NETWORK TRAFFIC MONITORING AND CONTROL FOR ANDROID

DISSERTATION

submitted by

NIKHIL GEORGE CB.EN.P2CYS13017

in partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY

श्रद्धावान् लभते ज्ञानम्

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

JULY 2015

NETWORK TRAFFIC MONITORING AND CONTROL FOR ANDROID

DISSERTATION

submitted by

NIKHIL GEORGE CB.EN.P2CYS13017

in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY

IN

CYBER SECURITY

Under the guidance of

Prof. Prabhaker Mateti
Associate Professor
Computer Science and Engineering
Wright State University
USA

श्रद्धावान् लभते ज्ञानम्

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

JULY 2015

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, COIMBATORE - 641 112

BONAFIDE CERTIFICATE

This is to certify that this dissertation entitled **"NETWORK TRAFFIC MONITORING AND CONTROL FOR ANDROID "** submitted by **NIKHIL GEORGE, Reg. No. CB.EN.P2CYS13017** in partial fulfillment of the requirements for the award of the **Degree of Master of Technology** in **CYBER SECURITY** is a bonafide record of the work carried out under my guidance and supervision at Amrita School of Engineering.

Dr. Prabhaker Mateti	**Dr. M. Sethumadhavan**
(Supervisor)	(Professor and Head)

This dissertation was evaluated by us on ....

INTERNAL EXAMINER EXTERNAL EXAMINER

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, COIMBATORE -641 112

TIFAC-CORE IN CYBER SECURITY

DECLARATION

I, NIKHIL GEORGE, (Reg. No: CB.EN.P2.CYS13017) hereby declare that this dissertation entitled **"NETWORK TRAFFIC MONITORING AND CONTROL FOR ANDROID "** is a record of the original work done by me under the guidance of **Prof. Prabhaker Mateti**, Associate Professor, Wright State University, and this work has not formed the basis for the award of any degree / diploma / associateship / fellowship or a similar award, to any candidate in any University, to the best of my knowledge.

Place: Coimbatore

Date: Signature of the Student

COUNTERSIGNED

Dr. M. Sethumadhavan

Professor and Head, TIFAC-CORE in Cyber Security

ACKNOWLEDGEMENTS

At the very outset, I would like to give the first honors to **Amma, Mata Amritanandamayi Devi** who gave me the wisdom and knowledge to complete this dissertation under her shelter in this esteemed institution.

I express my gratitude to my guide, **Prof. Prabhaker Mateti**, Associate Professor, Computer Science and Engineering, Wright State University, USA, for his valuable suggestion and timely feedback during the course of this dissertation.

I would like to thank **Dr M. Sethumadhavan**, Professor and Head of the TIFAC-CORE in Cyber Security, for giving me useful suggestions and his constant encouragement and guidance throughout the progress of this dissertation.

I express my special thanks to my colleagues who were with me from the starting of the dissertation, for the interesting discussions and the ideas they shared. Thanks also go to my friends for sharing so many wonderful moments. They helped me not only enjoy my life, but also enjoy some of the hardness of this dissertation.

In particular, I would like to thank **Mr. Praveen K**, Assistant Professor, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham, Coimbatore and **Mrs. Jevitha K. P** Assistant Professor, TIFAC-CORE in Cyber Security, Amrita Vishwa Vidyapeetham, Coimbatore, for providing me with the advice, infrastructure and all the other faculties of TIFAC-CORE in Cyber Security and all other people who have helped me in many ways for the successful completion of this dissertation.

Contents

List of Figures

List of Tables

Abstract

Android mobile operating system is designed to have an open architecture. This allows to extend the functionalities and capabilities of Android by installing applications. Majority of these applications and many android system service connect to Internet. Android does not provide a detailed network monitoring or filtering mechanism for users. Network Ombudsman is a network monitoring, filtering and reporting tool for android devices with context awareness and application categorizing capability. The proposed system monitors all the network connections on the mobile device and enables the users to have a time line based in depth view into the past network traffic. The reporting functionality allows to create custom filters rules to run queries on the logged network traffic information, this can be a great help for incident investigation.

Keywords: Network Ombudsman, Firewall, Traffic monitoring, Android, iptables, nftables

Chapter 1

Introduction

Android is the most popular and widely used mobile operating system as of 2013. The Android kernel is based on the long term support Linux kernel. Currently versions of Android is build based on Linux kernel 3.4 or later. The source code of Android is released by Google under Open Source license. This open architecture have nurtured the growth of applications developed by the huge community of developers. These applications helps to extend the capabilities and functionalities of the Android operating system.

An Android device is a fully networked device, say, compared to a desktop PC. Not surprisingly, 25% of smart phone [9] users use mobiles to go online rather that a computer. These users use apps to bring in social me- dia feeds, news, weather reports, traffic updates, emails, sync data, etc. From a device used only for telephony and SMS, Internet connectivity helped mobiles evolve into much needed devices in our daily lives.

From a device used only for telephony and messaging, Internet connectivity helped Mobiles devices to evolve and become a crucial part of our daily lives. Apart from making telephone calls and sending SMS mobile devices now helps to keep in touch with our social networking services, read news, check emails, do e-commerce, collaborate and share ideas with other peoples etc. So Internet connectivity have become one of the crucial feature on a mobile.

Many companies allow employees to connect their mobile devices to the cooperate network as a part of Bring Your Own Device policy. When the personal devices are used for official works the device will begin to accumulate cooperate information along with the personal information, also there will be applications that are used for the internal company related operations. Also the cooperate network polices may not allow certain application generated traffic over their network. Android is designed for normal users and so does not have any measures to isolate or protect the cooperate application or data from unauthorized access.

1.1 Problem Statement

Almost all of the applications and various Android services connect to Internet for various reasons like fetching data from server, uploading data to servers, sharing data from services or for reasons that are not made known. Users have very little control over the traffic flowing in and out of the mobile device. Android even though based on Linux kernel does not provide easy and effective network monitoring or filtering tools.

A user cannot investigate the traffic to a detailed level like the IP or domain to which a particular application or service is connecting or number of connection made to a domain on a particular hour on a particular day etc. Users cannot filter the past traffic information based on custom filters. Though many third party applications address similar kind of issues none enables such a detailed view of past network traffic data. Many monitoring tools only do real time traffic analysis only.

In the case of traffic filtering even if Android have iptables with in its kernel there is no provision for the users to make use of it. Users have to rely on third party firewall applications that require root privileges for adding or removing the rules to iptables. Rooting a device expose the device to security risks that would otherwise not exist. Also the Linux community have phased off the iptables and is moving toward the use of more efficient nftables which is not made a part of Android kernel but already made part of the standard Linux kernel from 3.13 version. Applications that dont rely on the iptables for filtering make use of the VPN server to redirect all the traffic from the device through the VPN server and filtering rules will be implemented on that server. But in such cases the VPN server can have a look on user data that is coming and going to the device and it will be great blow on the privacy of the user. Also Android treats all the applications equally and we cannot categories them and setup a group policy like disable the Internet connectivity of a particular group when connected to a particular WiFi network. Such a kind of context awareness and dynamic filtering is would have helped users who are using their devices in cooperate networks as a part of BYOD. The issues addressed by this project are as below:

- Lack of in depth time line based monitoring of Network connections.
- Provide detailed insight into past data e.g Data transfered on Oct 1, 2014.
- Provide custom filters for viewing past traffic data (AND, OR, NOT) e.g Data transfered on Oct 1, 2014 AND Target IP address .
- Monitor network traffic of user-space and kernel process.
- Monitor inter-process communication.
- Context based dynamic filtering of network traffic.
- Real-time device monitoring, to provide in-sight into active process and network connections.

1.2 Contributions Made

We contribute network monitoring and filtering subsystem and an app that we call Network Ombudsman. The Android framework service will continuously monitors and logs all the network activity on the device. The open source app called NEOM is designed to provide device statistics like real-time listing of process and network connections. The app also enables the users to firewall apps based on dynamic context aware filtering rules. The filtering is achieved via rules that can be the blacklist based manual rules or context based dynamic rules. The app also provides provisions to set the context and configure the filter polices for each context. The context can be a WiFi network or geographical location. Once the context becomes true the corresponding rules will be activated.

1.3 Organization

The reminder of this thesis is organized as follows: In chapter 2 we discuss about the background required for network monitoring and filtering along with the evaluation of network data sources available in Linux. Chapter 3 describes our proposed system. In section 4 we discuss related work. Section 5 presents evaluation of our work. Future work and conclusions make up the Section 6.

Chapter 2

Background

2.1 Status of Android Security

According to the *Google Report Android Security 2014 Year in Review* [Google.com (2015)] many significant improvements in platform security were made in year 2014.

1. *Android sandbox strengthened with SELinux.* Android 4.4 did not require SELinux to be enabled in all system domains, but with Android 5.0 SELinux is required in enforcing mode in all the domains. This provides improved protection against security flaws by preventing exposure of system functionality to apps.

2. *Improved Full Disk Encryption.* From Android 3.0 onwards full device encryption was possible using the key that us encrypted with the screen lock secret and this key is stored on the device and not sent to any where or exposed to any apps.

   From Android 5.0 onwards `scrypt` is used to prevent brute-force attacks of the user password and also the key is binded with the device hardware keystore to prevent off the device brute-force attacks.

3. *Multi-user and Guest Mode.* From Android 4.2 multi user was there for the tablet devices, Android 5.0 brought in multi-user on mobile devices along with the guest mode with restricted access to the device with out access to data and apps.

4. *Smart Lock via trusted devices.* Smart Lock trustlets are introduces in Android 5.0 which allows devices to be unlocked automatically when close to trusted devices via NFC or bluetooth.

5. *Improved Security Services* for Android designed by Google protects the user from harmful apps. One among the security service is the Verify apps that monitoring and verifies apps that are installed and check for the Potential Harmful Applications(PHA). Because of Verify Apps the PHA in 2014 was below 1%. Less that 0.15% of devices that download apps only from Google play have PHA installed.

During the period of 2014 73 security issues was patched and patches was rolled out to AOSP. The main patched vulnerabilities are SSL Heartbleed Vulnerabilities, FakeID [Forristal (2014)], OEM/ SOC Specific Vulnerabilities and Application Vulnerabilities.

2.2 A Quick Overview of Android AOSP Source Code

Open Handset Alliance led by Google give birth to Android. The members of Open Handset Alliance contributed to the Android source. Android was developed as an open platform with fully open source code base with no single central point of failure where any one industry partner could control or restrict the customizations or optimizations of other. The Android open software stack is designed with compatibility for a wide array of devices with variety of form factors. The list of folders inside Android source tree includes external, prebuilts, frameworks, packages, hardware, libcore, cts, device, development, art, bionic, ndk, sdk, system, dalvik, developers, build, bootable, pdk, tools, libnativehelper, docs, abi.

The Table 2.1 below shows the SLOC count for each directory. `sloccount` command was used to gather this result.

2.2.1 About directories in Android source tree

1. bionic - This folder have the source of all bionic C library files. Bionic is a port of BSD standard C library, by Google as a replacement of `glibc` in Android with support for x86, ARM and ARM thumb CPU instruction sets

SLOC	Directory
22326485	external
6938774	prebuilts
3141545	frameworks
2236953	packages
1472234	hardware
570990	libcore
442462	cts
373763	device
302484	development
265609	art
255308	bionic
251930	ndk
215828	sdk
182210	system
79759	dalvik
52632	developers
22208	build
17437	bootable
13255	pdk
3313	tools
1824	libnativehelper
732	docs
450	abi
Total Physical Source Lines of Code (SLOC)	39,168,185

TABLE 2.1: SLOC for each directory in Android sources

and kernel interfaces. Also includes folder `libc` - have multiple architectures that is supported to use bionic, `libdl`, `libm` - have the necessary math libraries, `libstdc++`, `libthread_db`, `linker`.

2. bootable - Have the `bootloader` and `recovery` folders. The `recovery` folder have code for creating the recovery program. This folder can be considered as the start point for creating the custom recovery.

3. build - This is the entry point of the Android build system, the `envsetup.sh` file. `build` folder have the makefiles, configuration for target devices and tools used for the build process.

4. cts - Compatibility Test Suite or the `cts` is an automated testing harness to make sure that a build complies with the Android specification.

5. dalvik - Source code for the Dalvik virtual machine.

6. art - Source code implementation of ART virtual machine.
7. development - Have the ndk, sdk and apps that are not part of the deployed app in the Operating system e.g `WidgetPreview`.
8. device - Device specific configuration files.
9. docs - Have tutorials, sample and references to AOSP
10. frameworks - Have the source of all the framework services like `System Server`, `Activity Manager` etc.
11. hardware - Hardware related source mainly Android HAL library functions.
12. ndk - Contains NDK documentations, build scripts and helper files for building the NDK, NDK source code and library files.
13. packages - Apps that are available as part of standard OS like camera, Dialer etc. Also have content providers available in the framework.
14. system - Contains different utilities used by Android for maintenance and startups. For e.g `adb`, `mkbootimg`, `run-as`, `fastboot`, etc. The init application source file and disk utilities and other system utilities resides in this folder.
15. prebuilt - Have prebuilt files that are distributed in binary form like prebuilt gdbserver and QEMU kernel, x86 based QEMU kernel for emulator etc.
16. external - Contains source for HTTPS API for Java from Apache, Apache Harmony Java runtime, Apache Xalan-Java a processor for transforming XML documents etc.

