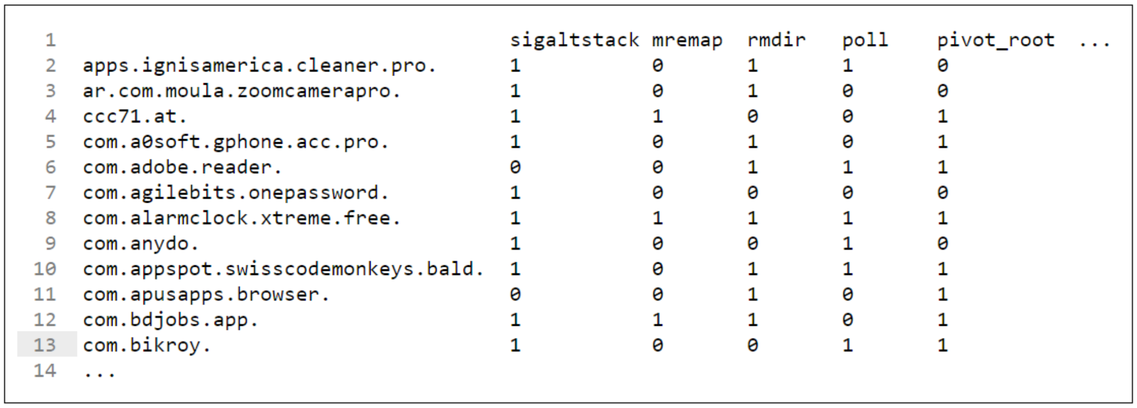
5.3 Evaluating our model

We wrote a java program (Provided in Appendix B, See Listing B.1) which further processes these files and assess our model.

The program divides the trace files into two datasets, training and validation. 50 malwares and 50 non-malware traces are chosen randomly and put in the validation dataset. The rest of the traces are used to train the classifier. The details of the training and classification steps are described in the following sub-sections.

5.3.1 Training

The program aggregates all the traces in the training dataset and produce two relation matrices M_{mal} and M_{nmal}. M_{mal} and M_{nmal} are defined in the previous chapter. A sample relation matrix is shown in figure 5.2.



```
                                     sigaltstack mremap  rmdir  poll   pivot_root  ...
apps.ignisamerica.cleaner.pro.       1           0       1      1      0
ar.com.moula.zoomcamerapro.          1           0       1      0      0
ccc71.at.                            1           1       0      0      1
com.a0soft.gphone.acc.pro.           1           0       1      0      1
com.adobe.reader.                    0           0       1      1      1
com.agilebits.onepassword.           1           0       0      0      0
com.alarmclock.xtreme.free.          1           1       1      1      1
com.anydo.                           1           0       0      1      0
com.appspot.swisscodemonkeys.bald.   1           0       1      1      1
com.apusapps.browser.                0           0       1      0      1
com.bdjobs.app.                      1           1       1      0      1
com.bikroy.                          1           0       0      1      1
...
```

Figure 5.2: A sample relation matrix between syscalls and apps

The two relation matrices are used to calculate the **Goodness rating**s of all syscalls.

5.3.2 Classification

After **Goodness rating**s of all apps have been calculated, our model is ready to calssify an unlabeled app as malware or non-malware, given its system call trace. The same program calculates Goodness raings of all applications in the validation dataset, using the equation given in section 4.2. If the Goodness rating of an app is lower than a **Threshold** **(**T**)**, our model/program flags the app as a malware, otherwise the app is considered to be non-malware.

In this chapter, we discussed the experiment for a single run. Actually the experiment was run multiple times, with different **Threshold** values. The reason behind this and the results achieved from our experiment is outlined in the following chapter.

Chapter 6

Results

The results achieved from our experiment described in chapter 5 is discussed in detail in this chapter.

In section 4.2, we introduced a **Threshold** (τ) and stated that the value of this threshold should be derived experimentally. It is dependent on the training dataset. Therefore, to find a reasonable value for τ, we tried different thresholds and for each threshold, we ran our classifier for all validation apps and calculated different metrics like **accuracy** (α), **recall** (ρ), **specificity** (σ), **precision** (ϕ) and **F-measure** (F). These metrics are defined in section 3.3. We started from a threshold value of -200 and ended with 1500, with step 10. So the classifier was run a total of 171 times (each time with all validation apps).

