Android Dashboard for Past and Present Processes

DISSERTATION

Submitted by

R.ANUSUYA   CB.EN.P2CYS14002

in partial fulfillment for the award of the degree of

MASTER OF TECHNOLOGY
IN
CYBER SECURITY

TIFAC-CORE IN CYBER SECURITY
AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

AUGUST 2016

Android Dashboard for Past and Present Processes

DISSERTATION

Submitted by

R.ANUSUYA   CB.EN.P2CYS14002

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

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

AUGUST 2016

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING,COIMBATORE -641 112

BONAFIDE CERTIFICATE

This is to certify that this dissertation entitled **"Android Dashboard for Past and Present Processes"** submitted by **R.ANUSUYA (Reg.No : CB.EN.P2.CYS14002)** 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 EXAMINERS

AMRITA VISHWA VIDYAPEETHAM
AMRITA SCHOOL OF ENGINEERING,COIMBATORE
TIFAC-CORE IN CYBER SECURITY

DECLARATION

I, R.ANUSUYA (Reg.No: CB.EN.P2.CYS14002) hereby declare that this final project report entitled **"Android Dashboard for Past and Present Processes"** 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

Abstract

Mobile phones are now indispensable devices. We use our smart phones for all our day to day works. Among all the mobile operating systems, Android is the most widely used. Though we use our Android phone for many activities, we are unaware of the activities happening inside it. A savvy user would like to know which processes use the most memory and with what our apps are communicating. As of now, we do not have any service that monitors the Android OS from within and tells what the running processes are doing and what the old processes were doing.

This thesis contributes to monitoring the Android OS constantly. It presents a dashboard for getting the details of present and past processes. It analyses the process behavior with tools like strace and ltrace. These details are logged and are uploaded to the cloud. The logged data are analysed to provide the provenance of a process. We use Android's API like RunningAppProcessInfo, Activity Manager, etc., to retrieve the details of present running processes. Android dashboard for past and present application, provides the end user with the details of all the happenings inside Android device by analysing the processes. The source code and related documents of this thesis are available in the link https://github.com/anurp/DashboardOfProcess

Keywords: Android, OS Monitor, Android Processes, Strace, ltrace.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As smartphones and tablets become more popular, the operating systems for those devices become more important. Android is such an operating system for low powered devices. Like all operating systems, Android enables applications to make use of the hardware features through abstraction and provides a defined environment for applications. Most Android applications are written in Java and run in virtual machines. For this purpose, Android features the Dalvik/ART virtual machine which executes its own byte code. Android applications make use of advanced hardware and software, as well as local and served data, exposed through the platform to bring innovation and value to consumers. Securing an open platform requires a robust security architecture and rigorous security programs. Android was designed with multi-layered security that provides the flexibility required for an open platform while providing protection for all users of the platform. Android seeks to be the most secure and usable operating system for mobile platforms by re-purposing traditional operating system security controls to protect user data, protect system resources (including the network) and provide application isolation. To achieve these objectives, Android provides these key security features.

- Robust security at the OS level through the Linux kernel.
- Mandatory application sandbox for all applications.
- Secure interprocess communication.
- Application signing.
- Application-defined and user-granted permissions.

1.1 Objective

The objective of the thesis is to satisfy the curiosity a user has in knowing what is running on the Android device and what was running before. This thesis is not meant for any forensic investigation or malware analysis. But can provide significant insight into security.

1.2 Problem Statement

A general deficiency of majority of OS is that it is difficult to see what is going on now, and also what happened in the past. This is the situation with Android OS also. Since Android is an open source operating system, it has gained popularity among developers. So plenty of new apps are developed and is available in the market. Nowadays, people prefer apps over websites, as it can be accessed on the go. Security is a major concern while using apps. Plenty of malicious apps are easily available. We may install it unknowingly. We do not know what these apps are doing inside our device. It would be useful if we have a system that constantly monitors all the present and past processes. As of now, we have only apps for monitoring the current running process. *Android Dashboard for Past and Present Processes* comes up with a monitoring tool for the present and past processes and works at the kernel level.

1.3 Organization

This thesis is divided into six chapters.

Chapter 2 provides all needed background knowledge to understand OS processes and applications. It gives an overview of Android OS, Processes, Applications, System Calls and Utility tools like strace and ltrace. Chapter 3 discusses the architecture, methodology, and implementation of getting the provenance of a present and past processes. Chapter 4 discusses the apk developed, the features it has and provides snapshots of the application. Chapter 5 discusses in detail the methods used for getting the results. Chapter 6 discusses the work and papers by others, related to this thesis. Chapter 7 discusses the evaluation of the thesis. The overhead of the thesis and how this thesis could have been done in an efficient method, are discussed here. Chapter 8 discussed the conclusion and future work Chapter 9 has the appendix.

Chapter 2

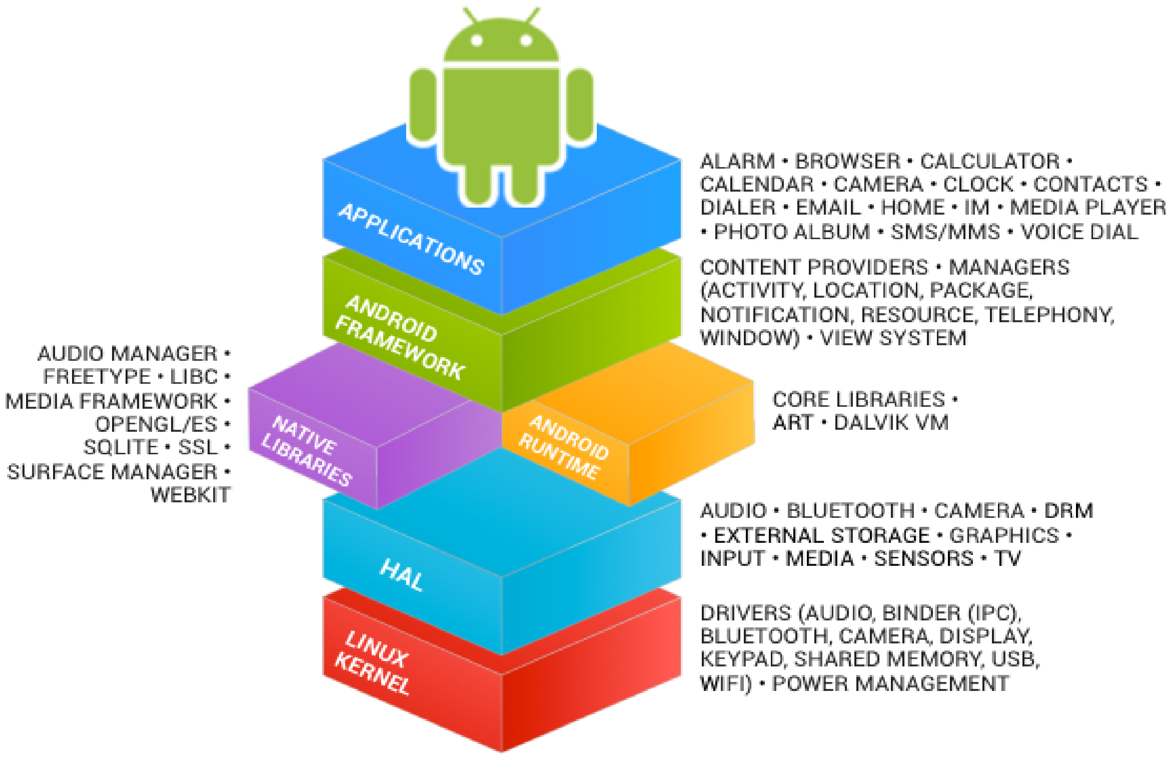
Background

2.1 Android OS

2.1.1 Structural Overview

The Android software stack as shown in Figure 2.1 can be subdivided into five layers: The kernel and low-level tools, native libraries, the Android Runtime, the framework layer and on top of all the applications The kernel in use is a Linux 3.18.10(used in Marshmallow) series kernel, modified for special needs in power management, memory management, and the runtime environment. Right above the kernel run some Linux typical daemons like bluez for Bluetooth support and wpa supplicant for WiFi.