2.3 Android Network Devices

Android device have multiple network interfaces the Figure **??** below shows all the network interfaces available on a Nexus 4 device. The listing below shows the network interface statistics from the `/proc/net/dev` file.

```
Inter-|   Receive                      |  Transmit
 face |bytes    packets errs drop fifo |bytes    packets errs drop fifo
rev_rmnet1:       0       0    0    0   0        0       0    0    0
rmnet6:       0       0    0    0   0  0        0       0    0    0
rev_rmnet5:       0       0    0    0   0        0       0    0    0
rmnet1:       0       0    0    0   0  0        0       0    0    0
rmnet_usb2:       0       0    0    0   0        0       0    0    0
    lo:   49379     579    0    0   0  49379    579     0    0    0
  sit0:       0       0    0    0    0  0        0       0    0    0
dummy0:       0       0    0    0    0  0        0       0    0    0
rev_rmnet3:       0       0    0    0   0        0       0    0    0
 wlan0:       0       0    0    0    0  0        0       0    0    0
rmnet_usb0: 16759980   26746    0    0  5066263  27468   0    6    0
rmnet_usb3:       0       0    0    0   0        0       0    0    0
  p2p0:       0       0    0    0    0  0        0       0    0    0
rmnet_smux0:       0       0    0    0  0        0       0    0    0
```

Details of interfaces shown in the listing above are described below.

- *wlan0* is the WiFi interface
- *rmnet* is the interface for data service. This is a device in QUALCOMM MSM Interface(QMI) [Gautam (2013)] framework for data services. According to the QMI architecture rmnet defines channels for data transfers and control. This interface is interface that provide data connection via 2G/3G/4G/LTE.
- *lo* indicates the loopback interface
- *p2p* is the point to point tunnel interface, used for VPN connectivity.
- *usb0* indicates the USB tethering interface, used to share the network connection on the mobile device to a computer via USB port.
- *sit* is Simple Internet Transition interface used to tunnel IPV6 through and IPV4 connections.
- *dummy* is dummy network interface, used to simulate a network computing environment when there is no active network interface.

2.4 Android Networking Layer

2.4.1 Linux Network Infrastructure

Android runs on top of Linux, and hence inherits nearly all of Linux networking infrastructure.

`/proc` is the mount point for `procfs` virtual file system. `procfs` is the kernel mechanism to make the kernel internal data structure available to the user space programs. This can be considered as the information center of Kernel. The files in `/proc/net`, `/proc/uid-stat` and `/sys/class/net` have almost all the information required for the monitoring the network connections. `/proc/net` has the files `tcp, tcp6, udp` and `udp6` which have the details of active connections. `/proc/uid-stat` has directories for each user process and each directory contain `tcp-rcv` and `tcp-snd` files that gives the total tcp traffic send and received. `/sys/class/net` has directories corresponding to each network interface and each of this directory have the `statistics` directory that have the network traffic statistics corresponding to that particular device.

A snippet from the files `/proc/net/tcp,udp,raw` is shown below. A few columns, namely `txqueue:rxqueue, tr:m->when, retrnsmt, uid, timeout, ref, pointer, drops` are omitted, because they are uninteresting.

```
sl local-address   rem-address st    uid  inode
0: 0301A8C0:C59A 2EDC3AD8:01BB 01  10036  305440
1: 0301A8C0:EAEF E9AFE636:01BB 01  10103  315692
2: 0301A8C0:BA96 26DC3AD8:0050 06      0  0
3: 0301A8C0:BA95 26DC3AD8:0050 01  10122  313031
```

Each line above, except the first, represents an open socket. The `sl` value is the kernel hash slot for the socket, the `localaddress` is the local address and port number pair. The `remaddress` is the remote IP address and port number pair (if connected). `st` is the internal status of the socket.

2.4.2 VPN, `iptables`, and `nftables`

Android has in-built support for Point-to-Point Tunneling Protocol (PPTP), Layer Two Tunneling Protocol(L2TP), and VPN (Virtual Private Network) server connections.

Android inherits `iptables` mechanism and the namesake utility from Linux. Many firewalling apps on Google Play are well designed GUI for these. Rules can be set based on IP address or protocol to allow or block the traffic. But this requires the root privilege which is not available by default on Android devices. So many users usually prefer VPN based firewall over `iptables` sacrificing the privacy.

One major concern here is that VPN servers can see/capture the traffic from the device. This is a serious privacy issue because the VPN server will be able to see the login credentials for sites that don't use HTTPS, and even tamper the data passed in and out of the device etc.

`iptables` can be used to make logs of all the incoming and outgoing network connections. These logs have source and destination IP and port numbers but lacks the process information. `iptables` are used for implementing the filtering `nftables` is a replacement of `iptables` in Desktop Linux,and we wish to bring it to Android.

2.5 What is Network Monitoring and Control?

Smart phones have become part of personal life. Apps had made the lives more easy, social, productive and fun. But does this app respect the privacy of its users is a question that is hard to answer. Apps exchange information via Internet but Android, out of the box, does not provide network monitoring or filtering facilities. So it is very difficult for an user to find out to whom the apps on their device are talking to. In order to improve privacy the network communication of apps must be put under scrutiny.

By Network monitoring the project aims to monitor all the network connections in and out of the android device, including inter-process communication. All the monitored connections will be logged and uploaded onto the could server. While logging only the attributes of the connection are stored and never the data that is transmitted. The logging enables users to do postmortem analysis on the device network connections. The cloud server helps to find out malicious traffic by comparing the connections attributes to the black list of IP address.

Network control implies filtering or restricting network activity on the device. This project provides two types of filtering one based on manual black list and another based on context based dynamic black list. The context indicates events like being at a particular location or connecting to a particular WiFi network. Also the cloud server is capable of pushing filter rules based on malicious app discovery.

2.6 Linux Audit System

Linux Auditing System [Kerrisk (2013)] keeps track of what is happening in the system. It has logging facility and operates at the kernel level. Linux audit can look into file modifications and system calls. Rules can be set such that any modification or access of files will be logged or any call of a particular system call can be logged etc. auditd is a user-space component to the Linux Auditing System. Its responsible for writing audit records to the disk. Viewing the logs is done with the ausearch or aureport utilities.

We can make use of `auditd` to monitor all the connect system calls required to open the sockets. Configuring the audit rules is done with the `auditctl` utility. The command line `auditctl -a exit,always -F arch=b64 -S connect -k MYCONNECT` will make a log entry any time `connect` system call is invoked. Below is a log entry for the `firefox` process.

```
type=SYSCALL msg=audit(1422373709.783:557):arch=c000003e syscall=42
success=no exit=-101 a0=3c a1=7f7a9c6bccc0 a2=1c a3=10 items=0
ppid=1988 pid=19749 auid=4294967295 uid=1000 gid=1000 euid=1000
suid=1000 fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=(none)
ses=4294967295 comm=444E53205265737E65722023353132
exe="/usr/lib/firefox/firefox" key="MYCONNECT"
```

But the `auditd` logs lack information such as destination and source IP addresses, port number, protocol, connection status etc. So audit logs are not that useful for network monitoring.

2.7 Evaluation of network data sources

2.7.1 `/proc` file system

proc is the mount point for procfs virtual file system. procfs is the kernel mechanism to make the kernel internal data structure available to the user space programs.This can be considered as the control and information center of Kernel. Reading from the files give the runtime information about the kernel and writing into specific files allows to change the kernel parameters at the runtime. Many of the system utilities are read calls to the files in `/proc`. For example 'lsmod' is equivalent to cat `/proc/modules`, 'lspci' is equivalent to cat `/proc/pci`, `sysctl` is equivalent to altering the files in `/proc/sys`, this folder have all the configurable kernel parameters as files that are writable.

Files in `/proc` are not actually files, they are pseudo files and are not stored on the disk. That is why the ls -ls shows size as zero and the created data is the current date, also 'file' command gives the output "empty" as the contents of these files are generated on fly when these files are accessed. Most of the `/proc` folder have two kind of contents, numerical named and non numerical named.Numerical named are the folders named after the pid of each process. These folders have the information regarding process. Non-Numerical named files and folders describe some aspect of kernel operation. As an example, the file `/proc/net` contains network related information.

2.7.1.1 Description about files and folders in `/proc`

Each of the files in this folder contains information about some aspect of the Linux networking and the monitoring module of Network Ombudsman mainly fetches all the required data regarding the incoming and outgoing connections.

- Numbered folders - Process specific entries, number corresponds to PID of process, contents are
  - cmdline - Command line arguments
  - environ - Values of environment variables
  - fd - Directory, which contains all file descriptors
  - mem - Memory held by this process

  - stat - Process status
  - status - Process status in human readable form
  - cwd - Link to the current working directory
  - exe - Link to the executable of this process
  - maps - Memory maps
  - root - Link to the root directory of this process
  - statm - Process memory status information

- `/proc/net` - Status information about network protocols
  - `/proc/net/dev` - Statistics information about the configured network interface.There is one line for each logical network interface known to the kernel. In each line the interface name is followed by two sets of statistics, receive and transmit, for that interface.
  - `/proc/net/tcp,udp,raw` - Information about open tcp, udp and raw socket.Each line in each of these files represent a open socket.
  - `/proc/net/arp` - Have the arp tables, shows the mac to IP address mapping locally cached by the kernel.
  - `/proc/net/route` - Have the kernel routing tables.
- `/proc/parport` - Information about parallel ports on the system
- `/proc/sys/kernel` - Information about general kernel behaviors
- `/proc/sys` - Configurable kernel parameters by using `sysctl`

2.7.1.2 Determining owner of a network connection

Information about all TCP,UDP and RAW sockets can be gathered from the files `/proc/net/tcp`, `/proc/net/udp`, `/proc/net/raw` files, each line in these files represents an active open socket. The last column of data is labeled 'inode' and it provides a unique identifier for the socket (Figure 2.2). This inode number helps to identify the owner of each connection. Each process will have a `/proc/[pid]/fd` folder. This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor, and which is a symbolic link to the actual file. For file descriptors for pipes and sockets, the entries will be symbolic links whose content is the file type with the inode. For example, socket:[24580] will be a socket and its inode is 24580 as shown in listing below.

```
sl local_address   rem_address st    uid  inode
0: 0301A8C0:C59A 2EDC3AD8:01BB 01  10036  24580
1: 0301A8C0:EAEF E9AFE636:01BB 01  10103  315692
2: 0301A8C0:BA96 26DC3AD8:0050 06      0  0
3: 0301A8C0:BA95 26DC3AD8:0050 01  10122  313031
```

The contents of /proc/2564/fd is listed as below.

```
lrwx------ 1 root root 64 Jul 11 12:09 20 -> socket:[170076]
lr-x------ 1 root root 64 Jul 11 12:09 21 -> pipe:[170077]
l-wx------ 1 root root 64 Jul 11 12:09 22 -> pipe:[170077]
lrwx------ 1 root root 64 Jul 11 12:09 23 -> socket:[166505]
lrwx------ 1 root root 64 Jul 11 12:09 24 -> socket:[24580]
lrwx------ 1 root root 64 Jul 11 12:09 25 -> socket:[166508]
```

To find out the owner of a particular connection we have to find out which process owns the inode number associated with that connection. So a lookup table have to be maintained with [pid inode] as key-value pair. This table have to be updated periodically and also when a lookup for inode fails. When ever the owner of a process have to be looked up, the lookup table can be consulted to find out the pid of the process that owns this inode. From the pid, the process name can be found out from the `/proc/[pid]/exe` file.

2.7.1.3 Data source for files in `/proc`

As described earlier the files in `/proc` are virtual files and are pointers to the actual information that resides in the kernel. When the files are accessed the contents are dynamically generated. This is achieved by means of call back function associated with these files. This association is set to each file when they are created in `/proc` via `proc_create_data` function. Function declaration is as below.

```
struct proc_dir_entry *proc_create_data(const char *name,
umode_t mode, struct proc_dir_entry *parent,
const struct file_operations *proc_fops,void *data);
```

The parameters can be described as 'name': The name of the proc entry, 'mode': The access mode for proc entry, 'parent': The name of the parent directory under

`/proc`, `'proc_fops'`: The structure in which the file operations for the proc entry will be created, 'data': If any data needs to be passed to the proc entry.

The definition of `proc_fops` structure used to indicate the call back functions of `/proc/net/tcp` as defined in the Linux Source`/net/ipv4/tcp_ipv4.c` file is given below.

```
static const struct file_operations tcp_afinfo_seq_fops = {
.owner   = THIS_MODULE,
.open    = tcp_seq_open,
.read    = seq_read,
.llseek  = seq_lseek,
.release = seq_release_net
};
```

It will be invoked upon open,read,seek and lock release of the `/proc/net/tcp` file. So in case of read the system will invoke the `seq_read` function defined in the Linux Source`/fs/seq_file.c` file. So like wise each file in the `/proc` have call back functions invoked when a file operation is performed on them.

2.7.2 Linux Auditing System

Linux Auditing System keep track of whats happening in the system. It operates in the kernel level and have the logging facility with in itself. Linux audit can look into file modifications and system calls. We can set rules such that any modification or access of files will be logged or any call of a particular system call can be logged etc.

`auditd` is a user-space component to the Linux Auditing System. It's responsible for writing audit records to the disk.Viewing the logs is done with the `ausearch` or `aureport` utilities. Configuring the audit rules is done with the `auditctl` utility.

The `connect` system call is called when a connection is made from the system to a server by any process. So the network activity can be tracked by setting up a monitor for 'connect' system call, we can make use of `auditd` utility to add the new rule like below.

```
# auditctl -a exit,always -F arch=b64 -S connect -k MYCONNECT
```

-a – append the rule to the existing action[exit, always] rules -F – represents the architecture -S – system call to monitor -k – Filter text/key that will be added to the log

The above rule will make a log entry every time connect system call is invoked. Below is an example log entry for Firefox process.

```
type=SYSCALL msg=audit(1422373709.783:557): arch=c000003e syscall=42
success=no exit=-101 a0=3c a1=7f7a9c6bccc0 a2=1c a3=10 items=0
ppid=1988 pid=19749 auid=4294967295 uid=1000 gid=1000 euid=1000
suid=1000 fsuid=1000 egid=1000 sgid=1000 fsgid=1000 tty=(none)
ses=4294967295 comm=444E53205265737E65722023353132
exe="/usr/lib/firefox/firefox" key="MYCONNECT"
```

By default the logs will be populated in the `/var/log/audit/audit.log` file. To search the audit logs by the filter key or system call ausearch utility can be used, for example as below. `-sc` is used to specify the system call.

```
# ausearch -sc connect
```

A log snippet from the filtered logs looks like this

```
time->Tue Jan 27 21:10:47 2015 type=SOCKADDR msg=audit(1422373247
.275:446): saddr=01002F7661722F72756E2F6E7363642F736F6..type=SYSCALL
msg=audit(1422373247.275:446): arch=c000003e syscall=42 success=no
exit=-2 a0=4 a1=7fffb65c53e0 a2=6e a3=7fa3923fd7b8 items=0 ppid=201
68 pid=20223 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
sgid=0 fsgid=0 tty=pts25 ses=4 comm="aureport" exe="/sbin/aureport"
key="MYCONNECT"
```

As seen in the logs, the Linux Audit system can be used to monitor any activity only based on a system call or file modification. The logs created will tell the details about the process that triggered that event to happen. The information about the event itself very limited in the logs. So the Linux Auditing logs cannot be considered as a suitable data collection point for the monitoring the network activities because audit logs can only give the details of when a system call is invoked by a process.