Ideally, the **Threshold (τ)** should be zero, but our experiment showed better accuracy for other values. To be exact, the best accuracy (87%) is achieved when we use a threshold value between 300-340.

The **Threshold (τ)** vs **Accuracy** (α) graph is shown in Figure 6.1.

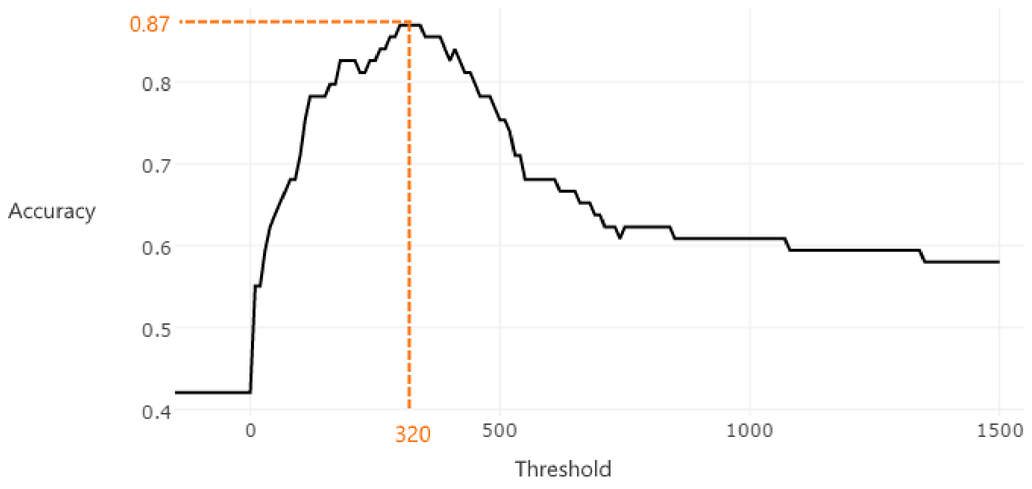


Figure 6.1: **Threshold (τ)** vs **Accuracy** (α) graph

The effect of threshold on other performance metrics of the classifier is shown in Figure 6.2 to 6.5.

With increasing threshold, the classifier would tend to classify more apps as malware. So precision falls with increasing threshold. **Threshold** (τ) vs **precision** (ϕ) graph in figure 6.2 shows $\phi = 87.8\%$ for $\tau = 320$, which is more than acceptable.

Specificity (σ), also decreases as threshold increases. In figure 6.3, **Threshold** (τ) vs **specificity** (σ) graph shows $\sigma = 82.7\%$ for $\tau = 320$. So 320 is a good value for threshold considering specificity.

Detection rate of known malwares as malwares (recall) increases with threshold. Again, the **Threshold (τ)** vs **recall** (ρ) graph in figure 6.4 shows an excellent value of **recall**, $\rho = 90.1\%$ for $\tau = 320$.

And at last, Figure 6.5 also shows an excellent **f-measure** of 0.889 for $\tau = 320$.