As Android is supposed to run on devices with little main memory and low powered CPUs, the libraries for CPU and GPU intensive tasks are compiled to device optimized native code. Basic libraries like the libc or libm were developed especially for low memory consumption and because of licensing issues on Android. In this layer, the surface manager handles screen access for the window manager from the framework layer. Opposing to other frameworks, the media framework resides in this layer, as it includes audio and video codecs that have to be heavily optimized. The Android Runtime consists of the Dalvik virtual machine and the Java core libraries. The Dalvik virtual machine is an interpreter for byte code that has been transformed from Java byte code to Dalvik byte code. Dalvik itself is compiled to native code whereas the core libraries are written in Java, thus interpreted by Dalvik/ART. Frameworks in the Application Framework layer are written in Java and provide abstractions of the underlying native libraries and Dalvik capabilities to applications. Android applications run in their own sandboxed Dalvik VM and can consist of multiple components: Activities, services, broadcast receivers and content providers. Components can interact with other components of the same or a different application via intents.


APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

[Gurung, 2015]

Figure 2.1: Android OS Architecture

2.2 Processes in Linux and Android

2.2.1 What is a Process?

A process is a program in execution and is meant to carry out tasks within the operating system. A process is a run time volatile entity created by the system call exec().

2.2.2 Difference between Program and Process

We cannot use *Process* and *Program* interchangeably. A program is a file containing algorithm expressed in some suitable programming language or notation. It does not execute and just resides on the disk. Hence, it is a passive entity. Whereas a process has a lifecycle. It is born, runs for some time and then dies and hence it is an *active* entity. A program can be executed by multiple persons, each running a copy of the same program file, but each person is executing a different process. It can be said that a program is only a part of a process.

2.2.3 Properties of a Process

A process may be created by other processes through system calls like fork or spawn. The process which creates a process is the *Parent* process and the process created is called the *Child* process. Every process will have a parent process except the very first process created: *init*. The parent process is expected to outlive the child process. Every process will be identified using the process identifier *PID*. It is assigned to the process by the OS. Usually, it will be 1+ the last born process pid. The parent PID will also be associated with a process. The PID of init process is 1. Every process has an owner.

A process will use many system resources, during its lifetime. It will use the CPU to run its instructions and the system's physical memory to hold its data. It will open, access, modify or create files within the filesystems and may directly or indirectly use other physical devices or resources in the system.

2.2.4 Process in Memory

Process memory has four sections.

- The text section has the compiled program code, read from the non-volatile storage when the program is launched.
- The data section has global and static variables which are allocated and initialized prior to executing main.
- The heap is for dynamic memory allocation and is managed via calls to new, malloc, free, delete, etc.
- The stack is used for local variables. The space in the stack is freed when the variables go out of scope.

When processes are swapped out of memory and later restored, additional information must also be stored and restored. Important among them are the program counter and the value of all program registers.

2.2.5 Process Control Block

For every process, there is a Process Control Block, PCB, which stores the following details.

- Process State - State of the process such as running, waiting,.
- Process ID, and parent process ID.
- CPU registers and Program Counter - They are saved and restored when swapping processes in and out of the CPU.
- CPU-Scheduling information - Contains information like priority information and pointers to scheduling queues.
- Memory-Management information - Contains page tables or segment tables.
- Accounting information - time consumed by kernel CPU, account numbers, limits, etc.
- I/O Status information - Devices allocated for this process, opened file tables, etc.

2.2.6 Process Vs Threads

- Processes are disjoint from each other. Their address spaces are disjoint. They are unaware of each other. OS schedules their execution.
- A thread is part of a process. Several threads share a process and the address space is overlapping. The language (eg., C, JAVA) runtime schedules the execution.

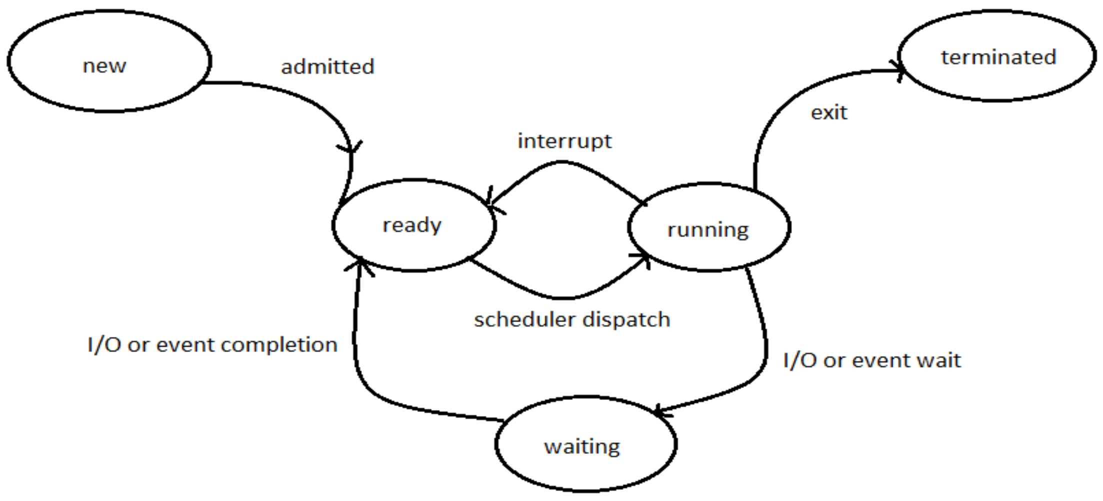


Figure 2.2: Process States

2.2.7 Process States

Each process may be in any one of the five states mentioned below.

- New - The process is in the initial stage of being created.
- Ready - The process has all the resource available and is ready to execute, but the CPU is not working on the instructions of this process.
- Running - The CPU is working on the instructions of this process.
- Waiting - This process is not running at the moment, as it is waiting for some other resources for its job. For example, it may wait for a keyboard input, interprocess message or for a child process to finish.
- Terminated- The process has completed.

2.2.8 Processes in Android

In Android, each application gets a unique Linux user ID. The system sets permissions for all the files in an application so that only the user ID assigned to that application can access them. Each process has its own Dalvik/Art VM. Every application runs in its own Linux process. A process can have multiple threads.

In Android, all components of an application run in the same process. But if we want to change which process a certain component belongs to, we can change this in the manifest file.

The manifest entry for component elements - Activity, Service, Receiver, and Provider supports an android:process attribute that can specify a process in which that component should run. We can also set android:process such that components of different applications run in the same process, provided that the applications should have the same Linux user ID and must be signed with the same certificates.

In Android, application process's lifetime is not controlled by the application itself. It is determined by the memory available in the system, parts of the application and the importance of the things to the user.

Android determines which process to be killed, when the memory is low, based on importance hierarchy. This hierarchy is formed based on the components running in them and the state of the components. The classification of process types, based on importance hierarchy are listed in the order of importance.

- Foreground Process - A process that is required for what the user is doing, currently.

- Visible Process - This is holding an *Activity* that is visible to user, but it is not in the foreground.

- Service Process - A process that is holding a *Service* that started with startService() method. Background MP3 plaayback is an example of Service Process.

- Background Process - This holds an *Activity* component that is not currently visible to the user.

- Empty Process - This process does not hold an active application component.

It is clear that application components play a major role in determining the lifetime of an application process.

2.2.9 Applications and Task

Android applications are run by processes and their included threads. The two terms task and application are linked together tightly, given that a task can be seen as an application by the user. In fact tasks are a series of activities of possibly multiple applications. Tasks basically are a logical history of user actions, e.g. the user opens a mail application in which he opens a specific mail with a link included which is opened in a browser. In this scenario the task would include two applications (mail and browser) whereat there are also two Activity components of the mail application and one from the browser included in the task. An advantage of the task concept is the opportunity to allow the user to go back step by step like a pop operation on a stack.