2.7.3 `iptables` and VPN

Android has in-built support for Point-to-Point Tunneling Protocol (PPTP), Layer Two Tunneling Protocol(L2TP), and VPN (Virtual Private Network) server connections.

One major concern here is that VPN servers can see/capture the traffic from the device. This is a serious privacy issue because the VPN server will be able to see the login credentials for sites that dont use HTTPS, and even tamper the data passed in and out of the device etc.

Android inherits iptables mechanism and the namesake utility from Linux. Many firewall apps on Google Play are well designed GUI for these. Rules can be set based on IP address or protocol to allow or block the traffic. But this requires the root privilege which is not available by default on Android devices. So many users usually prefer VPN based firewall over iptables sacrificing the privacy.

iptables can be used to make logs of all the incoming and outgoing network connections. These logs have source and destination IP and port numbers but lacks the process information. iptables are used for implementing the filtering.

`iptables` is a command line user space utility for editing the packet filtering rule set. By default it allows all the traffic. This is targeted for the system administrators and it requires root privileges for execution. iptables requires a kernel that features the ip_tables packet filter. This includes all 2.4.x and later kernel releases. iptables can also be used to log all the packets that are coming and going out of the system. Below is a rule to do the same.

```
$ sudo iptables -A INPUT -j LOG --log-level 4 --log-prefix 'Log '
```

The flag description are as below: -A – append the rule to the end of INPUT chain -j – target or action if a match occurs -log-level – The levels are, 0 emerg, 1 alert, 2 crit, 3 err, 4 warning, 5 notice, 6 info, 7 debug –log-prefix – The prefix to be for the logs.

By default the logs will get appended along with the kernel logs /var/log/kern.log this can be changed to a file of our choice. iptables log entry will look like below.

```
Jan 28 21:00:10 Mercury kernel: [ 6218.793792] Log IN=wlan0 OUT=
MAC=dc:85:de:a1:0a:af:40:a8:f0:9c:a2:00:08:00 SRC=82.198.29.149
```

```
DST=10.12.52.71 LEN=1500 TOS=0x00 PREC=0x00 TTL=49 ID=877
DF PROTO=TCP SPT=443 DPT=35358 WINDOW=36 RES=0x00 ACK URGP=0
```

The details regarding the log are as below:
Log – log prefix
IN=wlan0 – Input interface, wireless interface
OUT= – Output interface, Empty value for locally received packets.
MAC=dc:85:de:a1:0a:af:40:a8:f0:9c:a2:00:08:00 – Destination MAC=dc:85:de:a1:0a:af, Source MAC=40:a8:f0:9c:a2:00, Type=08:00 (ethernet frame carried an IPv4 datagram)
SRC=82.198.29.149 – Source IP address
DST=10.12.52.71 – Destination IP address
LEN=1500 – Total length of IP packet in bytes
TOS=0x00 – Type Of Service, "Type" field.
PREC=0x00 – Type Of Service, "Precedence" field
TTL=49 – Time To Live is 49 hops
ID=877 – Unique ID for this IP datagram, shared by all fragments if fragmented.
DF – Don't Fragment
PROTO=TCP – Protocol name or number.Netfilter uses names for TCP, UDP, ICMP, AH and ESP. Other protocols are identified by number in `/etc/protocols`
SPT=443 – Source port (TCP and UDP). A list of port numbers is in your `/etc/services`.
DPT=35358 – Destination port
WINDOW=36 – The TCP Receive Window size.
RES=0x00 – Reserved bits
ACK – Acknowledgement flag.
URGP=0 – The Urgent Pointer allows for urgent, "out of band" data transfer.

iptable logs have more useful information that the audit logs. Also unlike the proc file system all the packets are actually logged, and it won't be missed between the polling intervals. But the main problem with these logs is that we can't identify which process that generates these packets. So iptables have to be used along with the `/proc/net` files to find the information we need.

Chapter 3

Proposed System

Network Ombudsman is a network monitoring and filtering tool for android devices with context awareness. It is built into the android framework and have an android application for interacting with the user.

Proposed system provides the users, ability to monitor the incoming and outgoing data traffic, current running process on the android device in a very detailed manner. Network Ombudsman is capable of tracking all the connection details that includes destination or source ip, port and domain details. The monitoring also includes the internal socket traffic between processes in the same device.

The network logs collected from the network monitor are uploaded to the cloud server which analyze them. This helps to rate the data hungriness of the applications and malicious ranking and based on this Network Ombudsman is capable of giving a heads up alert to the user if any ill-reputed apps are installed on the system. This system is also context aware and is capable of dynamically switching between the filter polices without user intervention. The context can be location or connection to a particular Wi-Fi network.

3.1 Requirement Specification

- Detailed monitoring and filtering of the Network connections on mobile device.
- Monitoring of inter-process communication via sockets with in the device.

- Enable the user to create blacklist/whitelist for restricting apps from accessing the internet.
- Allow the user to categories the app into different domains like personal and corporate.
- Context based blocking and unblocking for application categories/domains.
- Targeted blocking of ill-reputed application, IP or domain.
- Will have an application for interacting with user to display the monitoring reports, Firewall filter management and crowd sourcing input collection.
- The report interface
  - Enable the users to have a fine grained detailed look into the network traffic
  - Destination and Source IP, Port and Domain names of current and past connections
  - Data transferred via each connections along with the time line
  - Default option to view the total network activity with the IP, PORT, domain and data transferred details grouped by application or IP or domain.
  - Custom filters available to view the report based on application, time, date, IP, domain, data transferred etc. multiple attributes can be combined with AND, OR or NOT operators to generate custom reports.
  - All the reports are powered mainly from the data source at the cloud.
- The firewall interface
  - Application blacklist/whitelist editor
  - Application domain manager

3.2 Why Android Needs a Network Ombudsman

ombudsman: A person (such as a government official or an employee) who investigates complaints and tries to deal with problems fairly.
Every user of a smart phone would like to be able to ask questions and make demands of the following kind.

- What connections did my device make between 1:00 PM and 3:23 AM on May 22, 2015?
- How many of those connections were to www.google.com and how long did they last?
- Show me the IP addresses that were connected to the device between Mar 30, 2015 and May 21, 2015.
- When I am at the GPS locations X, Y, Z, shut down all roaming services.
- When connected to WiFi network with particular SSID, block apps A,B C from accessing Internet.

We are adding a new service that can answer the above questions and demands. We are not planning on interpreting such questions in a natural language. Instead, we are building an app with an intuitive navigation that will get you those answers. Answering these queries depends on us having captured all past connections. Keeping all past data implies moving most of it to cloud storage to preserver device storage space. The ability to query gives us an edge over the conventional firewalls. Network Ombudsman is not an intrusion detection system (IDS). It is not designed to detect any intrusion or privacy breaches by itself but enables the identification of such events by other tools that can retrospectively analyze.

3.3 High Level System Description

The Network ombudsman system comprises of (i) a new Android Framework component and (ii) an App that we call the NEtwork OMbudsman or NEOM for short. The Framework component is a network monitoring service that runs continuously in the background, monitors and logs all the network connections. The logs are periodically uploaded to the cloud server. The key components of the Network Ombudsman or NEOM app are (i) Real-time network and process Monitor, (ii) Network Filter, and (iii) Cloud Server. Real-time network and monitor shows all the active network connections with IP, port, protocol and connections status details. The process monitor shows the memory consumption, process name, package name of all the currently running process.

The cloud server stores all the logs uploaded from device and analyzes for malicious activity. It also creates new filter rules to block an app, if any malicious activity is deduced from the logs.

All the monitored data that will be logged by the monitoring service will be uploaded into the cloud server by the log uploader. The android application is the user interface for the Network Ombudsman. User can view the reports, change the firewall rules, group apps into domains and take part in the crow sourcing reputation building of the apps. The cloud server stores all the logs from the devices.

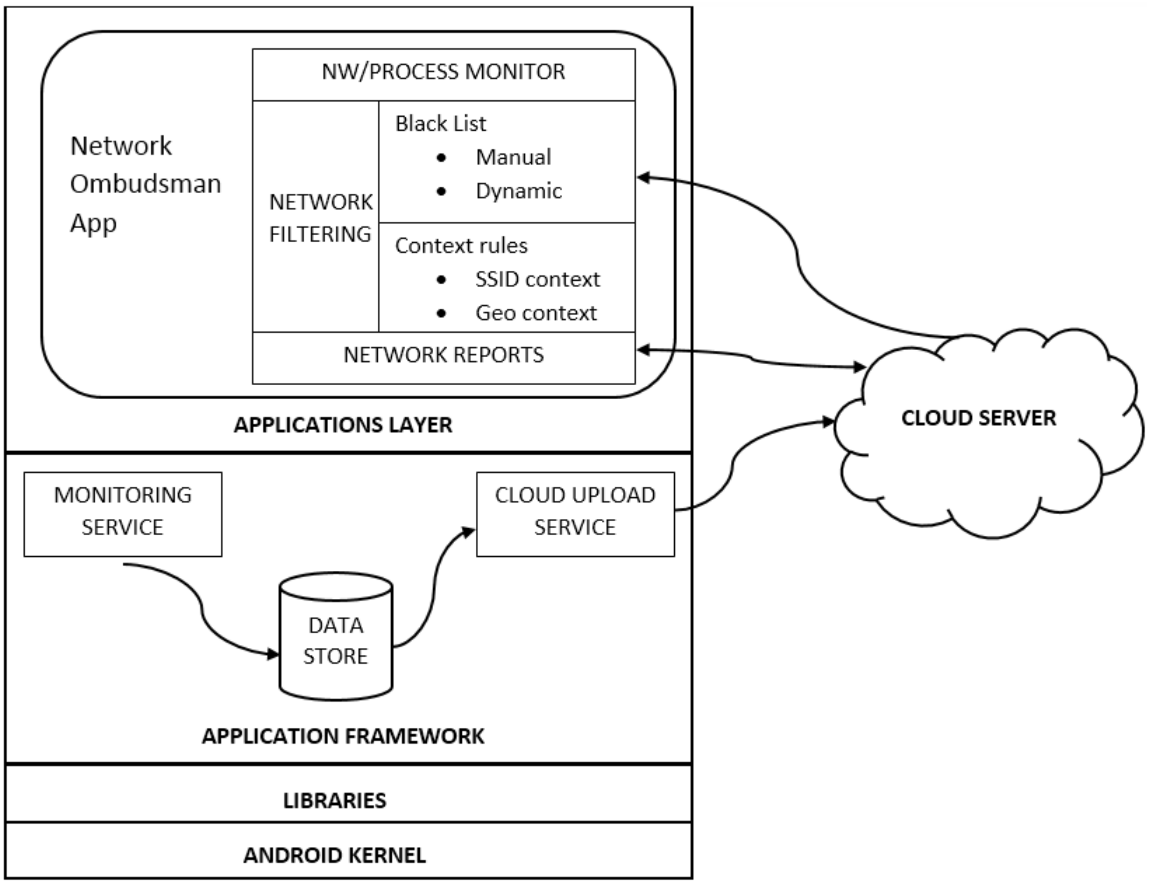


FIGURE 3.1: High Level Architecture

3.4 Detailed description of proposed system

3.4.1 Network Monitoring Service

The network monitoring background service keeps a tab on all the network connections by reading the details from Linux networking infrastructure (Section 2.7.1) and logs them to flat files. These logs are uploaded to a cloud storage server at regular intervals.

To discover the owner process of each active network connection, a look-up hash table is maintained by the monitoring service. This hash table has the pid and process name corresponding to each inode number. A single process can have multiple open sockets and so multiple inodes can be associated with it. The mapping in hash table will look similar to that shown below.

```
90960: [10666, com.android.chrome]
68557: [13061, com.piazza.android]
13127: [13247, com.skype.raider]
43126: [10518, com.google.android.apps.plus]
56555: [11725, com.google.android.calendar]
16735: [11369, com.linkedin.android]
```

In the hash table, the `inode` number is the key and it's corresponding value is the combination of `pid` and process name. This look up hash table is updated periodically. The data for this hash table is obtained as follows. Each process has an individual directory under `/proc`, with the directory name same as its pid number. The sub-directory named `fd`, has the list of all the open sockets, with corresponding `inode` numbers. Monitoring service iterates over the `fd` folders of all the processes and keeps the hash table updated. Once the owner for each socket is identified that information is concatenated with the socket information along with the time stamp and written into a log file.