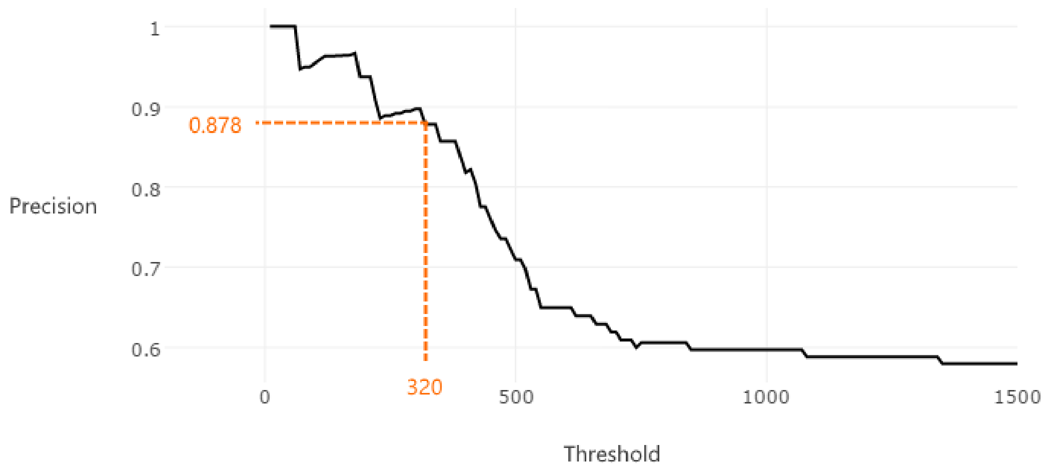


Figure 6.2: **Threshold (τ)** vs **precision** (ϕ) graph

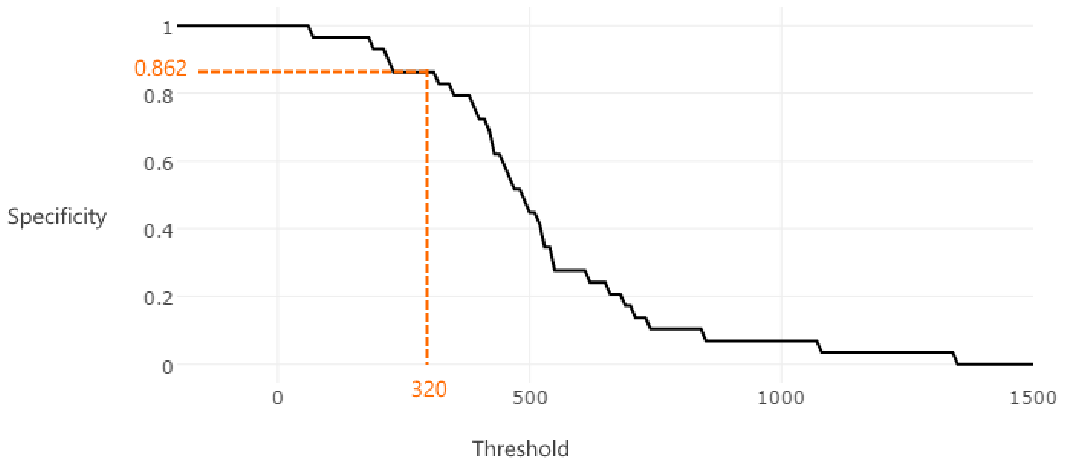


Figure 6.3: **Threshold (τ)** vs **specificity** (σ) graph

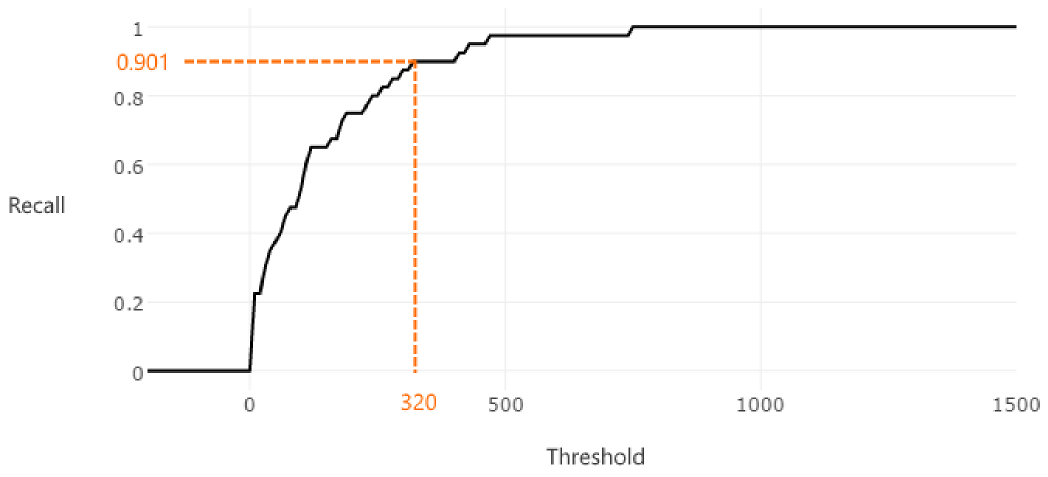


Figure 6.4: **Threshold (τ)** vs **recall** (ρ) graph

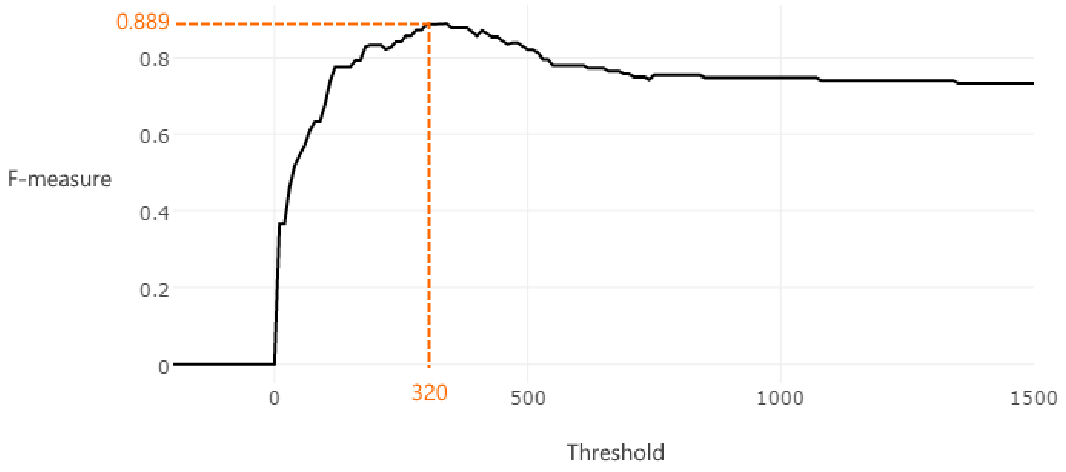


Figure 6.5: **Threshold (τ)** vs **f-measure** (F) graph

52 True Positives (actual malwares that were correctly classified as malwares)	**7 False Positives** (non-malwares that were incorrectly classified as malwares)
6 False Negatives (malwares that were incorrectly classified as non-malwares)	**35 True Negatives** (actual non-malwares that were correctly classified as non-malwares)

Figure 6.6: **Confusion Matrix** with $\tau = 320$

According to all these performance metrics, 320 seems to be a very plausible value as **Threshold**, (τ) for our model. The **Confusion Matrix** of our model resulted from using $\tau = 320$ is shown in Figure 6.6.

Although we used widely variying types of malware and non-malware application samples in training and validation of our data, it is very difficult to amass a set of malware and non-malware apps that correctly emulates the distribution of all malwares in the wild and all non-malware apps in Google Play Store. So our experimentally achieved value for parameters like threshold might not be a good choice in all cases. But if we can feed the classifier a decent representative set of malwares and non-malwares, it should produce very usable results.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we proposed a detection method based on behavioral analysis of malwares in android. We did this presentation in several chapters, a summary of which is given below.

In the first Chapter, we defined our motivation behind the research, as well as the problem statement. We also mentioned our objectives and stated our contributions.

In the second chapter, we analyzed all related works, their summary, strength of their work and limitations. We gave a brief description to each in the first section and in the second, we presented a comparison between our work and all those aforementioned.