2.2.10 Native Process Vs Android Application

- Application - An application is a software that is designed to help the user to do a specific task. It interacts with the user.
- Process - A program contains code for performing tasks, what an application requires. A process is a program in execution. An application accomplishes its task by using processes.

2.2.11 Components in Android Lifecycle

The most important components of the Android application are

- Activity -

  represents a user interface screen, window or form; an Android application can have one or more activities; an agenda application can have an activity to manage contacts, an activity to manage meetings and one to edit an agenda entry; each Activity has its own lifecycle independent from the application process lifecycle; each Activity has its own state and it is possible to save it and restore it; activities can be started by different applications (if it is allowed); has a complex lifecycle because applications can have multiple activities and only one is in the foreground; using the Activity Manager, the Android System manages a stack of activities which are in different states (starting, running, paused, stopped, destroyed); in the Android SDK is implemented using a subclass of Activity class which extends the Context class;

- Intent - represents an entity used to describe an operation to be executed; is a message transmitted to another component in order to announce an operation; somehow similar to the event-handler concept from .NET or Java; an asynchronous message used to activate activities, services and broadcast receivers; implemented by the Intent class;

- Service - a task that runs in the background without the user direct interaction; implemented by a subclass of Service;

- Content provider - a custom API used to manage application private data; a data management alternative to the file system, the SQLite database or any other persistent storage location; implemented as a subclass of ContentProvider; a solution to share and control (with permissions) data transfer between applications (i.e. Android system provides a content provider for the users contact information);

- Broadcast receiver - a component that responds to system-wide broadcast announcements; somehow similar to the global (or system event) handler concept; implemented as a subclass of BroadcastReceiver.

2.2.12 Current Process and Past Process

This thesis works on the details of the present and past process. We discussed the different states of a process in the previous section. The process in running state is termed as present or current process. The process which was running and got terminated is termed as the past process.

2.3 /procfs file system

/proc file system is a virtual file system in Unix which acts as a window to the Linux OS. This file system does not exist on disk, but the kernel creates it in memory. So this is also called a pseudo filesystem. ls -l shows size as zero and also file command gives the output /textitempty as the files are generated instantly when these files are accessed.

It contains all the information regarding the processes and other system information. There are process specific and Kernel specific entries within this filesystem. The mount point of this file system is called /proc. This is mounted at boot time. Many of the system utilities are read calls to the files in /proc. For example, lsmod is equivalent to cat /proc/modules. Another example is the free command, which reads the memory information from /proc/meminfo file, formats it, and displays it. This file system is also used to change certain kernel parameters at run time. sysctl is a utility used to change kernel parameter. It is equivalent to altering the files in /proc/sys, which has all the configurable kernel parameters as files that are writable. By altering the files located under /proc directory we can alter the kernel parameters in runtime.

There are two types of folders within this directory - numbered and unnumbered. The numbered folders correspond to the pids of all the running processes. Each running process has a folder inside this directory namely /proc/PID, where PID is the process id. Many details regarding the process can be obtained from files within /proc/PID

2.3.1 Numbered Files within /proc directory

Some of the important files within a numbered directory are

- cmdline - command line of the command.
- environ - environment variables.
- fd - Contains the file descriptors which are linked to the appropriate files.
- limits - Contain the information about the specific limits to the process.
- mounts - mount related information
- stat - Process status
- status - Process status in human readable form
- statm - Process memory status information

Following are the important links under each numbered directory (for each process).

- cwd - Link to the current working directory of the process.
- exe - Link to the executable of the process.
- root - Link to the root directory of the process.

2.3.2 Files in /proc directory

- /proc/cpuinfo - information about CPU.
- /proc/meminfo - information about memory.

- /proc/loadvg - load average.
- /proc/partitions - partition related information.
- /proc/version - Linux version.

2.3.3 Sysfs File System

Sysfs is a virtual filesystem similar to /proc. It is exported by the Kernel. It was introduced in Linux 2.6 Kernel. The files within Sysfs contain details about devices and drivers. This file system enables to view the device topology of a system as a simple file system. Some files in this file volume are writable, for configuration and control of devices attached to the system. The mount point for Sysfs is on /sys. It is the union of proc, devfs and devpty file system. This file system lists the devices and buses attached to the system into a file system hierarchy that can be accessed from user space. It is designed in a structured way to handle the device and driver specific options that have previously resided in /proc/, and contain the dynamic device addition previously offered by devfs.

There are seven sub-directories within sysfs are block, bus, class, devices, firmware, module, and power.

- The /block/ directory - It contains one directory for each of the registered block devices on the system.
- The /bus/ directory -It provides a view of the system buses.

- The /class/ directory - It contains directories that group together similar devices such as network devices, SCSI tape drives, ttys and other miscellaneous devices.

- The /devices/ directory - It provides a view of the device topology of the system. It has the hierarchy of device structures inside the kernel.

- The /firmware/ directory - It contains a system-specific tree of low-level subsystems such as ACPI, EDD, EFI, and so on.

- The /power/ directory - It contains system-wide power management data.

2.4 System Calls in Linux

System calls are methods called by an application to request a service from the Operating System. The primary aim of an Operating System is to provide Process Management, File Management, Memory Management and other things like timing, scheduling and network management. An application uses the required System Call to perform the service it requires. Hence, they are said to be an interface between the application and Operating System Kernel. There are over 300 system calls in Linux.

2.4.1 System Calls Related with Process Management

- fork() - Creates a child process identical to parent

- waitpid() - Wait for a child to terminate

- s=execve() - Executes the program pointed to by filename.

- exit(status) - Terminate process execution and return status

2.4.2 File Management Related System Calls

- fd=open(file, how,.) Open a file for reading, writing or both.

- s=close(fd) Close an open file

- n = read(fd, buffer, nbytes) Read data from a file to a buffer

- n = write(fd, buffer, nbytes) Write data from buffer to a file

- position = lseek(fd, offset, whence) Move the file pointer

- s = stat(name,&buff) Get a file's status information

2.4.3 Network Connection Related System Calls

- socket - Creates a socket.

- connect - This system call is used to connect referred to by the file descriptor to the addr specified

- bind - Assigns a local protocol address to a socket

- listen - Listens for connections on the socket

- accept - This is called by the TCP server to return the next completed connection from the front of the completed connection queue.

- recvfrom - Receives a message from a socket.

- close - Marks the socket as closed and returns to the process immediately.

2.5 Strace

Strace is a tool used to trace the system calls and signals used by a process. It is a useful diagnostic and debugging tool. It intercepts and records the system calls called by a process and the signals received by a process. The system call's name, its arguments, and its return value are printed to stdout or to the file specified with the -o option.

2.5.1 Options

Here are some of the important options to be used with Strace

Table 2.1: Options to be used with Strace

Options	Description
-h	Trace child processes as they are created by currently traced processes as a result of the fork(2) system call.
-o filename	Write the trace output to the file filename rather than to stderr.
-p pid	Attach to the process with the process ID pid and begin tracing.
-e expr	A qualifying expression which modifies which events to trace or how to trace them.
-e trace=set	Trace only the specified set of system calls.
-e trace=file	Trace all system calls which take a file name as an argument.
-e trace=process	Trace all system calls which involve process management. This is useful for watching the fork, wait, and exec steps of a process.
-e trace=network	Trace all the network related system calls.
-tt	Prefix each line of the trace with time in microseconds
-s strsize	Specify the maximum string size to print (the default is 32).
-v	Print unabbreviated versions of environment, stat, termios, etc. calls.
-T	Show the time spent in system calls. This records the time difference between the beginning and the end of each system call.