The attributes collected in the log are listed in Table 3.1 below.

timestamp	Date and time of connection log
localIP	Device IP address
localport	Local port number of the process
remIP	Remote IP address
remport	Remote port number
tcpConStatus	TCP connection status
inode	iNode number of the socket
pid	Process ID
processName	Name of the process
protocol	Application layer protocol used
interface	Network connection Interface

Table 3.1: Description of attributes collected in logs

Logs are periodically sent to the server. Logs uploaded successfully are removed from the log file. This will keep the size of log file in check. The below listing shows a snippet of logs generated by the network ombudsman service.

```
2015-04-0810:13:37:853|192.168.0.112|40869|173.194.38.183|443|
TCP_SYN_SENT|inode:53310|pid:2555|com.android.chrome |protocol:
HTTPS|wlan0:|Recv:44942087|Trns:265139937

2015-04-0810:13:37:853|192.168.0.112|35349|54.225.64.222|443|
TCP_SYN_SENT|inode:70387|pid:2555|com.android.chrome|protocol:
HTTPS|wlan0:|Recv:44942087|Trns:265139937

2015-04-0810:13:37:853|192.168.1.3|39802|179.60.192.3|443|
CLOSE_WAIT|inode:68557|pid:13061|com.piazza.android|protocol:
HTTPS|wlan0:|Recv:21942087|Trns:165139937

2015-04-0810:13:37:853|192.168.1.3|39556|103.20.92.129|443|
ESTABLISHED|inode:16735|pid:11369|com.linkedin.android|protocol:
HTTPS|wlan0:|Recv:44942087|Trns:265139937

2015-04-0810:13:37:853|192.168.1.3|48760|64.4.46.55|443|
ESTABLISHED|inode:13127|pid:13247|com.skype.raider|protocol:
HTTPS|wlan0:|Recv:139157|Trns:189857|Trns
```

Each line shows the details of a single socket opened by a process. A single process may have multiple active sockets and each of those will be captured. Each attribute is separated by a vertical bar symbol, so that parsing is easy. The first field is the time stamp of when the socket is active, then comes the device local IP/port address, the remote IP/port address, status of tcp connection , unique inode identification number for the connection, process ID, name of the process, application layer protocol, and lastly the interface through which connection is handled.

The log data from all the devices will be analyzed by the cloud server for malicious activity. E.g., discover connections made to known malicious domains, or malicious process activity, etc.

3.4.2 Cloud Server

The cloud server maintains a black list, seeded with a known list of malicious IP/port addresses, domains names. This is updated with reports gathered from client devices registered.

The cloud server analyzes the logs uploaded from each device and looks for anomalies by comparing with the black list. From the uploaded logs, destination IP is extracted and checked with black lists on the server. A match indicates that the app that generated the traffic is malicious, as it is contacting a known malicious IP address. So, to block this kind of traffic, a filter rule will be created by the server, that filters out further traffic to this malicious IP address.

The filter rules are `iptable` rules in case of blocking traffic to an IP or domain. If the app has to be blocked the cloud server will push the app name to the client devices and client devices will find out the `UID` of the app and creates `iptable` filter rule with `-- uid-owner`.

The user will be notified about the new filter rule, downloaded from the cloud and can choose to apply or ignore the filter policy. The details of the app that generated the traffic will also be included in the notification. A user can uninstall the app or can choose to ignore the notification.

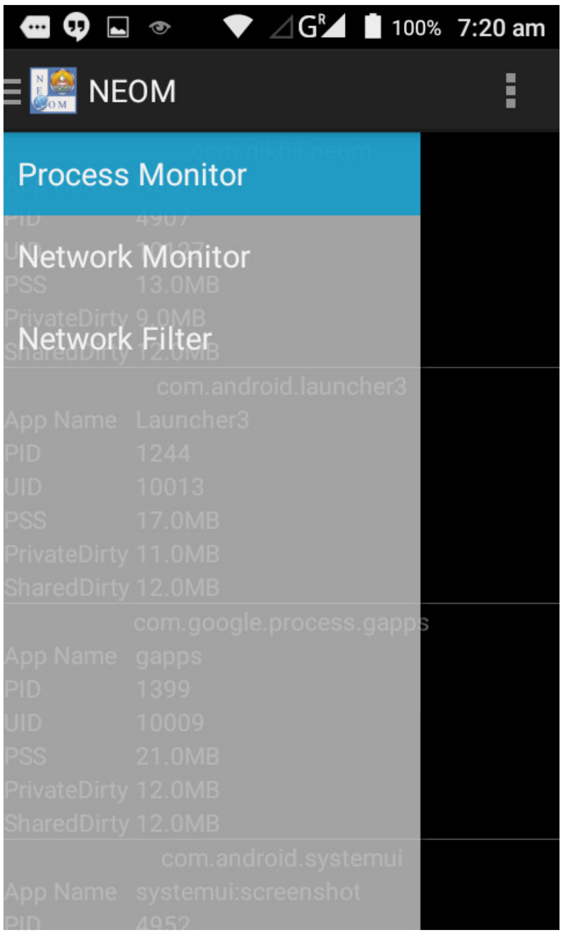
3.4.3 Network Ombudsman App

This section summarizes the internals of our NEOM app. Versions of the app are GPL open sourced and stored at `https://github.com/nikhilgeo/NEOM_App`. The screen shots shown below are taken from the alpha version of our app.

The NEOM app shown in Figure 3.2 is designed with a navigation drawer layout, with three drawer list items: Process Monitoring, Network Monitoring and Filtering. The app has a single MainActivity and multiple Fragments, one each for the items in the navigation drawer. The selection of each item in the navigation drawer loads the corresponding fragment.

3.4.3.1 Process Monitoring Tab

In the Process Monitoring tab (Figure 3.3), all the currently active processes are listed with their process names, process IDs, User IDs, application names and



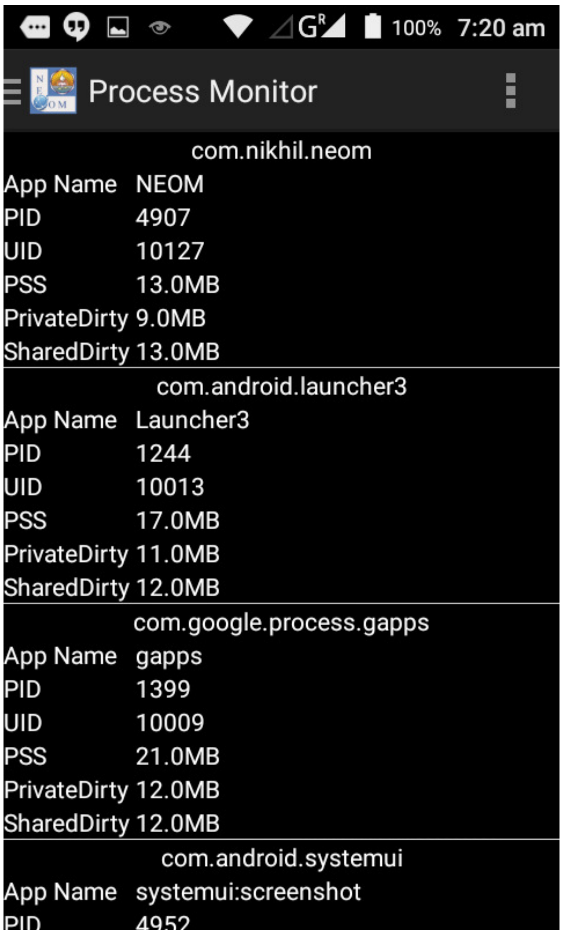
```
PSS           13.0MB
PrivateDirty  9.0MB
SharedDirty   12.0MB
              com.android.launcher3
App Name      Launcher3
PID           1244
UID           10013
PSS           17.0MB
PrivateDirty  11.0MB
SharedDirty   12.0MB
              com.google.process.gapps
App Name      gapps
PID           1399
UID           10009
PSS           21.0MB
PrivateDirty  12.0MB
SharedDirty   12.0MB
              com.android.systemui
App Name      systemui:screenshot
PID           4952
```

FIGURE 3.2: Menus

memory usage details. `getRunningAppProcesses()` from the `ActivityManager` class is used to fetch all the running process. This function returns a list of type `RunningAppProcessInfo`. This list have information about the process that we require. In order to find the name of the app that created the process `getApplicationLabel()` function from class `PackageManager` is invoked. In some cases for framework level services that are not started by any app there won't be any app name, in that case we show the package name itself. Code listing below shows the above functions.

```
ActivityManager activityManager = (ActivityManager) getActivity().
getSystemService(Context.ACTIVITY_SERVICE);
List<ActivityManager.RunningAppProcessInfo> pidsTask =
activityManager.getRunningAppProcesses();
```

For each process memory usage is also shown to track how heavy a process is. There are two kinds of memory allocation in Android.



com.nikhil.neom

App Name	NEOM
PID	4907
UID	10127
PSS	13.0MB
PrivateDirty	9.0MB
SharedDirty	13.0MB

com.android.launcher3

App Name	Launcher3
PID	1244
UID	10013
PSS	17.0MB
PrivateDirty	11.0MB
SharedDirty	12.0MB

com.google.process.gapps

App Name	gapps
PID	1399
UID	10009
PSS	21.0MB
PrivateDirty	12.0MB
SharedDirty	12.0MB

com.android.systemui

App Name	systemui:screenshot
PID	4952

FIGURE 3.3: Process Monitor

- *Private (Clean and Dirty) RAM.* This is the memory that is being used individually by the process. Private indicates the amount of RAM that the system gets when the process is terminated. Generally, the most important portion of Private is private dirty RAM, which is the most expensive because it is used individually by a process and its contents exist only in RAM so can not be paged out to the internal storage. All Dalvik and native heap allocations are in the private dirty RAM; Dalvik and native allocations shared with the Zygote process are shared dirty portion of RAM.

- *Proportional Set Size (PSS).* This is a measurement of process's RAM use that takes into account sharing pages across multiple processes. Any RAM pages that are unique to the process directly contribute to its PSS value, while pages that are shared with other processes contribute to the PSS value only in proportion to the amount of sharing. For example, a page that is shared between two processes will contribute half of its size to the PSS of each process.

One of the characteristic of the PSS measurement is that you can add up the PSS across all processes to determine the actual memory being used by all processes. This means PSS is a good measure for the actual RAM weight of a process and for comparison against the RAM use of other processes and the total available RAM.

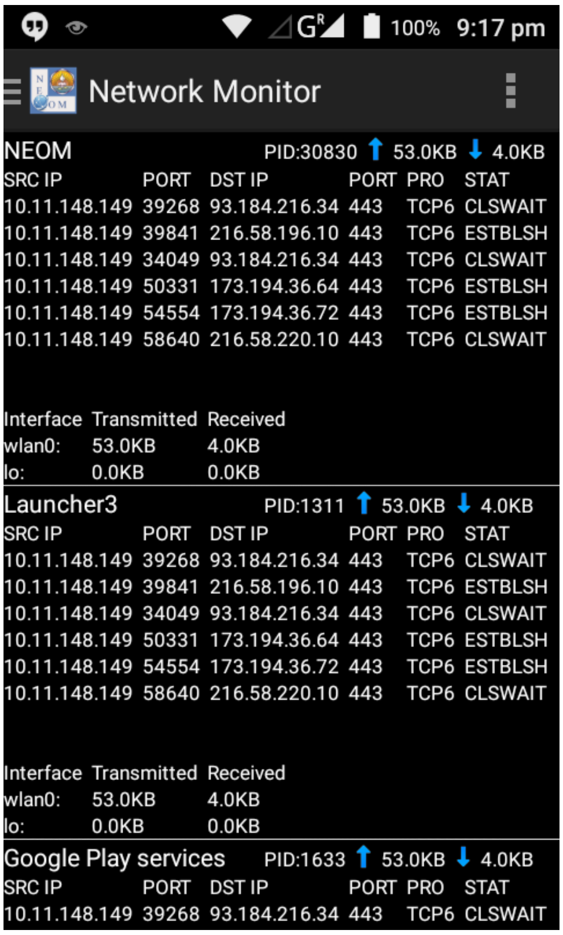
To find the memory consumption details of each process `getProcessMemoryInfo()` function from class `ActivityManager` class is used. This function will return an array of type `MemoryInfo` using this array Total PSS, Total private dirty and total shared dirty was fetched using `getTotalPss()`, `getTotalPrivateDirty()`, `getTotalSharedDirty()` functions. PSS (Total Proportional Set Size) is the count of pages a process has in memory exclusively, plus the pages divided by the number of other process sharing. Total private dirty is the allocated RAM for a process that cannot be paged out to a disk and effectively which will become free when process is terminated. Below is the code snippet that shows how memory usage information is gathered.

```
mi = activityManagerMEM.getProcessMemoryInfo(new int[] { PID });
memInfoModel[0] = mi[0].getTotalPss() / 1000;// In KB
memInfoModel[1] = mi[0].getTotalPrivateDirty() / 1000;
memInfoModel[2] = mi[0].getTotalSharedDirty() / 1000;
```

3.4.3.2 Network Monitoring Tab

The Network Monitoring tab (Figure 3.4) shows all the currently active network connections. The connections are grouped together by process and listed under process name. For each connection, details like source IP, destination IP, source port, destination port, connection status, and protocol details are displayed. Connection from same source and destination IP but using different port number is treated as a unique. No other monitoring apps on Android gives these amount of details.