In the third chapter, we demonstrated first step to design a detection method, the network traffic analysis. In the first section and Second, we respectively summarize and elaborate our method which is as follows. Firstly, we record all the outgoing HTTP packets and creates dump file which contain information of which port is accessing which URI. We periodically executed netstat command throughout the duration of packet dumping and saved output. It gives information of which port number is being used by which application. Now, we aggregate these two maps to create a time-sequenced log of application and the URLs each application tried to contact (App-URL table). We search the URLs in the App-URL table for known malicious domains. If an application tries to connect to a rogue domain (URL), we flag it as a malware. We can also enrich our blacklist by adding other domains contacted by a flagged application. In the final section, we showed the findings.

In the fourth chapter, we discussed our second approach for malware detection which uses system call traces of applications to predict malicious activity. The process was as follows- we use a set of applications consisting both known and unknown malwares as the training dataset. The system call trace of a single application is a list of system calls the application used during execution. After collecting system call traces, we aggregate this traces to create two binary relation matrices M_{mal} and M_{nmal}. Then, we calculate the Goodness rating of j^{th} syscall. From it, we define the Goodness Rating of that application and classify the app as malware or non-malware. We also define and calculate some metrics in this chapter.

In the fifth chapter, based on the theory described in the previous chapter, we went into action. We showed a continuous process, algorithm to code (reiterated in Appendix).

In the sixth chapter, we showed the result obtained from our given model. We used tables and graphs to present our findings in a detailed manner.

7.2 Future Works

This section is a speculation on how our thesis titled "Behavioral Malware Detection Approaches for Android" is a precursor to many research possibilities in the future.