2.5.2 Some Problems in Strace

- Strace interferes with process flow

  The use of strace with a process causes additional computational overhead and many contexts switch between the program and strace. When system performance is a priority, we should be careful in using Strace.

- Strace Output Readable to All Users

  The output of strace is in a readable format to all users. It may disclose some sensitive information. For example, it may display the private SSH key in read() while tracing the execution of SSH.

2.6 ltrace

- ltrace can be used to trace shared library calls. This can be very useful to get insight into the program

- It is an useful debugging and reverse engineering tool.

- Programs use system calls to access system resources such as opening a file or connecting to the network. Library calls are calls made to the GNU C library which provides a more programmer-friendly interface to the system.

- This tool is very useful for debugging user-space applications to determine which library call is failing. It is also capable of receiving signals for segmentation faults, etc.

2.6.1 Debugging a program with ltrace

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.
4.  int main()
5.  {
6.    FILE *fp = fopen("rfile.txt", "w+");
7.     fprintf(fp+1, "Invalid Write\n");
8.     fclose(fp);
9.     return 0;
10. }
```

When we compile and run this program

```
anusuya@ubuntu:~/source$ gcc file.c -Wall -o file
anusuya@ubuntu:~/source$./file
Segmentation fault (core dumped)
```

We can ltrace to debug this code and find the details of error

```
anusuya@ubuntu:~/source$ltrace ./file
__libc_start_main(0x8048454, 1, 0xbfc19db4, 0x80484c0, 0x8048530 <unfinished
     ...>
fopen("rfile.txt", "w+")                        = 0x9160008
fwrite("Invalid Write\n", 1, 14, 0x916009c <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++content...
```

The first line states the libc function call main(), with its parameters and it terminated unfinished. Since we saw the segmentation fault, so unfinished main() aligns with our understanding.

Next line has listed the fopen() call to open rfile.txt in write plus mode and returning a valid file pointer address. Moving on to the next library call, we have fwrite which is unfinished. Here is the bug because of which program crashed. So focusing on this call and analyzing, notice the file pointer containing the address in fwrite() call doesnt match the one in fopen() call. Hence, the crash. The program is trying to write onto a file using file pointer which is invalid following a SIGSEGV signal. Getting back to our program, the bug is in line 7 of our program where the file pointer being passed to the fprintf() call is tampered. So, we just found the bug.

```
        fprintf(fp, "Invalid Write\n");
```

2.7 Strace and ltrace comparision

The advantage of strace over ltrace is that the information is often simpler. Not all library calls map to system calls. The disadvantage is that not all library calls have the same name as their system call equivalent. For example, the fork() library call uses the clone() system call which can also be used for threading as well as making a new process. ltrace only traces calls from the executable to the libraries the executable is linked against. It does not trace calls between libraries. Hence, calls such as to the printf() function (which itself resides in the libc shared library) is not shown. Also, there is no option to include the library name in the output for each of the called functions. But we can still trace scripts by tracing the script interpreter.

```
 ltrace python myscript.py
```

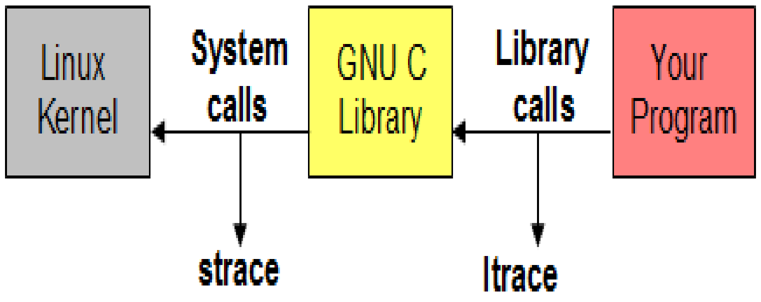


Figure 2.3: ltrace and strace

2.8 Cloud Upload

This thesis is going to generate a lot of log files as a part of using tools like strace and ltrace. It is difficult to store and manage a large number of files on a local disk. So we prefer to use cloud storage, a method of storing our digital data in logical pools where the physical storage spans across multiple servers and locations and the physical environment are owned and managed by a hosting company, usually called Cloud Storage Providers. The cloud storage providers are responsible for keeping the data available and accessible, and the physical environment protected and running. People and organizations buy or lease storage capacity from the providers to store user, organization, or application data.

Cloud storage can provide the benefits of greater accessibility and reliability, rapid deployment, strong protection for data backup, archival and disaster recovery purposes, and lower overall storage costs as a result of not having to purchase, manage and maintain expensive hardware. There are many benefits of using cloud storage, however, cloud storage does have the potential for security and compliance concerns that are not associated with traditional storage systems.

2.8.1 Types Of Cloud Storage

- Personal/Mobile Cloud Storage It is used for storing individual's data in the cloud. The person will be given access to the data from anywhere. It can be used to backup data in the mobile device. It is subset of Public Cloud Storage.

- Public Cloud Storage This is used by enterprises where they have rented or subscribed for the cloud storage for a certain period of time. The cloud server is fully managed by the cloud storage provider. Anyone with valid access and user credentials can access the data from anywhere in the world.

- Private Cloud Storage This type is used by an individual or a company. The cloud storage provider sets up the data center in the user's space or premise. This provides more security and restricted access to data.

- Hybrid Cloud Storage This is a combination of public and private cloud storage. Here some data is stored in public cloud storage. Sensitive data is stored using private cloud storage, in the user's place and has restricted access.

  We will be using the personal cloud storage for our thesis.

Chapter 3

Provenance of a Process

3.1 Provenance of a Process

The word *provenance* means origin, source or history of ownership of an object. Provenance of a process means getting the details of its origin, source and also what all it does during its lifetime. By knowing the provenance of a process, we will be able to tell the time of creation of the process, by which process it was triggered, what are all the child processes it invoked, it activities during its life period and also the time it ended It should also provide all the details of files it opened, created, written and also the network connections it spawned.

3.2 Method

The provenance of a process is obtained by analysing the system calls and library calls a process used while execution.

Monitoring Linux utility tools like strace and ltrace are used to monitor and record the system calls and library calls a process used.

These tools are attached to all the running process. The output obtained using these tools are logged. The logged details are then analysed.

Program for running the strace and ptrace tools is written in a script.

This script should be always up and running, whenever the device is booted.

3.2.1 Past Process

Whenever a process gets terminated, the output of the strace and ltrace are sent to a log file. The log files are interpreted to get the provenance of a process.

3.2.2 Current Process

The details of current processes can be gathered from the /proc directory. So we have to define an API such that it reads the files within /proc directory and displays the details.

We can use JNI(Java Native Interface) or Android API's like ApplicationInfo, ActivityManager.

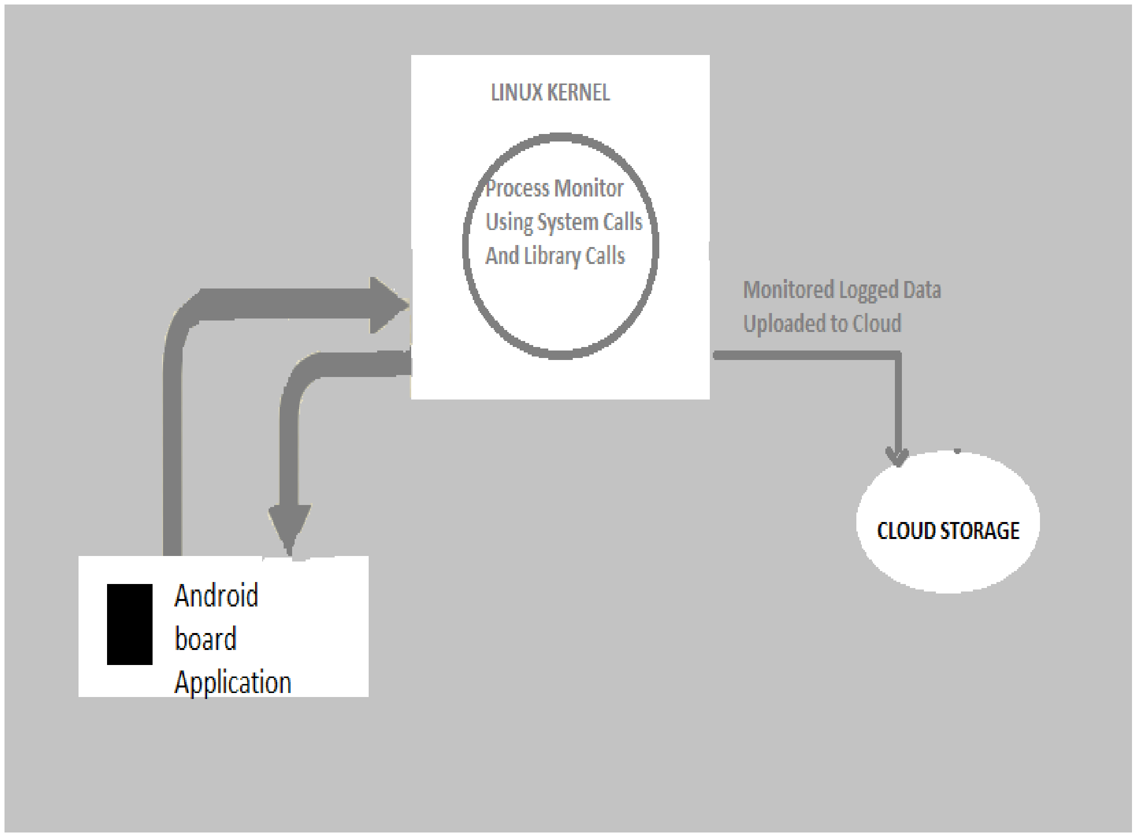

LINUX KERNEL
Process Monitor Using System Calls And Library Calls
Monitored Logged Data Uploaded to Cloud
CLOUD STORAGE
Android board Application

Figure 3.1: Architecture Of the System

Chapter 4

Problem Solution Architecture

This section describes solutions to various problems that evolved while developing and implementing the thesis.

4.1 Strace in Android

Strace is not a built in tool in Android. So we have to install it in Android. The Android device must be rooted for installing strace. The following steps explains how to install strace in Android.

- Download the strace source distribution from the SourceForge strace project
- Extract the strace package on your linux machine.
- Set your compiler information:

  ```
  export CC=your_cross_compiler_folder/bin/arm-none-linux-gnueabi-gcc
  export STRIP=your_cross_compiler_folder/bin/arm-none-linux-gnueabi-strip
  export CFLAGS="-O2 -static"
  ```

- Add cross_compiler folder /bin into your $PATH.

- Change your current folder to the strace folder (where you extracted the source), use command.

```
./configure --host=arm-linux
```

- Compile

```
make
```

- Verify result

```
  file strace
```

The output would be like

```
  strace: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for GNU/Linux
      2.6.14, statically linked, not stripped
```

4.2 Filtering Strace Output

The output of strace can quickly become very large and synchronous output to strerr or a log file may greatly impair program performance. If we know up front which system calls are of importance, we may limit the output to one or a few kernel calls (or e.g. with -e file system calls with file names). This slows down program execution considerably less and makes subsequent evaluation of the strace output much easier. For our thesis, we concentrate mostly on file and network related system calls.

4.3 Disadvantages Of Using JNI

- Bad C/C++ code in native library will cause core dumps / segmentation faults that the JVM cannot recover from. This may result in crashing of the whole app.

- Difficult to debug runtime error in native code.
- Potential security risk.

4.4 Storage

A cloud storage for the collected log files is preferred, as the log files keeps on getting accumulated while the processes are running.

All the logged details will be uploaded in cloud server. A log uploader will be uploading all the logs to clouds in a configurable way like configurable time interval for log upload. All the uploaded logs will be deleted from the device.

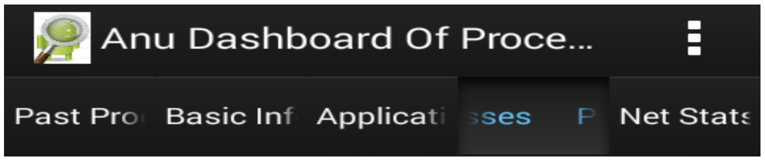
JSON and OAuth API is used for cloud integration. There are many free cloud storage providers available. We can choose any one of them, for our cloud upload.

Chapter 5

Implementation and Results

5.1 APK Created

We created an application named *ANU DASHBOARD OF PAST AND PRESENT PROCESS*. This has various 5 tabs - Past Process, Basic Info, Application, Process and Net Stats.



The tabs

- Past process has process name, process id, time started, time ended, files opened, files written, files read, network connection it spawned.
- Basic Info has details of internal storage, sd card storage, battery, memory, processor.
- Application Info has details of all the running application, its package name, user id, install date, file size.

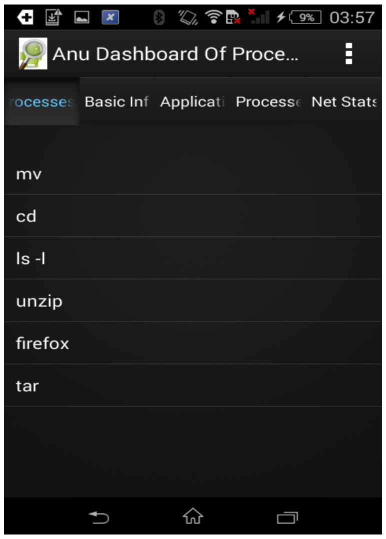
- Process has details of running processes, process id, user id, state, importance, lru

- Net stats - It displays the network details, local/remote address, state.

A shell script is created to get the strace of all the running process and once it terminated, it will be stored in a text file, with the timestamp appended to the name.

5.1.1 Past Process

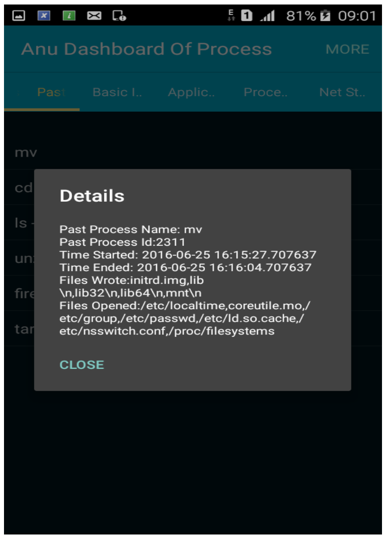
In the past process tab, the list of all the past processes name are listed based in a chronoligaical order. We can select the time from when the past processes should be listed - last one hour, last one day and last one week.



mv
cd
ls -l
unzip
firefox
tar

Figure 5.1: List of Past Processes

When we click on each past process, we get the details like pid, time started, time ended, files opened, files written, network connection it used.



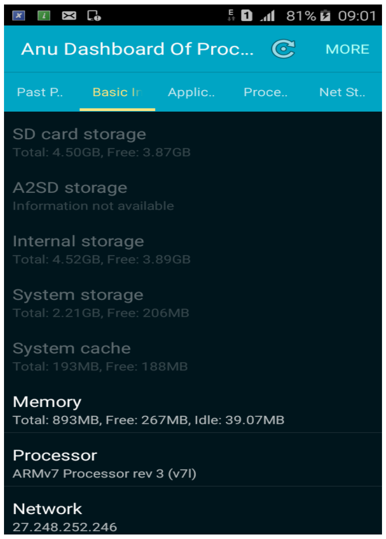
Details

Past Process Name: mv
Past Process Id:2311
Time Started: 2016-06-25 16:15:27.707637