To get all the network connection details, first all the running process are found out using `getRunningAppProcesses()` from the `ActivityManager` class. Once the PID of the process are extracted we go to look into the `/proc/pid/net/tcp`, `/proc/pid/net/udp`, `/proc/pid/net/tcp6`, `/proc/pid/net/udp6` files to fetch all the active tcp, tcp6, udp and udp6 network connections. We also parse the `/proc/pid/net/dev` file to get the network interface statistics, it's from this file we find out which network interface is being used by the the process and



```
NEOM                                 PID:30830 ↑ 53.0KB ↓ 4.0KB
SRC IP        PORT  DST IP        PORT PRO  STAT
10.11.148.149 39268 93.184.216.34 443  TCP6 CLSWAIT
10.11.148.149 39841 216.58.196.10 443  TCP6 ESTBLSH
10.11.148.149 34049 93.184.216.34 443  TCP6 CLSWAIT
10.11.148.149 50331 173.194.36.64 443  TCP6 ESTBLSH
10.11.148.149 54554 173.194.36.72 443  TCP6 ESTBLSH
10.11.148.149 58640 216.58.220.10 443  TCP6 CLSWAIT

Interface Transmitted Received
wlan0:    53.0KB      4.0KB
lo:       0.0KB       0.0KB
Launcher3                            PID:1311 ↑ 53.0KB ↓ 4.0KB
SRC IP        PORT  DST IP        PORT PRO  STAT
10.11.148.149 39268 93.184.216.34 443  TCP6 CLSWAIT
10.11.148.149 39841 216.58.196.10 443  TCP6 ESTBLSH
10.11.148.149 34049 93.184.216.34 443  TCP6 CLSWAIT
10.11.148.149 50331 173.194.36.64 443  TCP6 ESTBLSH
10.11.148.149 54554 173.194.36.72 443  TCP6 ESTBLSH
10.11.148.149 58640 216.58.220.10 443  TCP6 CLSWAIT

Interface Transmitted Received
wlan0:    53.0KB      4.0KB
lo:       0.0KB       0.0KB
Google Play services                 PID:1633 ↑ 53.0KB ↓ 4.0KB
SRC IP        PORT  DST IP        PORT PRO  STAT
10.11.148.149 39268 93.184.216.34 443  TCP6 CLSWAIT
```

FIGURE 3.4: Network Monitor

amount of data transmitted via the interfaces. The below code spinet shows how `/proc/pid/net/tcp` file is parsed and connections details are extracted.

```
BufferedReader in = new BufferedReader(new FileReader("/proc/"+
          PID + "/net/tcp"));
String line;
while ((line = in.readLine()) != null) {
 line = line.trim();
 String[] fields = line.split("\\s+", 10);
 int fieldn = 0;
 if (fields[0].equals("sl")) {
    continue;}
 Connection connection = new Connection();
 String src[] = fields[1].split(":", 2);
 String dst[] = fields[2].split(":", 2);
 connection.src = getAddress(src[0]);
 connection.spt =   String.valueOf(getInt16(src[1]));
 connection.dst = getAddress(dst[0]);
 connection.dpt = String.valueOf(getInt16(dst[1]));
 connection.uid = fields[7];
 connection.pro = "TCP";
 Integer conStat = getInt16(fields[3]);
 connection.stat = states[conStat - 1];
 connections.add(connection);
}
```

The tcp connection status are derived from the array `String states[]`. The integer value corresponding to each connection state are as described in the Table 3.2. The connection status starts with integer value 1 for `TCP_ESTABLISHED`. Each status is stored in each array location and retrieved by decrementing the integer value by 1 to get the corresponding array index.

Connection Status	Integer Value
TCP_ESTABLISHED	1
TCP_SYN_SENT	2
TCP_SYN_RECV	3
TCP_FIN_WAIT1	4
TCP_FIN_WAIT2	5
TCP_TIME_WAIT	6
TCP_CLOSE	7
TCP_CLOSE_WAIT	8
TCP_LAST_ACK	9
TCP_LISTEN	10
TCP_CLOSING	11
TCP_MAX_STATES	12

Table 3.2: Connection Status and Integer Value

3.4.3.3 Network Filtering

From the Filtering tab users can configure the manual black list and context based filtering rules. Network Ombudsman makes use of `iptables` for implementing firewall. The firewall rules are based on the following.

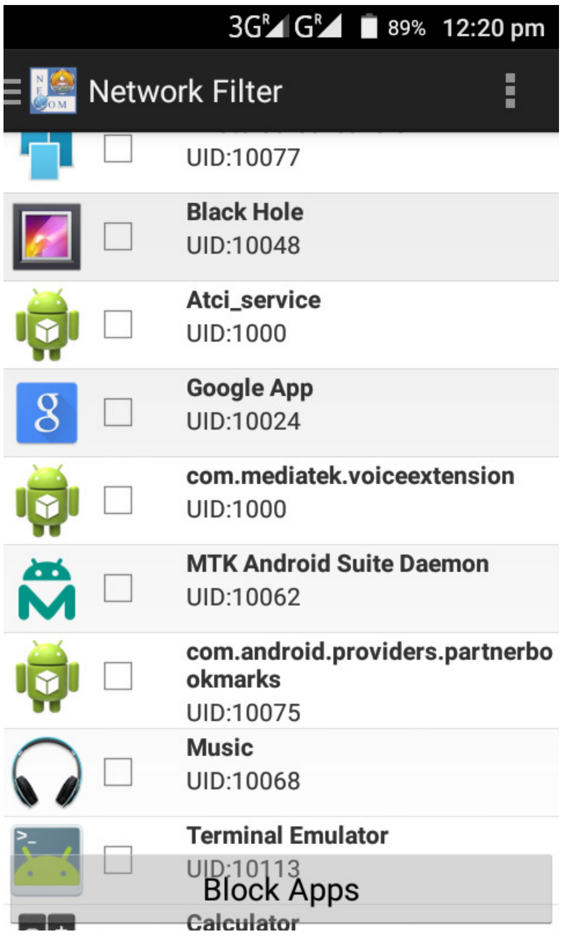
Manual Black List names apps, IP address or domain names to be blocked, set by the user.

Dynamic Black List list of apps and IP address that are found to be malicious by the cloud server and pushed to the device.

Geographical Location Context allows filter rules to be set automatically when the device happens to be located at a pre-configured location.

Network SSID based filter rules are activated once the device gets connected to a pre-configured WiFi network. This is useful for users that connect their device to corporate networks as a part of BYOD polices. The rules can be configured such that personal apps will be automatically firewall when device gets connected to corporate network.

Figure 3.5 shows screen from which manual black list can be managed. User can select the app that have to blocked from network access. Figure 3.6 shows the context configuration screen where the user can add geographical locations and WiFi SSID as the context. Once the context is set, corresponding rules can be edited by clicking on the setting icon next to the active context list.


3G 89% 12:20 pm

Network Filter

UID:10077

Black Hole
UID:10048

Atci_service
UID:1000

Google App
UID:10024

com.mediatek.voiceextension
UID:1000

MTK Android Suite Daemon
UID:10062

com.android.providers.partnerbookmarks
UID:10075

Music
UID:10068

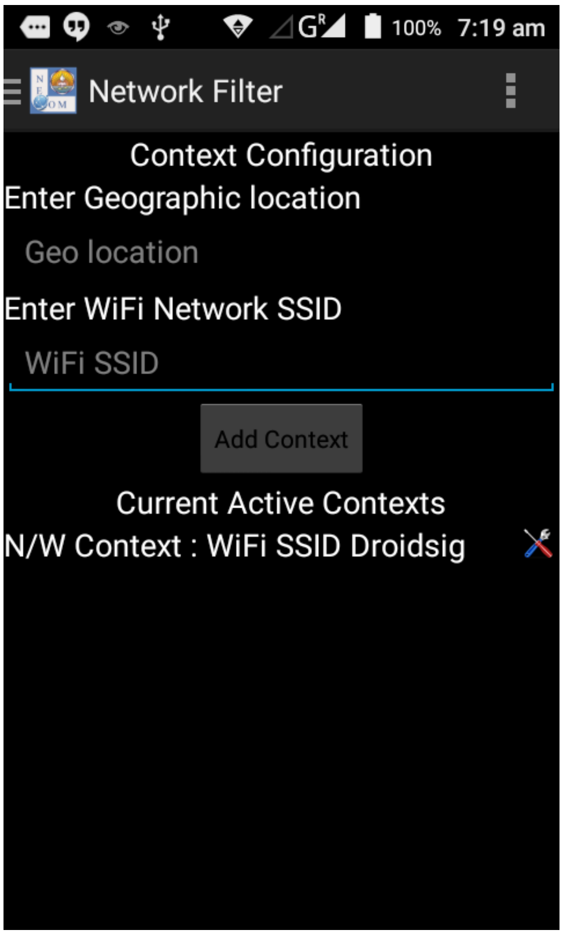
Terminal Emulator
UID:10113

Block Apps

Calculator

FIGURE 3.5: Network Filter

Apart from the manual filtering rules, the dynamic filtering rules are pushed by the cloud server. The server analyze the logs uploaded from the mobile device and find out new apps or IP that are malicious. Server will then create filtering rules and push them to the client devices.



100% 7:19 am

Network Filter

Context Configuration

Enter Geographic location

Geo location

Enter WiFi Network SSID

WiFi SSID

Add Context

Current Active Contexts

N/W Context : WiFi SSID Droidsig

FIGURE 3.6: Context Rule Configuration

To implement the filtering for apps UID is used in the `iptable` rules. In Android each app is given a unique UID so to control the network connections from a particular app we can make use of `-m owner --uid-owner` in the iptable rules. An example rule to block all the outgoing connections from app with UID 10024 is as below:

```
iptables -A OUTPUT -m owner --uid-owner 10024 -j DROP
```

In the above rule -A indicates to append the rule to already existing rules in the OUTPUT chain. -j represents the action that has to be taken when a match is found. To delete an existing rule instead of -A we should use -D like in below:

```
iptables -D OUTPUT -m owner --uid-owner 10024 -j DROP
```

Execution of the `iptables` commands requires root privilege, to obtain root privilege below steps were used in NEOM app.

- Used `java.lang.Runtime.exec(String command)` to execute the su command as a separate process.
- By default, the created subprocess does not have its own terminal or console. All its standard I/O (i.e. `stdin`, `stdout`, `stderr`) operations will be redirected to the parent process, where they can be accessed via the streams obtained using the methods `getOutputStream()`, `getInputStream()`, and `getErrorStream()`. The parent process uses these streams to feed input to and get output from the subprocess.
- Used `Process.getoutputStream()` method to provide the commands to be executed in the su shell. Output to the stream is piped into the standard input of the process represented by this Process (su) object

The code is as below depicts the starting a root privileged shell process.

```
Process suProcess = Runtime.getRuntime().exec("su");
DataOutputStream outputs = new DataOutputStream(suProcess.getOutputStream());
//outputs connected to input stream of su
os.writeBytes(Command_to_exe + "\n");
```

When `su` process is created the Supersu.apk will mediate the granting of root permission as shown in Figure 3.7. Figure 3.8 shows the `iptable` being updated with new rules after a block rule is updated from NEOM, to capture this result `adb shell` was used to list the `iptable` rules on the mobile device.

To identify the WiFi context changes a broadcast receiver was set up to receive `android.net.wifi.STATE_CHANGE` and `android.net.wifi.WIFI_STATE_CHANGED`. The intent filters we registered in the `AndroidManifest.xml`. The discussed broadcast will be sent by the system when device is connected/disconnected to a new WiFi network. In short when ever the wifi state changes. The receiver class name is specified in the `AndroidManifest.xml` file. It is that class which will be invoke when the broadcast is received. The class should inherit `BroadcastReceiver` class and should override `onReceive` function. The `onReceive` will be invoked

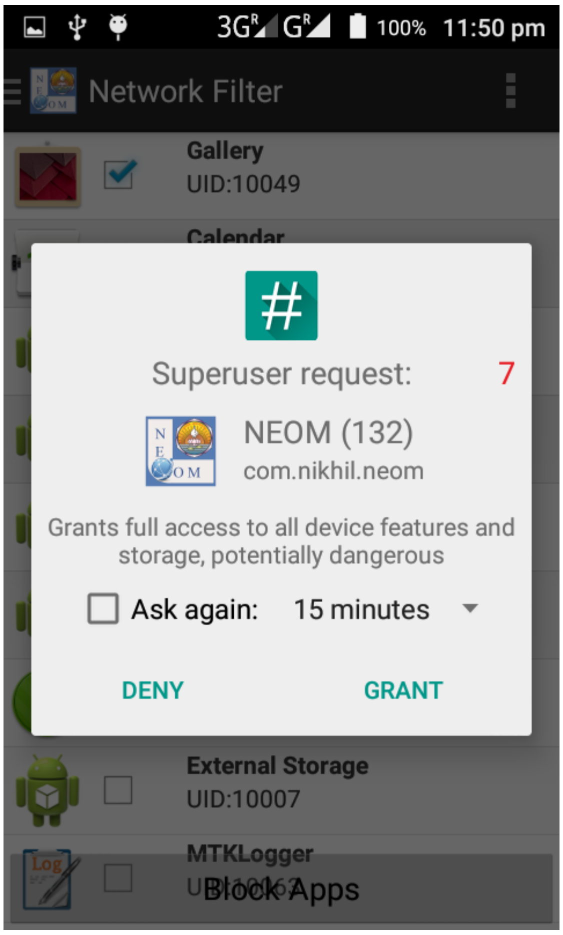
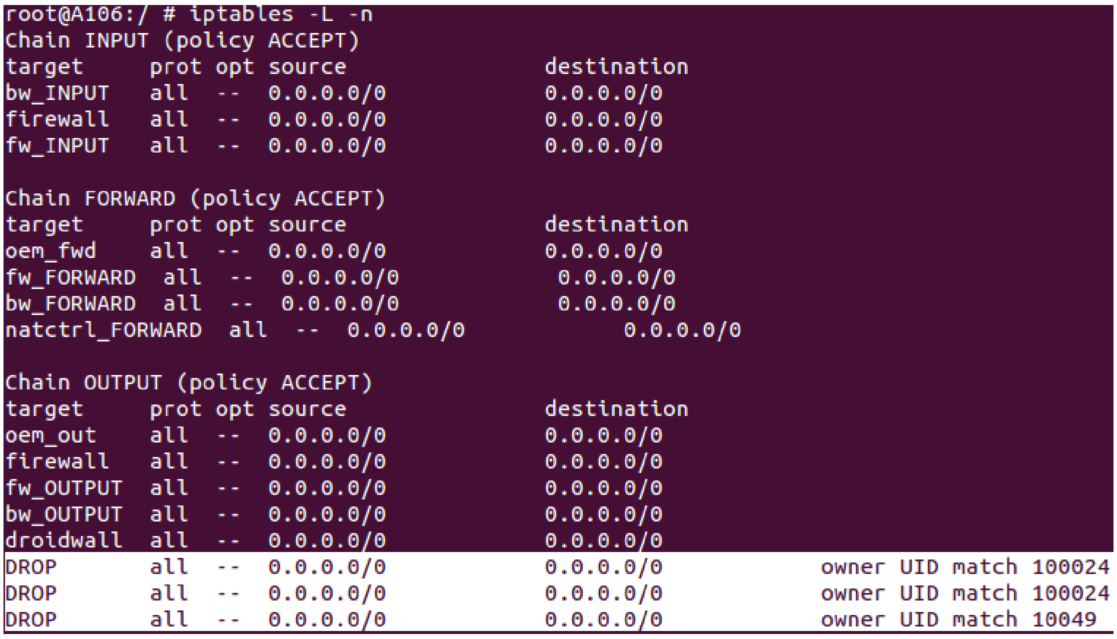