- ***In Network-based Analysis:*** Our work in network-based detection method is one of the few definitive models among the available

Bibliography

[1] Smartphone OS Market Share, 2015 Q2, "http://www.idc.com/prodserv/smartphone-os-market-share.jsp"

[2] Kaspersky Lab and INTERPOL Report on Every Fifth Android User Faces Cyber-Attacks, "http://www.kaspersky.com/about/news/virus/2014/Every-Fifth-Android-User-Faces-Cyber-Attacks"

[3] The Number of Financial Attacks Against Android Users Tripled in 2014, "http://www.kaspersky.com/about/news/virus/2015/The-Number-of-Financial-Attacks-Against-Android-Users-Tripled-in-2014"

[4] Srikanth Ramu, "Mobile Malware Evolution, Detection and Defense," EECE 571B, TERM SURVEY PAPER, APRIL 2012.

[5] M. Chandramohan, H. B. K. Tan, "Detection of Mobile Malware in the Wild," IEEE Explore ISSN - 0018-9162 September 2012.

[6] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," IEEE Symposium on Security and Privacy, San Francisco, CA, May

[7] D. Iland, A. Pucher, T. Schauble, "Detecting Android Malware on Network Level," Technical representation UC Santa Barbara, 2012.

[8] Y. J. Ham and H. Lee, "Detection of Malicious Android Mobile Applications Based on Aggregated System Call Events," International Journal of Computer and Communication Engineering, Vol. 3, No. 2, March 2014.

[9] Y. Lin , Y. Lai , C. Chen , H. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," Computers and Security, vol. 39, p. 340 to 350.

[10] T. Isohara, K. Takemori, A. Kubota "Kernel-based Behavior Analysis for Android Malware Detection," Seventh International Conference on Computational Intelligence and Security (CIS), 2011.

[11] netstat, a Linux Binary File "https://en.wikipedia.org/wiki/Netstat"

[12] Wireshark, a network protocol analyzer for Unix and Windows "https://www.wireshark.org/"

[13] Shark for root, a network packet capture for android rooted device "https://play.google.com/store/apps/details?id=lv.n3o.shark"

Appendix A

Glossary

(In the Order of Appearance in this literature)

Cracked Applications Premium applications need a license key/passcode to work, a cracked application is illegally modified version of that premium app in order to bypass that verification

System Call A function/procedure call made by a process to System/Kernel

Trojan-SMS malware A Trojan is hidden code segments inside a “normal-looking” app, A Trojan-SMS malware targets Messaging Option in User’s mobile device

Greywares A benign application that is vague about what it does

Mal-advertising Malware attacks through In-app-advertisement

Virtual Machine (VM) A Virtual environment simulating an original device

DNS Domain Name Server

Malicious Repackaged Applications (MRAs) Malicious codes piggy-backed into a Benign Android Application

URL Uniform Resource Locator, the means to access an indicated resource

PORT numbers A port is a logical construct that identifies a service or process. A port number is an identifier of a specific port

ADB Android Device Bridge, a communication binary to communicate between a UNIX computer and android device

Classifiers In Artificial Intelligence, a Classifier is a function that use pattern matching to determine a closest match

Rooted device Rooting is the process of allowing users of devices running Android to attain privileged control (known as root access) over various Android subsystems. A rooted device is root-access enabled

strace strace is a diagnostic/debugging utility for Linux used to monitor interactions between processes and the Linux kernel, which include system calls, signal deliveries etc.

timeout enables a linux command to run with a time limit

Android NDK The NDK is a toolset that allows to implement parts of an app using native-code languages such as C and C++

AAPT Android Asset Packaging Tool, a part of the SDK (and build system) and allows to view, create, and update Zip-compatible archives (zip, jar, apk)

Confusion Matrix In the field of machine learning, a confusion matrix, also known as a contingency table or an error matrix , is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix)

Appendix B

Code of system call based classifier

The java code given below classifies malware and non-malware apps given the system call traces of training and validation apps. This code is also available to download at https://github.com/devmhd/syscall-dump-aggregator.