Time Ended: 2016-06-25 16:16:04.707637
Files Wrote:initrd.img,lib
\n,lib32\n,lib64\n,mnt\n
Files Opened:/etc/localtime,coreutile.mo,/
etc/group,/etc/passwd,/etc/ld.so.cache,/
etc/nsswitch.conf,/proc/filesystems

Figure 5.2: Details of Past Processes

5.1.2 Basic Info

In the basic info tab details like battery, SD card storage, A2SD storage, internal storage, system storage, system cache, processor and memory are listed.



SD card storage
Total: 4.50GB, Free: 3.87GB

A2SD storage
Information not available

Internal storage
Total: 4.52GB, Free: 3.89GB

System storage
Total: 2.21GB, Free: 206MB

System cache
Total: 193MB, Free: 188MB

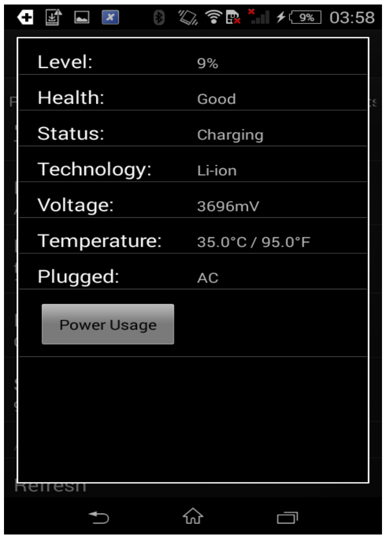
Memory
Total: 893MB, Free: 267MB, Idle: 39.07MB

Processor
ARMv7 Processor rev 3 (v7l)

Network
27.248.252.246

Figure 5.3: Storage

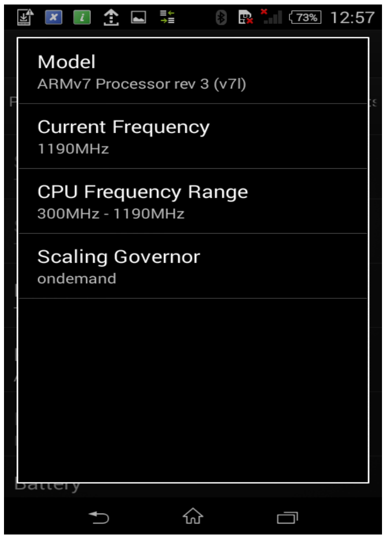
When we click on the battery item, we get details like level, health, status, technology, voltage, temperature, plugged.


03:58

Level:	9%
Health:	Good
Status:	Charging
Technology:	Li-ion
Voltage:	3696mV
Temperature:	35.0°C / 95.0°F
Plugged:	AC

Figure 5.4: Battery Information

When we click on the processor, we get details like model, current frequency, cpu frequency range and scaling governor.


12:57

Model
ARMv7 Processor rev 3 (v7l)

Current Frequency
1190MHz

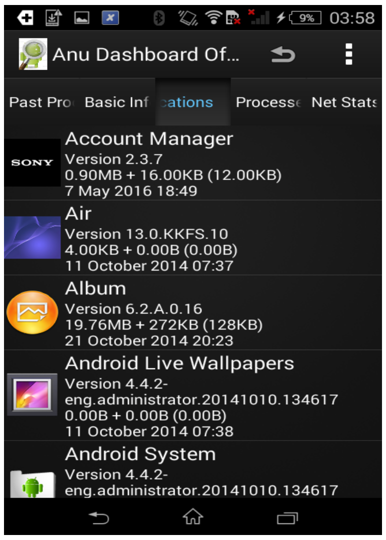
CPU Frequency Range
300MHz - 1190MHz

Scaling Governor
ondemand

Figure 5.5: Processor Details

5.1.3 Application

Under the *Application* tab, all the current running applications are listed.



Account Manager
Version 2.3.7
0.90MB + 16.00KB (12.00KB)
7 May 2016 18:49

Air
Version 13.0.KKFS.10
4.00KB + 0.00B (0.00B)
11 October 2014 07:37

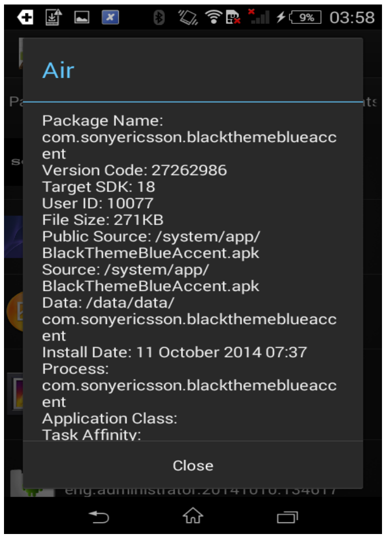
Album
Version 6.2.A.0.16
19.76MB + 272KB (128KB)
21 October 2014 20:23

Android Live Wallpapers
Version 4.4.2-eng.administrator.20141010.134617
0.00B + 0.00B (0.00B)
11 October 2014 07:38

Android System
Version 4.4.2-eng.administrator.20141010.134617

Figure 5.6: All The Running Application

When we click on each application, we get details like package name, user id, file size, public source, install date, permission, flags and enabled.

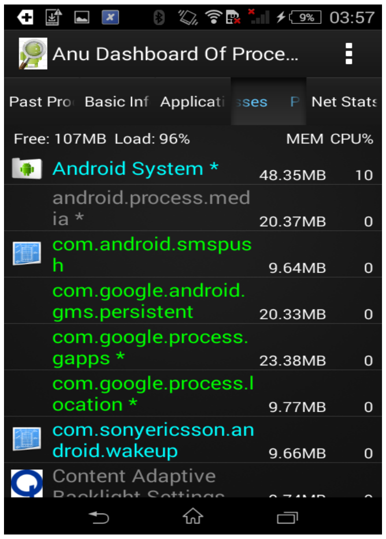


Package Name: com.sonyericsson.blackthemeblueaccent
Version Code: 27262986
Target SDK: 18
User ID: 10077
File Size: 271KB
Public Source: /system/app/BlackThemeBlueAccent.apk
Source: /system/app/BlackThemeBlueAccent.apk
Data: /data/data/com.sonyericsson.blackthemeblueaccent
Install Date: 11 October 2014 07:37
Process: com.sonyericsson.blackthemeblueaccent
Application Class:
Task Affinity:

Figure 5.7: Details Retrieved Of An Application

5.1.4 Processes

Under the *Processes* tab, all the current processes are displayed. Color differentiation is given for different type of processes like - foreground process(blue), background process(yellow), system services(grey).

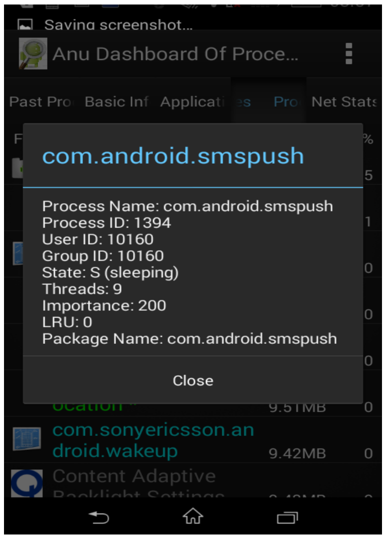


Anu Dashboard Of Proce...

Past Pro Basic Inf Applicati ses P Net Stats

```
Free: 107MB  Load: 96%                 MEM   CPU%
Android System *                  48.35MB     10
android.process.media *           20.37MB      0
com.android.smspush                9.64MB      0
com.google.android.gms.persistent 20.33MB      0
com.google.process.gapps *        23.38MB      0
com.google.process.location *      9.77MB      0
com.sonyericsson.android.wakeup    9.66MB      0
Content Adaptive Backlight Settings
```

Figure 5.8: Process List



Saving screenshot...

Anu Dashboard Of Proce...

com.android.smspush