FIGURE 3.7: SuperSu mediating root access for NEOM



```
root@A106:/ # iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
bw_INPUT   all  --  0.0.0.0/0            0.0.0.0/0
firewall   all  --  0.0.0.0/0            0.0.0.0/0
fw_INPUT   all  --  0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
oem_fwd    all  --  0.0.0.0/0            0.0.0.0/0
fw_FORWARD  all  --  0.0.0.0/0            0.0.0.0/0
bw_FORWARD  all  --  0.0.0.0/0            0.0.0.0/0
natctrl_FORWARD  all  --  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
oem_out    all  --  0.0.0.0/0            0.0.0.0/0
firewall   all  --  0.0.0.0/0            0.0.0.0/0
fw_OUTPUT  all  --  0.0.0.0/0            0.0.0.0/0
bw_OUTPUT  all  --  0.0.0.0/0            0.0.0.0/0
droidwall  all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0            owner UID match 100024
DROP       all  --  0.0.0.0/0            0.0.0.0/0            owner UID match 100024
DROP       all  --  0.0.0.0/0            0.0.0.0/0            owner UID match 10049
```

FIGURE 3.8: New rules added to OUTPUT chain in iptables

when ever the broadcast is received. So in this function we have to implement the loading of appropriate rules. Below shown is code snippet from `AndroidManifest.xml` showing the broadcast receiver registration.

```
<receiver android:name="com.nikhil.neom.WifiReceiver" >
<intent-filter android:priority="100" >
<action android:name="android.net.wifi.STATE\_CHANGE" />
<action android:name="android.net.wifi.WIFI\_STATE\_CHANGED" />
</intent-filter>
</receiver>
```

In order to identify the WiFi SSID we made use of `getConnectionInfo()` function from the `WifiManager`. The `getConnectionInfo()` function will return `WifiInfo` object which provide `getSSID()` function to retrieve the WiFi SSID. The code for retrieving the WiFi SSID is as below.

```
WifiManager wifiManager = (WifiManager) context
.getSystemService(Context.WIFI_SERVICE);
WifiInfo wifiInfo = wifiManager.getConnectionInfo();
String ssid = wifiInfo.getSSID();
```

The settings option menu in the app helps to configure the upload frequency/interval of the collected log files by the framework service. Also the framework service can be paused or stopped from starting at the boot up via settings.

3.4.4 Log Uploads

- All the logs created by the monitoring daemon will be uploaded to the cloud server
- A log uploader will be uploading all the logs to clouds in a configurable way like configurable time interval for log upload, burst or stealth mode etc.
- All the uploaded logs will be deleted from the device.

3.4.5 Reports

- The data for the reports are pulled out of the logs that are stored in the cloud or on the device
- Users will be able to create custom reports by joining connection attributes with AND,NOT and OR
- A detailed timeline based inspection of past network traffic will be possible
- Implemented as an app.
- Various data representation methods will be available like charts, graphs etc.

3.5 Proof-of-Concept Monitoring on Ubuntu Linux

As a proof of concept we implemented network monitoring from the `/proc` on Ubuntu 14.04 Linux. Android kernel is nothing but a main stream Linux kernel. So all the kernel level functionalities and most of the utilities are available in Linux too.

The implementation of Monitoring module in Linux is completed. The source code can be found on `https://github.com/nikhilgeo/NEOM`

The implementation of Monitoring module is in JAVA. The monitoring module will track all the network connection going in and out of the system.

For tracking the connections the module will periodically parse the `/proc/net/tcp` and `/proc/net/udp` files. The file is read in one go rather than line by line as the files contents are dynamic and created while reading. So
t java.nio.file.Files class is used to do non-blocking read from the file using the `Files.readAllBytes()` function. The code for reading is shown below.

```
public String readFile_InOneGO(String path) {
String fileContents = "";
Charset encoding = Charset.forName("UTF-8");
try {
byte[] encoded = Files.readAllBytes(Paths.get(path));
return new String(encoded, encoding);
    }
catch (Exception ex) {
System.out.println("Error in Utilities.readFile_InOneGO : " +
ex.getMessage()); }
return fileContents;
}
```

Once the file contents are read they are parsed to extract the connection detail. For this first the entire data that is read from the file as a string is splitted into lines and stored in *String* array. Then reach line is iterated and splitted where ever space occurs, Figure 3.2 shows the content structure of the `/proc/net/tcp` files. Each line in the `/proc/net/tcp` file represents a single connections. Once each line is splitted by space we get remote address, local address which is a combination of IP address and port number. Apart from this we get the port number and inode number. Code below shows the splitting and how each attribute is extracted.

```
private static void processFile(String Connections, String timestamp) {
 System.out.print(Connections);
 ArrayList<NW_Interfaces> nw_inter_list = new
 ArrayList<NW_Interfaces>();
 List<String> pid_pname = new ArrayList<String>();
 String tcpConArray[] = Connections.split("\n");
 //Split to line by line
 for (int tcpConIndex = 1; tcpConIndex < tcpConArray.length;
 tcpConIndex++) {
  StringBuilder log_per_con = new   StringBuilder(timestamp);
  tcpCon = tcpConArray[tcpConIndex];
  String tcpIndividualConn[] = tcpCon.split("\\s+");
  String local_SocketHEX[] = tcpIndividualConn[2].split(":");
  //split Local IP:Port
  local_IP = utilities.little_endianIP_to_decimal(
  local_SocketHEX[0]);
  local_port = utilities.hex_to_decimal(local_SocketHEX[1]);
  String rem_addressHEX[] = tcpIndividualConn[3].split(":");
  //split Remote IP:Port
  rem_IP = utilities.little_endianIP_to_decimal(rem_addressHEX[0]);
  rem_port =   utilities.hex_to_decimal(rem_addressHEX[1]);
  protocol = utilities.get_protocol_name(rem_port);
  conStatusCode = Integer.parseInt(utilities.hex_to_decimal
  (tcpIndividualConn[4]));}}
```

The remote address and the local address in the `/proc/net/tcp` is represented in the little endian form we use the custom function `little_endianIP_to_decimal()`. To convert the little endian to decimal we have to invert the little endian and then change from hex to decimal. The function `little_endianIP_to_decimal()` is as below.

```
public String little_endianIP_to_decimal(String little_endian_IP) {
    String octect, IPAddr;
    int decimal;
    StringBuilder reversed_IP = new StringBuilder("");
    for (int index = little_endian_IP.length() - 1; index >= 0;
    index = index - 2) {
        octect = little_endian_IP.substring(index - 1, index + 1);
        decimal = Integer.parseInt(octect, 16);
        reversed_IP = reversed_IP.append(decimal);
        reversed_IP = reversed_IP.append(".");
    }
    IPAddr = reversed_IP.toString();
    return IPAddr.substring(0, IPAddr.length() - 1);
}
```

Once we have the the attributes of the connection next main thing is to find out the the owner process of each connection for this the `inode` details in each connection comes in handy. But to use the `inode` number the system should have a map of all the `inode` numbers and process that owns them. This details is kept in a hashtable.

From the look up in the hashtable we could get the details of which process owns a connection and with the the attributes of connections we intent to collect is complete and can now be logged into file. The log will look like below. Each attribute is separated by a separator pipe symbol, so that it will be easy to parse the logs and extract the information needed.

```
2015-04-0810:13:37:853|192.168.0.112|40869|173.194.38.183
|443|TCP_SYN_SENT|inode:53310|pid:2555|com.android.chrome
|protocol:HTTPS|wlan0:|Recv:44942087|Trns:265139937

2015-04-0810:13:37:853|192.168.0.112|35349|54.225.64.222
|443|TCP_SYN_SENT|inode:70387|pid:2555|com.android.chrome
|protocol:HTTPS|wlan0:|Recv:44942087|Trns:265139937

2015-04-0810:13:37:853|192.168.1.3|39802|179.60.192.3
|443|CLOSE_WAIT|inode:68557|pid:13061|com.piazza.android
|protocol:HTTPS|wlan0:|Recv:21942087|Trns:165139937
```

Each item in the log is separated by a pipe symbol so that it is easy for parsing the logs and extracting information from them.

Chapter 4

Related Work

As the mobile devices have resource and power constraints all the heavy computing of monitoring and detecting network traffic anomaly could not be done on the device and was shipped out to the cloud. The cloud-based Intrusion Detection proposed by Houmansadr et al. (2011) runs an Emulator on the cloud. The incoming traffic to the devices is duplicated and forwarded to the emulation platform that is running on the cloud. Real-time emulation of the device traffic is done on the cloud servers which is powerful enough to handle the device loads. The user input is replicated in real time on the cloud servers to have very limited bandwidth requirements. If any misbehavior is detected, countermeasure actions are sends to a non-intrusive software agent that is running on the device. This software agent which is in charge of only carrying out the received actions.

An attempt to bring light weight intrusion detection system, SNORT to mobile platform is LIDS by Kou and Wen (2011). Basically the SNORT was optimized and made light weight to run on mobile device. The optimizations include improving the key pattern matching algorithm and implementation of the CIDF model i.e. Event generators, Event Analysis, Response Unit and Event Database. The design of LIDS includes four parts: Packet capture, Snort, Output control and Storage. The packet capture is done using `libcap`, the Ruleset and Log Files are made into standard sqlite database or text files. The Rule Header is a combination of [rules action, protocol, source/destination IP, subnet mask, source/ destination port] + Rule Option [alert message and signature of exception packets]. As part of LIDS is an Android application for users to look up the detection results and update ruleset easily. Matching of packet strings against signature patterns is the most time and resource consuming operation and was optimized using the WMN or improved WM (Wu-Manber) [LAN and WANG (2008)] multi-pattern matching algorithm.

Apart from IDS, firewalls are also ported to mobile platform to make the network filtering possible, one such attempt is the use of ontology based firewall for privacy protection on Android device proposed by Vincent et al. (2011). The proposed semantic firewall takes its decisions on the basis of a set of privacy protection rules grounded on two ontologies identity of mobile phones users and privacy policies. Ontologies are used to represent both the concepts of identity and personal data and also to model privacy policies. To explain how the proposed firewall responds to privacy threat, a basic scenario using policy rules expressed in SWRL are created. Before building a semantic firewall preventing privacy breaches, record of all the data that need to be protected must be collected. The global architecture is grounded on a smart phone ontology written in OWL that includes two ontologies: the ontology designed to represent privacy policies and the ontology of the digital identity stored in the mobile. The firewall is in charge of populating the smart phone ontology with the individual's corresponding to the specific request. When a request is made, the firewall processes the request by calling the description logic reasoner (DL reasoner) in charge of inferences and decides the access.

Another approach for using ontology based firewall by Vincent et al. (2011) in smart phones is based on a set of privacy rules and application of description logic reasoning to decide whether access to private data by a given application may be authorized or not. Proposition consists a local application implemented on an Android device. This application relies on the JENA API [Apache (Apache)] to handle OWL ontologies. The ontology file is first loaded as a JENA object that will be used by the firewall. The application simulates a request made by a specific application to access some data and creates the corresponding individuals (instance of objects) within the ontology. The firewall then launches reasoners in order to classify the ontology regarding a set of privacy rules and finally the firewall takes its decision. In JENA API three inference modes are available: forward chaining, backward chaining and mixed chaining. None of them led to significant performance gain but reduces the loading time. To improve the loading time, using a local database instead of the OWL ontology file was also tried. This solution was also unsatisfactory as there is an incompatibility between JENA and SQLite on Android. Attempts to optimize the local firewall proved unsuccessful, and an alternative approach a distant server was used. This server stores the ontology and takes its decisions while only a local database is maintained to store the most recent results on the smart phone. With this approach, the decision time is only function of the connection bandwidth. The server side is implemented as a JAVA servlet accessible via HTTPS.

Device centric firewalls offered a huge overhead to the device performance an so cloud based firewalls made way to the Android platform. Once such proposal by Cai et al. (2013) makes use of cloud based firewall for filtering the SMS and calls on mobile device. USes a Client/Server structure where the server side collects, processes and shares the information while client sides receive, update information and intercept locally. The heavy task of process and analysis is done on the server side while the client sides only need to receive and update information, do interception and filtration work based on the updating data. Call filtration module in clients is responsible for the interception of offensive or harassing calls and uploading information which stored in local blacklist database of the clients. When a call comes, first, this number will be checked in the local black-list database and if the phone number is not found the normal answering can be done or otherwise the call will be intercepted. Data transmission system is designed for communication transmission between client base and the server, mainly supporting for push and collect service of cloud server [Cloud to Device Push Messaging on Android: a Case Study,]

TrustDroid [Bugiel et al. (2011)] attempts to achieve data privacy and security by grouping of apps into domains and isolating them from sharing the data. The firewall module that is part of the system, allows network data to be only read by a particular domain, or enables basic context-based policies such as preventing Internet access by untrusted applications while an employee is connected to the companys network. Android fails to enforce isolation at the network-level which would enable the deployment of basic context-aware policy rules. There is no means to deny Internet access for untrusted applications while the employee is connected to the company network. Also by default Android has no means to group applications and data into domains, where in according to TrustDroid a context or domain compromises of a set of applications and data belonging to one trust level (e.g., private, academic, enterprise, department, institution, etc.). TrustDroid provides application and data isolation by controlling the main communication channels in Android, namely IPC (Inter-Process Communication), files, databases, and socket connections. Applications are colored based on user data (stored in shared databases) based on a (lightweight) certification scheme which can be easily integrated into Android. Based on the applications colors, TrustDroid organizes applications along with their data into logical domains.

TrustDroid monitors all application communications, access to common shared databases, as well as file-system and network access, and denies any data exchange or application communication between different domains. We modified the standard Android kernel firewall to enable network filtering Network Traffic and socket control. This allows us to isolate network traffic among domains and enables the deployment of basic context-based policies for the network traffic. TrustDroid, assume three trust levels for applications: 1) pre-installed system apps 2) trusted third party apps 3) untrusted third party apps, which are retrieved from Android Market. While trusted and untrusted apps must be isolated from each other, system applications usually have to be accessible by all installed applications in order to preserve correct functionality of those applications and sustain both transparency and legacy compliance of our solution.