Listing B.1: Main.java

```
package com.devmhd;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class Main {

  static float maxgr = 0, mingr  = 10000;
```

```
private static HashMap<String, Float> goodnessRatings;
private static Graphable thVsACC, thVsTPR, thVsSPC, thVsPPV,
    thVsFmeasure;
public static ArrayList<String> removeEmptyStrings(String[] array){
  ArrayList<String> list = new ArrayList<String>();
  for(String str : array)
    if(!str.isEmpty())
      list.add(str);
  return list;
}

public static HashMap<String,Integer> getFrequenciesFromFile(File
    file) throws NumberFormatException, IOException{
  HashMap<String,Integer> map = new HashMap<String,Integer>();
  FileInputStream fStream = new FileInputStream(file);
  BufferedReader reader = new BufferedReader(new InputStreamReader(
      fStream));
  String line;
  boolean readingNow = false;
  int count = 0;
  ArrayList<String> record;
  String syscallname;
  Integer n_calls;

  while ((line = reader.readLine()) != null)   {
    if(line.isEmpty()) continue;
    if(line.startsWith("-")){
      readingNow = ! readingNow;
      continue;
    }

    if(readingNow){
      record = removeEmptyStrings(line.split(" "));
      syscallname = record.get(record.size()-1);
```

```
        n_calls = Integer.parseInt(record.get(3));
        map.put(syscallname, n_calls);
        count++;
      }
    }

    reader.close();
    if(count==0) System.out.println("No␣syscall:␣" + file.getName());
    return map;
  }

  public static HashMap<String, Float> normalize(HashMap<String,
      Integer> map, int total){

    HashMap<String, Float> newMap = new HashMap<String, Float>();
    for(Entry<String, Integer> entry : map.entrySet())
    {
      newMap.put(entry.getKey(), (float)entry.getValue()/(float)total);
    }
    return newMap;
  }

  public static HashMap<String, Float> aggregateFolder(File folder,
      String outputFileName){
    HashMap<String,Integer> syscallFrequency = new HashMap<String,
        Integer>(), singleAppFreqs;
    int n_apps = 0;
    for(File dumpFile : folder.listFiles()){
      n_apps++;
      try {
        singleAppFreqs = getFrequenciesFromFile(dumpFile);
        for(Map.Entry<String, Integer> entry : singleAppFreqs.entrySet
            ())
        {
```

```
        if(syscallFrequency.containsKey(entry.getKey())){
          syscallFrequency.put(entry.getKey(), syscallFrequency.get(
              entry.getKey())+ 1 );
        } else {
          syscallFrequency.put(entry.getKey(), new Integer(1));
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  PrintWriter writer;
  try {
    writer = new PrintWriter(outputFileName, "UTF-8");
    writer.println("" + n_apps);
    for(Map.Entry<String, Integer> entry : syscallFrequency.entrySet
        ())
    {
      String word = entry.getKey() + "," + entry.getValue();
      writer.println(word);
    }
    writer.close();

  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (UnsupportedEncodingException e) {
    e.printStackTrace();
  }
  return normalize(syscallFrequency, n_apps);
}

public static void calculateGoodnessRatings(HashMap<String, Float>
    goodnessMap, HashMap<String, Float> badnessMap){
```

```
    goodnessRatings = new HashMap<String, Float>();
    for(Entry<String, Float> entry : goodnessMap.entrySet()){
      goodnessRatings.put(entry.getKey(), entry.getValue() - ((
          badnessMap.get(entry.getKey()) == null)?0:(badnessMap.get(
          entry.getKey()))));
    }
  }

  public static float getGoodnessRating(String syscall){
    if(goodnessRatings.containsKey(syscall)){
      return goodnessRatings.get(syscall);
    } else {
      return (float) 0;
    }
  }

  public static boolean isMalware(File traceFile, float threshold)
      throws IOException{

    FileInputStream fStream = new FileInputStream(traceFile);
    BufferedReader reader = new BufferedReader(new InputStreamReader(
        fStream));
    String line;
    boolean readingNow = false;
    int count = 0;
    ArrayList<String> record;
    String syscallname;
    Integer n_calls;

    float sum = 0;

    while ((line = reader.readLine()) != null)    {
      if(line.isEmpty()) continue;
      if(line.startsWith("-")){
        readingNow = ! readingNow;
        continue;
      }
      if(readingNow){
        record = removeEmptyStrings(line.split("␣"));
        syscallname = record.get(record.size()-1);
        n_calls = Integer.parseInt(record.get(3));
        sum += (getGoodnessRating(syscallname) * (float)n_calls);
        count++;
      }
    }

    if(sum > maxgr) maxgr = sum;
    if(sum < mingr) mingr = sum;
    reader.close();
    if(count==0) System.out.println("No␣syscall␣in␣classification:␣" +
        traceFile.getName());
    return sum < threshold;
  }

  public static void runWithThreshold(float threshold){

    try{
      //aggregate results
      HashMap<String, Float> goodnessMap = aggregateFolder(new File("
          training-data/good"), "output-good.csv");
      HashMap<String, Float> badnessMap = aggregateFolder(new File("
          training-data/malware"), "output-malware.csv");
      calculateGoodnessRatings(goodnessMap, badnessMap);

      //validation
      float tp = 0, tn = 0, fp = 0, fn = 0;
      File malValidationFolder = new File("validation-data/malware");
      File nmalValidationFolder = new File("validation-data/non-malware
          ");

      for(File malwareTrace: malValidationFolder.listFiles()){
        if(isMalware(malwareTrace, threshold)) tp++;
        else fn++;
      }

      for(File nmalwareTrace: nmalValidationFolder.listFiles()){
        if(isMalware(nmalwareTrace, threshold)) fp++;
        else tn++;
      }

      thVsACC.add(new FloatPair(threshold, (tp+tn)/(tp+tn+fp+fn)));
      thVsTPR.add(new FloatPair(threshold, tp/(tp+fn)));
      thVsSPC.add(new FloatPair(threshold, tn/(fp+tn)));
      thVsPPV.add(new FloatPair(threshold, tp/(tp+fp)));
      thVsFmeasure.add(new FloatPair(threshold, 2*tp/(2*tp + fp + fn)))
          ;
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    thVsACC = new Graphable();
    thVsTPR = new Graphable();
    thVsSPC = new Graphable();
    thVsPPV = new Graphable();
    thVsFmeasure = new Graphable();

    for(float threshold = -200; threshold <= 1500; threshold +=10){
      runWithThreshold(threshold);
    }

    thVsACC.outputCsv("thVsACC.csv");
    thVsTPR.outputCsv("thVsTPR.csv");
    thVsSPC.outputCsv("thVsSPC.csv");
    thVsPPV.outputCsv("thVsPPV.csv");
    thVsFmeasure.outputCsv("thVsFmeasure.csv");
    System.out.println("" + mingr + "␣␣␣" + maxgr);
  }
}
```