```
Process Name: com.android.smspush
Process ID: 1394
User ID: 10160
Group ID: 10160
State: S (sleeping)
Threads: 9
Importance: 200
LRU: 0
Package Name: com.android.smspush
```

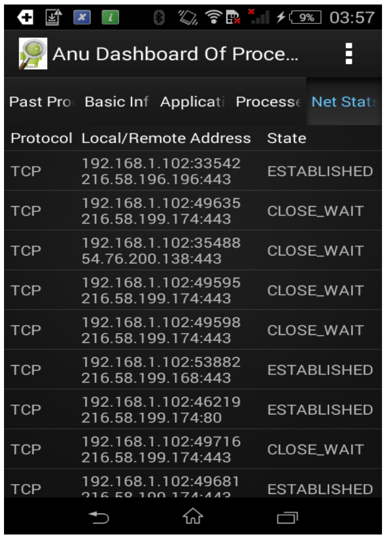
Close

Figure 5.9: Process Details

When we click on each process, we get details like process name, process id, user id, group id, state(sleeping, runnin,.), threads, importance, lru, package name.

5.1.5 Net Stats

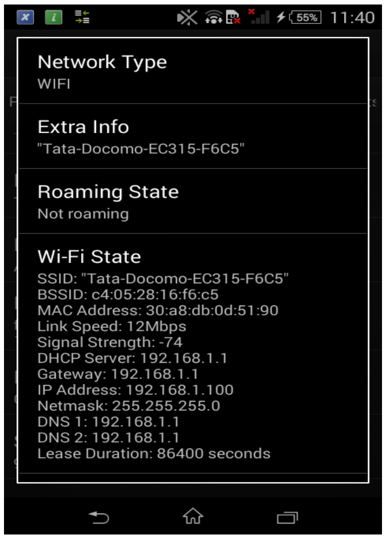
Under *Net Stats*, we get the details of the protocol used, local/remote address, state of the protocol.



Protocol	Local/Remote Address	State
TCP	192.168.1.102:33542 216.58.196.196:443	ESTABLISHED
TCP	192.168.1.102:49635 216.58.199.174:443	CLOSE_WAIT
TCP	192.168.1.102:35488 54.76.200.138:443	CLOSE_WAIT
TCP	192.168.1.102:49595 216.58.199.174:443	CLOSE_WAIT
TCP	192.168.1.102:49598 216.58.199.174:443	CLOSE_WAIT
TCP	192.168.1.102:53882 216.58.199.168:443	ESTABLISHED
TCP	192.168.1.102:46219 216.58.199.174:80	ESTABLISHED
TCP	192.168.1.102:49716 216.58.199.174:443	CLOSE_WAIT
TCP	192.168.1.102:49681	ESTABLISHED

Figure 5.10: Network State

The current network type, local address, roaming state, ip address are displayed.


11:40

Network Type
WIFI

Extra Info
"Tata-Docomo-EC315-F6C5"

Roaming State
Not roaming

Wi-Fi State
SSID: "Tata-Docomo-EC315-F6C5"
BSSID: c4:05:28:16:f6:c5
MAC Address: 30:a8:db:0d:51:90
Link Speed: 12Mbps
Signal Strength: -74
DHCP Server: 192.168.1.1
Gateway: 192.168.1.1
IP Address: 192.168.1.100
Netmask: 255.255.255.0
DNS 1: 192.168.1.1
DNS 2: 192.168.1.1
Lease Duration: 86400 seconds

Figure 5.11: Network Connection

5.2 Details of the APK

The stastistics of the sourcecode obtained using cloc is given in the Table 5.1

Table 5.1: Source Code Statistics

http://cloc.sourceforge.net

Language	Files	Blank	Comment	Code
JAVA	16	3118	699	14391
XML	25	68	1	813

5.3 Link to source code

The source code of the project is available in the following link. http://www.github.com/anurp/AnuDash

Related app to this thesis, OSMonitor source code are available in the link

http://www.github.com/anurp/OSMonitor

Chapter 6

Methods Used

6.1 Running Process Details

As we discussed the disadvantages of JNI, in the previous chapter, we are using Android classes and methods to get the information of processes and all the device and OS related information.

We want to list all the currently running processes with all the details. This can be achieved with the class ActivityManager.RunningAppProcessInfo.

To get the process name.

```
String procName = rap.procInfo.processName;
```

Process ID:

```
int procId = procInfo.pid;
```

To List The Processes Based On IMPORTANCE:

We have five fields to determine the importance of a process

```

RunningAppProcessInfo.IMPORTANCE_FOREGROUND;

RunningAppProcessInfo.IMPORTANCE_PERCEPTIBLE;
RunningAppProcessInfo.IMPORTANCE_VISIBLE;

RunningAppProcessInfo.IMPORTANCE_SERVICE;

RunningAppProcessInfo.IMPORTANCE_BACKGROUND;

RunningAppProcessInfo.IMPORTANCE_EMPTY;
```

lru:

```
procInfo.lru
```

6.2 Retrieving Application Details

Android Application is the software running in the device. To get the all the details related to Android running Applications, we can use the class ApplicationInfo.

To list all the Applications:

```
final List<ApplicationInfo> filteredApps = filterApps( allApps );
```

To get the Package name of the Application:

```
String pkgName = ai.appInfo.packageName;
```

6.3 Retrieving Network Connection Related Information

Network Connection Related Information can be obtained from class ConnectivityManager;

This class answers queries about the state of network connectivity. It also notifies applications when network connectivity changes. Get an instance of this class by calling

```
 Context.getSystemService(Context.CONNECTIVITY_SERVICE).
```

The primary responsibilities of this class are to

- Monitor network connections (Wi-Fi, GPRS, UMTS, etc.).
- Send broadcast intents when network connectivity changes.
- Attempt to fail over to another network when connectivity to a network is lost.
- Provide an API that allows applications to query the coarse-grained or fine-grained state of the available networks.
- Provide an API that allows applications to request and select networks for their data traffic.

We use the method getActiveNetwork() of this class to the network connection details like Network Type, Roaming State and Extra Info.

Returns a Network object corresponding to the currently active default data network.

The class WifiManager provides the primary API for managing all aspects of Wi-Fi connectivity. Get an instance of this class by calling

```
Context.getSystemService(Context.WIFI_SERVICE).
```

It deals with several categories of items.

- The list of configured networks.
- The list can be viewed and updated, and attributes of individual entries can be modified.

- The currently active Wi-Fi network, if any.
- Connectivity can be established or torn down, and dynamic information about the state of the network can be queried.
- Results of access point scans, containing enough information to make decisions about what access point to connect to.
- It defines the names of various Intent actions that are broadcast upon any sort of change in Wi-Fi state.

We use the method getConnectionInfo () to get details such as SSID, BSSID, Mac Address, Link Speed, Signal Strength of wifi connection.

We use the method getDhcpInfo() to get the details such as DHCP server address, gateway address and DNS address.

6.4 Battery Information Of The Device

Battery Manager helps us to retrieve the battery details of the device. The BatteryManager class contains strings and constants used for values in the ACTION BATTERY CHANGED Intent, and provides a method for querying battery and charging properties.

To know the level of battery.

```
int level = intent.getIntExtra( "level", 0 );
int scale = intent.getIntExtra( "scale", 100 );
String lStr = String.valueOf( level * 100 / scale ) + '%';
data.add( new String[]{
getString( R.string.batt_level ), lStr} );
```

To know the health of the battery, whether it is - Good, Over heat, Dead, Over Voltage, Failure.

```
int health = intent.getIntExtra( "health", BatteryManager.BATTERY_HEALTH_UNKNOWN );
```

To know the status of the battery - Charging, Discharging, Full , Not Charging.

```
int status = intent.getIntExtra( "status", BatteryManager.BATTERY_STATUS_UNKNOWN );
```

To know the temperature of the device.