WallDroid proposed by Kilinc et al. (2012) is another cloud based firewall solution which uses the VPN to filter the traffic from the device. WallDroid also have an app from which users are allowed to set the filter rules. Key components used by the WallDroid include VPN technologies like the Point to Point Tunneling Protocol (PPTP) and the Android Cloud to Device Messaging Framework (C2DM). Solution is based on the cloud keeping track of millions of applications and their reputation (good, bad, or unknown) and comparing traffic flows of applications with a list of known malicious IP servers. The architecture consist of main three components a VPN Server, the WallDroid Application server and a WallDroid App. WallDroid app can be considered as an Android Firewall application but with some extra functionality. Anti-malware solutions require the user to decide what to do, for applications which are not clearly safe and not clearly malware but the users are not the best to make such kind of decisions so the user is required to choose the security policy. The security policy examples are High Security, Medium Security, and Low Security.

Chapter 5

Evaluation

5.1 The Goal of Network Monitoring

An Android device is a fully networked device, compared to a desktop PC. Not surprisingly, 25% of smart phone users use mobiles to go online rather that a computer. From a device used only for telephony and SMS, Internet connectivity helped mobiles evolve into much needed devices in our daily lives. Because of its usage, the number and duration of network connections is also considerably higher.

At the moment (June 2015), Android does not have any infrastructure that can monitor all the network connections. Merely monitoring does not provide security but network ombudsman let users extract meaningful information regarding the network connections, optionally based on a time line. Network ombudsman provides a provision for users to carry out real-time analysis of all active network connections and processes. As the logs are saved and uploaded on to the cloud server it helps the users to do postmortem analysis also. So users can look into past device activity based on the network connections and process details. This mainly helps users to carry out incident investigation by oneself, by establishing a time-line of un-usual activity and looking into network connections made by the devices at that time and pin pointing the process that made malicious activity.

5.2 Comparison with Previous Work

Houmansadr et al. (2011) proposed filtering of malicious traffic by forwarding the traffic to the emulation platform running on the cloud. Then on the cloud detailed analysis of network traffic is carried out. As the traffic from the device need to be diverted to the cloud server it can be considered as a serious infringement on the user privacy. Because the cloud server will be able to see the data transmitted if the connection is not encrypted. Network ombudsman do not redirect the traffic to any central point of evaluation. The logs uploaded only have the connection attributes and not any connection data.

WallDroid [Kilinc et al. (2012)] is a firewall solution which also uses VPN to track apps and their reputation and compare traffic flows from apps with a list of known malicious IP servers. Here also the analysis happens on the cloud server and so cloud server can see all the unencrypted data transmitted that is transmitted from the device. Network ombudsman do not redirect the traffic and also neither of these have dynamic filter rules that changes based on a preset context.

NoRoot Firewall app available on Google Play store uses a VPN server to redirect the traffic and filter them at the VPN server. Users can set the rules in the app and these rules will be updated to the VPN server and will be saved device wise. Note that when traffic on the device is redirected to the VPN server for analysis the privacy of the user is compromised. Network log is another app that is also available on Play store which uses manual black/white list. From the app, users will be provided with the option to select black list or white list based filter. Apps can be added to the list and selected to block on WiFi or data connection. But none of these app does not have context based filtering or IP or domain based targeted filtering.

Kou and Wen (2011) propose porting `snort` IDS to mobile platform, but `snort` is rather heavy for a normal day to day mobile device and does not incorporate app categorization or context-awareness into filtering. Also more than detection Network Ombudsman concentrates more on monitoring and

TrustDroid [Bugiel et al. (2011)] is another firewall solution with context awareness to prevent data access between apps at diferent trust levels, for example private apps and corporate apps. The apps where categorized into different trust levels, and each trust level was isolated from another. TrustDroid have context awareness to switch between the trust levels. We borrowed the context switching from

TrustDroid but used it to dynamically switch the network filtering rules which was not implemented in TrustDroid.

Cai et al. (2013) proposed a white list and black list based solution for filtering the SMS and voice calls. The black list and white list are stored on a central server. Network Ombudsman also has black/white list based filter rules, but we always keep the rules, updated, on the mobile device and users have full control over the device. Also the filter rules are dynamically updated by the cloud server.

5.3 Impact on Battery Consumption

Network ombudsman is designed to smartly utilize the device battery. Both the app and background service keep track of the battery level by listening to the `ACTION_BATTERY_LOW` and `ACTION_BATTERY_OKAY` intent broadcasts. If the battery is low then the log upload is paused to conserve the battery.

The `BatteryManager` broadcasts an action when the device is plugged in or unplugged. `ACTION_POWER_CONNECTED` intents are listened to via broadcast receivers and if the device is connected to power then the log upload is resumed. The normal upload frequency is once every hour.

`ConnectivityManager` is used to check the Internet connectivity status, also `CONNECTIVITY_ACTION` is broadcast by the `ConnectivityManager` each time connectivity details change, e.g., switching between WiFi and data connection etc.

Before log uploads, network connectivity is checked and made sure that the device is connected to network with the function `getActiveNetworkInfo()` from the `ConnectivityManager`. Also upload service will listen to the broadcast intend `CONNECTIVITY_ACTION` which is triggered when ever there is any change in connection details. Listening to `CONNECTIVITY_ACTION` broadcast and rechecking the network status will help to identify when devices goes online.

5.4 Contribution to Lag

The app is designed to contribute “not too much lag”. All the user visible interfaces are worked out by the user interface thread or UI thread. Too much time consuming process or work on the UI thread will delay the rendering of visible components will contribute to the lag.

To reduce the lag all the heavy time consuming operations are taken off the UI thread. The two heavy computation that is taken off the UI thread are:

- In the *Process monitor* the `getJavaProcessPID()` function that is used to fetch all the running process, the gather the memory consumption details of the same.
- In the *Network monitor* the `getConnections()` function which is used to get all the active network connection, get the data transmitted and interface used for the connection.

Separate threads for above two functions were created using `Thread` class. The `run()` function in the `Thread` class was overriden to implement call these functions at regular interval as both Process monitor and Network monitor provided real-time process and network monitoring.

5.5 Local Storage

Mobile devices has become a key instrument for personal entertainment and work assistant. User are so conscious about the storage, apps consumes because they want to store more and more personal/work related files like music, videos, documents etc. Document editing and view desktop applications like Windows Office suite, Open Office, Adobe reader etc had made there way into Android platform and so now users carry around documents much more than earlier times. Because of all this one of the most precious resource on a mobile device is storage space.

Network ombudsman treats device storage very critically. The logs generated by network ombudsman service are rotated to the cloud server. Once the logs are uploaded into the server, the uploaded once are deleted from the log files. This will keep the size of the log files in check. Also the log upload interval can be configured via the settings, this can be used to clear out the logs fast.

In case of apps like Network log the size of the log file easily climbs to 4MB with in few minutes of running but in case of Network Ombudsman as the logs are rotated periodically to the cloud server.

If the cloud server is unavailable in the case of prolonged network unavailability, the logs are compressed till next successful upload, there by saving the space. If these accumulate beyond a threshold, a notification will be shown. Note that if there is no network activity no logs will be created.

5.6 Cloud Upload

The framework service keep on monitoring the network connections at a fixed configurable interval. This configuration can be done via the `settings` in the NEOM app. All the monitored connections are logged into flat files and uploaded into the cloud server. This upload is implemented to look into the system status so that the resources are wisely utilized. Upload service listens to the battery status broadcast that will be triggered off when the battery is low and pauses the upload until battery health is good. Also the upload frequency will be increased and the upload interval will be decreased when the device is plugged in to an charging source. Mainly `ACTION_BATTERY_LOW` and `ACTION_POWER_CONNECTED` intents are looked upon for this.

5.7 Status of the Implementation

5.7.1 Source code repository links

1. NEOM apk: https://github.com/nikhilgeo/NEOM_App
2. Network Monitor POC on Linux: https://github.com/nikhilgeo/NEOM

5.7.2 Network Monitor POC on Linux sloc count

File name	sloc
src/com/nikhilgeo/Inode_uid_process_Maping.java	184
src/com/nikhilgeo/NEOM.java	108
src/com/nikhilgeo/NW_Interfaces.java	67
src/com/nikhilgeo/Utilities.java	77
Total sloc	436

5.7.3 NEOM APK sloc count

File name	sloc
NEOM/AndroidManifest.xml	36
NEOM/res/layout/activity_main.xml	34
NEOM/res/layout/fragment_filter.xml	30
NEOM/res/layout/fragment_filter_home.xml	26
NEOM/res/layout/fragment_monitor.xml	17
NEOM/res/layout/fragment_navigation_drawer.xml	9
NEOM/res/layout/fragment_nwmonitor.xml	15
NEOM/res/layout/fragment_set_context.xml	30
NEOM/res/layout/interface_table_row.xml	33
NEOM/res/layout/nw_list_item.xml	153
NEOM/res/layout/nwfilter_row.xml	35
NEOM/res/layout/process_list_item.xml	105
NEOM/res/layout/setcontext_row.xml	23
NEOM/res/layout/table_row.xml	59
NEOM/src/com/nikhil/neom/ExecuteCMD.java	38
NEOM/src/com/nikhil/neom/FilterFragment.java	273
NEOM/src/com/nikhil/neom/Filter_home_Fragment.java	88
NEOM/src/com/nikhil/neom/LogCat.java	17
NEOM/src/com/nikhil/neom/MainActivity.java	96
NEOM/src/com/nikhil/neom/Model_Apps.java	7
NEOM/src/com/nikhil/neom/Model_Connection.java	11
NEOM/src/com/nikhil/neom/Model_Process.java	18
NEOM/src/com/nikhil/neom/MonitorFragment.java	331
NEOM/src/com/nikhil/neom/NWMonitorFragment.java	352
NEOM/src/com/nikhil/neom/NavigationDrawerFragment.java	242
NEOM/src/com/nikhil/neom/SetContextFragment.java	81
NEOM/src/com/nikhil/neom/Utilities.java	232
NEOM/src/com/nikhil/neom/WifiReceiver.java	34
NEOM/src/com/nikhil/neom/iptablesDBContract.java	21
NEOM/src/com/nikhil/neom/neomDbHelper.java	38
Total sloc	2484

Chapter 6

Conclusion

Network ombudsman addresses network filtering and monitoring on an Android device treating it as a full scale networked computer system. The NEOM app built as a part of Network Ombudsman provide the device owner with the ability for incident investigation on his own. Our log files are rotated and uploaded to a cloud storage server so that space on the devices does not become a constraint. At any point the user privacy is not breached as the net- work traffic is not redirected to any central server for process. Our app enables custom queries on the stored log data which can give users deep insight into the con- nections made by apps installed on their devices behind their backs.

6.1 Future Work

In the current implementation of network monitoring module, the port numbers are used to identifying the services based on `/etc/services`. Instead identification by inspection of network traffic can be implemented. This is important to identify the services correctly because services can be configured to run on non-standard ports also. But the feasibility of heuristics based implementation on low power device like Android has to be studied. Also another aspect that can be considered as a future expansion in Network Ombudsman is implementation of `nftables` instead of `iptables`. `nftables` have better flexibility in specifying filtering rules, rules re-usability, and in general are much more efficient and faster than `iptables`.

Chapter 7

Appendix

7.1 Function to read and process the /proc/net files in NEOM Service

```
private static void processFile(String Connections, String timestamp) {
    System.out.print(Connections);
    ArrayList<NW_Interfaces> nw_inter_list =
    new ArrayList<NW_Interfaces>();
    List<String> pid_pname = new ArrayList<String>();
    String tcpConArray[] = Connections.split("\n");
    inode_uid_process_maping.
    print_HashTable_getInode_pid_pname_mapping();
    for (int tcpConIndex = 1; tcpConIndex < tcpConArray.length;
    tcpConIndex++) {
        StringBuilder log_per_con = new StringBuilder(timestamp);
        tcpCon = tcpConArray[tcpConIndex];
        String tcpIndividualConn[] = tcpCon.split("\\s+");
        String local_SocketHEX[] = tcpIndividualConn[2].split(":");
        local_IP = utilities.little_endianIP_to_decimal(local_SocketHEX[0]);
        local_port = utilities.hex_to_decimal(local_SocketHEX[1]);
        String rem_addressHEX[] = tcpIndividualConn[3].split(":");
        rem_IP = utilities.little_endianIP_to_decimal(rem_addressHEX[0]);
        rem_port = utilities.hex_to_decimal(rem_addressHEX[1]);
        protocol = utilities.get_protocol_name(rem_port);
        conStatusCode = Integer.parseInt
        (utilities.hex_to_decimal(tcpIndividualConn[4]));
        tcpConStatus = tcp_status.values()[conStatusCode].toString();
        UID = tcpIndividualConn[8];
        inode = tcpIndividualConn[10];
        if (!inode.equals("0")) {
            pid_pname = inode_uid_process_maping.
            pid_processName_lookup(inode);
            if (pid_pname != null) {
                pid = pid_pname.get(0);
                processName = pid_pname.get(1);
                log_per_con.append("|" + local_IP + "|" + local_port +
                "|" + rem_IP + "|" + rem_port + "|" + tcpConStatus +
                "|inode:" + inode + "|pid:" + pid + "|" + processName +
                "|protocol:" + protocol);
                nw_inter_list = nw_interfaces.get_data_transfer(pid);
                for (NW_Interfaces item : nw_inter_list) {
                    log_per_con.append("|" + item.getInterface_Name() +
                     "| Recv:" +
                    item.getReceived_bytes() + "| Trns:" +
                    item.getTransmitted_bytes());
                }
            } //if (pid_pname != null)
            System.out.println(log_per_con.toString());
        } //if(inode == "0")
    }
}
```