Listing B.2: FloatPair.java

```
package com.devmhd;
public class FloatPair {
  public float f1, f2;
  public FloatPair(float f1, float f2) {
    super();
    this.f1 = f1;
    this.f2 = f2;
  }
  @Override
  public String toString() {
    return "" + f1 + "," + f2;
  }

}
```

Listing B.3: Graphable.java

```
package com.devmhd;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;

public class Graphable {
  private ArrayList<FloatPair> list;
  public Graphable(){
    list = new ArrayList<FloatPair>();
  }
  public void add(FloatPair pair){
    System.out.println(pair.toString());
    list.add(pair);
  }
  public void outputCsv(String filename){
    PrintWriter writer;
    try {
      writer = new PrintWriter(filename, ”UTF−8”);
      for(FloatPair pair : list)
      {
        String word = pair.f1 + ”,” + pair.f2;
        writer.println(word);
      }
      writer.close();
    } catch (FileNotFoundException | UnsupportedEncodingException e) {
      e.printStackTrace();
    }
  }
}
```

Listing B.4: Syscall-simulation.bat

```
@echo OFF

setlocal enabledelayedexpansion
 for %%a in (”∗.apk”) do (
   set FileName=%%~a

   rem //retrieve package name from an .apk
   aapt dump badging !FileName! | grep package | gawk -F: "{print $2}"
       > tmpfile

   rem //processing data recvd on last statement
   awk "{print substr($1,7,100)}" tmpfile > tmpfile2
   awk "{print substr($1, 0, length($1)-1)}" tmpfile2 > tmpfile3
   set /p pName=<tmpfile3

   echo !pName!



   rem //install .apk file into device
   adb install !FileName!

   rem //launches an application using only packageName
   adb shell monkey -p !pName! -v 50 --throttle 100

   rem //get process id for a package
   adb shell ps | grep  !pName! | awk "{print $2}"> processId
   set /p pId=<processId

   rem //run strace for 5 second, save it to packageName.csv file
   adb shell timeout -t 5 strace -c -p !pId! > !pName!.csv

   rem //force stop an application
   adb shell am force-stop !pName!

   rem //uninstall app
   adb shell pm uninstall -k !pName!
 )
```