7.2 Broadcast receiver to handle the WiFi context based filter rules in NEOM APK

```
public void onReceive(Context context, Intent intent) {
    this.context = context;
    String action = intent.getAction();
    if (action.equals(WifiManager.NETWORK_STATE_CHANGED_ACTION)) {
        WifiManager manager = (WifiManager) context
            .getSystemService(Context.WIFI_SERVICE);
        NetworkInfo networkInfo = intent
            .getParcelableExtra(WifiManager.EXTRA_NETWORK_INFO);
        NetworkInfo.State state = networkInfo.getState();
        if (state == NetworkInfo.State.CONNECTED) {
            String connectingToSsid =
            manager.getConnectionInfo().getSSID()
                        .replace("\"", "");
            Toast.makeText(context, "Connected to " + connectingToSsid,
            Toast.LENGTH_SHORT).show();
            check_applyRules(connectingToSsid);
            }
        if (state == NetworkInfo.State.DISCONNECTED) {
            List<String> blkdSSID = new ArrayList<String>();
            neomDbHelper mDbHelper = new neomDbHelper(context);
            SQLiteDatabase db = mDbHelper.getReadableDatabase();
            blkdSSID = getblkdSSID();
            for (String ssid : blkdSSID) {
                ArrayList<String> block_rules_arlist =
                new ArrayList<String>();
                block_rules_arlist = getRulesForSSID(ssid);
                ArrayList<String> block_rules_arlisttoDel =
                new ArrayList<String>();
                for (String rule : block_rules_arlist) {
                    block_rules_arlisttoDel.add(rule.replace("-A", "-D"));
                }
                String block_rules_ar[] = block_rules_arlisttoDel
                    .toArray(new String[block_rules_arlisttoDel.size()]);
                ExecuteCMD executeCMD = new ExecuteCMD();
                executeCMD.RunAsRoot(block_rules_ar);
                ContentValues cv = new ContentValues();
                cv.put(ssidInfo.COLUMN_NAME_ACTIVE, "N");
                String whereClause = ssidInfo.COLUMN_NAME_SSID + "=" + "?";
                db.update(ssidInfo.TABLE_NAME, cv, whereClause,
                    new String[] { ssid });
                db.close();
                Toast.makeText(
                    context, Integer.toString(block_rules_ar.length)
                    + "Rules for  " + ssid + " Removed..",
                    Toast.LENGTH_SHORT).show();
                }
            }
        }
```

7.3 Function to execute privileged commands from the NEOM APK

```
public String RunAsRoot(String[] cmds) {
    String cmdOutput = null;
    String read_line = null;
    try {
        Process suProcess = Runtime.getRuntime().exec("su");
        DataOutputStream os = new DataOutputStream(
            suProcess.getOutputStream());
        BufferedReader stdInput = new BufferedReader(new InputStreamReader(
            suProcess.getInputStream()));
        for (String tmpCmd : cmds) {
            if (tmpCmd != null)
                os.writeBytes(tmpCmd + "\n");
            }
        os.writeBytes("exit\n");
        os.flush();
        while ((read_line = stdInput.readLine()) != null) {
            cmdOutput = cmdOutput + read_line + "\n";
        }
    } catch (Exception e) {
      Log.i("NEOM:", e.getMessage());
      cmdOutput = "Error";
    }
    return cmdOutput;
}
```

7.3.1 Function to fetch all the installed apps

```
void getAllInstalledApps() {
 try {
  PackageManager pm = getActivity().getPackageManager();
  List<ApplicationInfo> installedApps = pm
   .getInstalledApplications(PackageManager.GET_META_DATA);
  for (ApplicationInfo appInfo : installedApps) {
   Model_Apps apps = new Model_Apps();
   apps.appName = appInfo.loadLabel(pm).toString();
   apps.uid = appInfo.uid;
   apps.icon = appInfo.loadIcon(pm);
   filter_applist.add(apps);
  }
 } catch (Exception ex) {
  Log.w("NEOM:", ex.toString(), ex);
}}
```

7.3.2 Function to write rules to the SQlite DB

```
long writeRulestoDB(int uid, String rule) {
 neomDbHelper mDbHelper = new neomDbHelper(getActivity());
 SQLiteDatabase db = mDbHelper.getWritableDatabase();
 ContentValues values = new ContentValues();
 long newRowId;
 if (wifiSSID != null)
  {
   // Create a new map of values, where column names are the keys
   values.put(iptblruleSSID.COLUMN_NAME_UID, uid);
   values.put(iptblruleSSID.COLUMN_NAME_RULE, rule);
   values.put(iptblruleSSID.COLUMN_NAME_SSID, wifiSSID);
   newRowId = db.insert(iptblruleSSID.TABLE_NAME, null,   values);
  } else { // Set the manual black based rules to DB
   // Create a new map of values, where column names are the keys
   values.put(iptblrule.COLUMN_NAME_UID, uid);
   values.put(iptblrule.COLUMN_NAME_RULE, rule);
   newRowId = db.insert(iptblrule.TABLE_NAME, null, values);
}
 db.close();
 return newRowId;
}
```

7.3.3 Function to read all filter rules from SQlite DB

```
private List<String> getAllRulesFrmDB() {
 List<String> uidLst = new ArrayList<String>();
 List<String> uidLstNoSSIDtmp = new ArrayList<String>();
 neomDbHelper mDbHelper = new neomDbHelper(getActivity());
 SQLiteDatabase db = mDbHelper.getReadableDatabase();
 String[] projection = { iptblrule.COLUMN_NAME_UID };
 uidLstNoSSIDtmp = getAllRulesFrmDBWithOutSSID();
 if (wifiSSID != null) {
  String whereClause = iptblruleSSID.COLUMN_NAME_SSID + "=" + "?";
  Cursor cursor = db.query(iptblruleSSID.TABLE_NAME,
  projection, // The columns to return
  whereClause, // The columns for the WHERE clause
  new String[] { wifiSSID },// The sort order
  null, null, null);
  if (cursor.moveToFirst()) {
   do {
    String uid = cursor.getString(cursor
   .getColumnIndexOrThrow(iptblruleSSID.COLUMN_NAME_UID));
    if (uidLstNoSSIDtmp.contains(uid))
     continue;
     uidLst.add(uid);
   } while (cursor.moveToNext());
  }
  if (cursor != null && !cursor.isClosed())
  cursor.close();
  db.close();
  }
  else {
   uidLst = uidLstNoSSIDtmp;
  }
  return uidLst;
}
```

7.3.4 Function to delete filter rules from SQlite DB

```
private void flushRules() {
 ExecuteCMD executeCMD = new ExecuteCMD();
 List<String> uidinDBLst = getAllRulesFrmDB();
 ArrayList<String> del_rules_arlist = new
 ArrayList<String>();
 for (String blkuid : uidinDBLst) {
  String rule = "iptables -D OUTPUT -m owner --uid-owner " +
  blkuid + " -j DROP";
  del_rules_arlist.add(rule);}
  String del_rules[] = del_rules_arlist
 .toArray(new String[del_rules_arlist.size()]);
```

```
  executeCMD.RunAsRoot(del_rules);
  neomDbHelper mDbHelper = new neomDbHelper(getActivity());
  SQLiteDatabase db = mDbHelper.getWritableDatabase();
  db.execSQL("delete from " + iptblrule.TABLE_NAME);
  if (wifiSSID != null) {
   db.execSQL("delete from " + iptblruleSSID.TABLE_NAME + "
   where " + iptblruleSSID.COLUMN_NAME_SSID + "='" +
   wifiSSID + "'");
   db.execSQL("delete from " + ssidInfo.TABLE_NAME + " where
   " + ssidInfo.COLUMN_NAME_SSID + "='" + wifiSSID + "'");
  }
 db.close();
 Toast.makeText(getActivity(), "Existing " +
 Integer.toString(uidinDBLst.size()) + " rules deleted",
 Toast.LENGTH_SHORT).show();
}
```

7.4 SQlite DB table schema definition class

```
public class iptablesDBContract {
 public iptablesDBContract() {}
 /* Inner class that defines the table contents */
 public static abstract class iptblrule implements BaseColumns {
  public static final String TABLE_NAME = "iptblrules";
  public static final String COLUMN_NAME_UID = "uid";
  public static final String COLUMN_NAME_RULE = "rule";
  }
 /* Inner class that defines the table contents */
 public static abstract class iptblruleSSID implements BaseColumns {
  public static final String TABLE_NAME = "iptblrulesSSID";
  public static final String COLUMN_NAME_UID = "uid";
  public static final String COLUMN_NAME_RULE = "rule";
  public static final String COLUMN_NAME_SSID = "ssid";
  }
 /* Inner class that defines the table contents */
 public static abstract class ssidInfo implements BaseColumns   {
  public static final String TABLE_NAME = "ssidInfo";
  public static final String COLUMN_NAME_SSID = "ssid";
  public static final String COLUMN_NAME_ACTIVE = "active";
 }
}
```

7.5 Function to read the data transferred from interfaces

```
public ArrayList<NW_Interfaces> get_data_transfer(String pid) {
    String dev_data;
    String interface_details[];
    String interface_details_row[];
    int interface_count;
    ArrayList<NW_Interfaces> nw_interface_list = new
    ArrayList<NW_Interfaces>();
    NW_Interfaces nw_interfaces;
    try {
        dev_data = utilities.readFile_InOneGO("/proc/" + pid + "/net/dev");
        interface_details = dev_data.split("\n");
        interface_count = interface_details.length;
        for (int interface_index = 2; interface_index < interface_count;
        interface_index++) {
        interface_details_row =
        interface_details[interface_index].trim().split("\\s+");
        if (!interface_details_row[9].equals("0") ||
        !interface_details_row[1].equals("0")) {
        nw_interfaces = new NW_Interfaces();
        nw_interfaces.setInterface_Name(interface_details_row[0]);
        nw_interfaces.setReceived_bytes(interface_details_row[1]);
        nw_interfaces.setTransmitted_bytes(interface_details_row[9]);
        nw_interface_list.add(nw_interfaces);
    }
    }
    return nw_interface_list;
    } catch (Exception ex) {
    System.out.println(ex.toString());
    }
    return null;}
```

7.6 Utility class for manipulating the IP address

```
public class Utilities {

    /**
     * TO DO: Find the name of interfaces
     * only one time, put it in some static
     *
     * @param path
     * @return
     */
    public String readFile_InOneGO(String path) {
        String fileContents = "";
        Charset encoding = Charset.forName("UTF-8");
        try {
            byte[] encoded = Files.readAllBytes(Paths.get(path));

            return new String(encoded, encoding);
        } catch (Exception ex) {
            System.out.println("Error in Utilities.readFile_InOneGO : " +
            ex.getMessage());
        }
        return fileContents;
    }

    public String hex_to_decimal(String hex) {
        int decimal;
        decimal = Integer.parseInt(hex, 16);
        return String.valueOf(decimal);
    }

    public String little_endianIP_to_decimal(String little_endian_IP) {
        //System.out.println("little_endian_IP =" + little_endian_IP);
        String octect, IPAddr;
        int decimal;
        StringBuilder reversed_IP = new StringBuilder("");

        for (int index = little_endian_IP.length() - 1; index >= 0;
        index = index - 2) {
            octect = little_endian_IP.substring(index - 1, index + 1);
            //System.out.println("octect" + octect);
            decimal = Integer.parseInt(octect, 16);
            //System.out.println("decimal" + decimal);
            reversed_IP = reversed_IP.append(decimal);
            reversed_IP = reversed_IP.append(".");
        }
        IPAddr = reversed_IP.toString();
        return IPAddr.substring(0, IPAddr.length() - 1);
    }

    public String get_protocol_name(String port) {
        try {
            int port_number;
            port_number = Integer.valueOf(port);
            switch (port_number) {
                case 20:
                case 21:
                    return "FTP";
                case 22:
                    return "SSH";
                case 25:
                    return "SMTP";
                case 53:
                    return "DNS";
                case 80:
                    return "HTTP";
                case 110:
                    return "POP3";
                case 143:
                    return "IMAP";
                case 443:
                    return "HTTPS";
                default:
                    return port;
            }
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            return port;
        }
    }}
```

Bibliography

Apache. Jena Ontology API. https://jena.apache.org/documentation/ontology/. [Online; accessed 7-Feb-2015].

Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. 2011. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 51–62.

Cai, M., Hou, Q., Jing, F., and Ding, Q. 2013. Research of cloud security communication firewall based on android platform. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press.

Forristal, J. 2014. Android fake id vulnerability. Tech. rep., BlackHat US 2014. Aug. https://www.blackhat.com/docs/us-14/materials/us-14-Forristal-Android-FakeID-Vulnerability-Walkthrough.pdf.

Gautam, M. 2013. QUALCOMM MSM Interface.

Google.com. 2015. Google report: Android security 2014 year in review. Tech. rep., Google.com. Apr. https://static.googleusercontent.com/media/source.android.com/en/us/devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf.

Houmansadr, A., Zonouz, S. A., and Berthier, R. 2011. A cloud-based intrusion detection and response system for mobile phones. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 31–32.

Kerrisk, M. 2013. AUDITD System Administration Utilities. http://man7.org/linux/man-pages/man8/auditd.8.html. [Online; accessed 29-May-2015].

Kilinc, C., Booth, T., and Andersson, K. 2012. Walldroid: Cloud assisted virtualized application specific firewalls for the android os. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE, 877–883.

Kou, X. and Wen, Q. 2011. Intrusion detection model based on android. In *Broadband Network and Multimedia Technology (IC-BNMT), 2011 4th IEEE International Conference on*. IEEE, 624–628.

LAN, J.-y. and WANG, Y.-h. 2008. Research snort and improvement boyer-moore algorithmic [j]. *Computer Engineering and Design 9*, 015.

Vincent, J., Porquet, C., Borsali, M., and Leboulanger, H. 2011. Privacy protection for smartphones: an ontology-based firewall. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*. Springer, 371–380.

Vincent, J., Porquet, C., and Oulmakhzoune, I. 2011. Ontology-based privacy protection for smartphone: A firewall implementation. In *International Conference on Secure Networking and Applications (ICSNA)*. 3–pages.