```
int temperature = intent.getIntExtra( "temperature", 0 );
```

To know the voltage of the device

```
int voltage = intent.getIntExtra( "voltage", 0 ); //$NON-NLS-1$

                        String vStr = String.valueOf( voltage ) + "mV"; //$NON-NLS-1$

                        data.add( new String[]{
                                        getString( R.string.batt_voltage ), vStr
                        } );
```

6.5 Cloud Upload Of Log Files

We use Dropbox to upload our collected log files. We have to use Core API to achieve this. The Core API is based on HTTP and OAuth and provides low-level calls to access and manipulate a user's Dropbox account. The Core API uses OAuth v2. A Dropbox account is required for transferring to and from Dropbox Cloud. We have to register our app in the App console within our Dropbox account. We have to define the app secret and app key while registration. This will be used in our Android app for upload and download of files.

```
final static private String APP_KEY = "INSERT_APP_KEY";
final static private String APP_SECRET = "INSERT_APP_SECRET";
```

We have to authenticate our app to the Dropbox. startOAuth2Authentication() method is called to authorize the app.

```
mDBApi.getSession().startOAuth2Authentication(MyActivity.this);
```

More about registering our app in Dropbox and authenticating is explained in details in appendix.

6.5.1 Upload Module

```
File file = new File("working-draft.txt");
FileInputStream inputStream = new FileInputStream(file);
Entry response = mDBApi.putFile("/magnum-opus.txt", inputStream,
                        file.length(), null, null);
Log.i("DbExampleLog", "The uploaded file's rev is: " + response.rev);
```

6.5.2 Download Module

```
File file = new File("/magnum-opus.txt");
FileOutputStream outputStream = new FileOutputStream(file);
DropboxFileInfo info = mDBApi.getFile("/magnum-opus.txt", null, outputStream, null);
Log.i("DbExampleLog", "The file's rev is: " + info.getMetadata().rev);
```

6.5.3 Deleting Files After Cloud Upload

Once the file is uploaded to the cloud, we have to delete the file immediately, to free the space in the device. There is a java method called mFile.delete(). Once we get the result that theace outp file has been uploaded successfully to the cloud, this method is called and the file gets automatically deleted in the disk.

6.6 Strace for all running processes

A shell script is used for getting the strace details of all the processes.

```
#!/usr/bin/env bash
pid=$(ps ax | perl -nle 'print $1 if /^ *([0-9]+)/')
for i in "$pid"
do
strace -p $i -o /tmp/strace_$(date).txt
done
```

A part of the strace output file obtained is shown below. .

```
2550  16:15:27.707637 execve("/bin/ls", ["ls", "-l"], ["TERM=xterm",        "
    LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd
    =40;33;01:or=40;31;01:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex
    =01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31"...,  "PATH=/usr/local/sbin:/usr/local/
    bin:/usr/sbin:/usr/bin:/sbin:/bin", "LANG=en_IN", "HOME=/home/anu", "LANGUAGE=
    en_IN:en", "DISPLAY=:0", "COLORTERM=gnome-terminal", "XAUTHORITY=/home/anu/.
    Xauthority", "SHELL=/bin/bash", "MAIL=/var/mail/root", "LOGNAME=root", "USER=root
    ", "USERNAME=root", "SUDO_COMMAND=/usr/bin/strace -e fork,read,write,open,close,
    link,execve,rename,dup,dup2,symlink,clone,vfork,setuid32,setgid32,chmod,fchmod,
    pipe,truncate,ftruncate,readv,recv,recvfrom,recvmsg,send,sendt"..., "SUDO_USER=anu
    ", "SUDO_UID=1000", "SUDO_GID=1000"]) = 0 <0.000366>
2550  16:15:27.708604 open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3 <0.000031>
2550  16:15:27.708785 close(3)          = 0 <0.000050>
2550  16:15:27.708999 open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|
    O_CLOEXEC) = 3 <0.000022>
2550  16:15:27.709071 read(3, "\177ELF
    \2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0[\0\0\0\0\0\0@\0\0\0\0\0\0X
    \5\2\0\0\0\0\0\0\0\0\0@\0008\0\10\0@
    \0\35\0\34\0\1\0\0\0\5\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0,
\366\1\0\0\0\0\0,\366\1\0\0\0\0\0\0\0
```

6.7 Getting The File Related Details From Strace Output

The system calls related to file management are open, read, write, close. We have to get the file descriptors to retrieve the files that are used while a process is executing. In Open system call, the first argument is the file opened. In Read syscall the first argument is the file descriptor. The file descriptor is the return argument of the Open syscall.

```


if (syscall.equals("open")) {

            String path = args.substring(1, args.lastIndexOf('\"'));
            String fd = retVal;
}
if (syscall.equals("read") || syscall.equals("pread") || syscall.equals("readv") ||
    syscall.equals("recv") || syscall.equals("recvfrom") || syscall.equals("recvmsg"))
     {
            String fd = args.substring(0, args.indexOf(','));
}
```

6.8 Getting the Network Connection Related Details From Strace Output

The system calls related to network connection are connect, recv. We get the socket from connect system call.

```
static final Pattern networkPattern = Pattern.compile("sin_port=htons\\(([0-9]+)\\),
    sin_addr=inet_addr\\(\"(.*)\"\\)");

if (syscall.equals("connect")) {
            String fd = args.substring(0, args.indexOf(','));
            String socket = args.substring(args.indexOf('{'), args.indexOf('}') + 1);
}
Matcher networkMatcher = networkPattern.matcher(path);
                if (networkMatcher.find()) {
                    vertex.addAnnotation("subtype", "network");
                    vertex.addAnnotation("address", networkMatcher.group(2));
                    vertex.addAnnotation("port", networkMatcher.group(1));
```

Chapter 7

Related Work

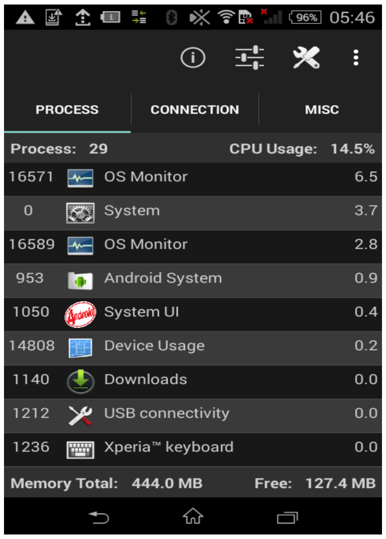
7.1 OS Monitor - A Case Study

OS Monitor[eolwral, 2015] is an open source Android app. It is a tool for monitoring our Android system. It displays information such as

- Processes - monitor all processes.
- Connections - display every network connection.
- Misc - monitor battery, processors, network interfaces and file system.
- Messages - search dmesg or logcat in real-time.

OS Monitor contains two major components, a native executable binary that runs on the background to collect system information, and a user interface.

When UI component sends a request, the background binary will collect and serialize data. The data will be transferred via IPC and renderred on screen. It uses 'C 'code to read the /proc file directory and Java Native Interface(JNI)



Process: 29 CPU Usage: 14.5%

16571	OS Monitor	6.5
0	System	3.7
16589	OS Monitor	2.8
953	Android System	0.9
1050	System UI	0.4
14808	Device Usage	0.2
1140	Downloads	0.0
1212	USB connectivity	0.0
1236	Xperia™ keyboard	0.0

Memory Total: 444.0 MB Free: 127.4 MB

Figure 7.1: Snapshot from OS Monitor app

The background binary works in single thread mode, all the requests are queued and processed.

Communication between Android App and native process is through Unix socket. All requests or responses are serialized by Google flatBuffer. It also has security features. OS Monitor is based on Unix socket with a randomized name. Knowing the randomized name, anyone can easily hijack it with his own app. A security token is used to prevent such hijack. Any connection needs to offer a token for validating, otherwise, the connection will be disconnected.

7.2 Network Monitoring and Filtering

Network Ombudsman [George, 2015] is a network monitoring and filtering tool. It is built in Android framework. It categorises and ranks applications as malicious and non-malicious The Network ombudsman system comprises of a monitoring service, filtering/firewall service, a log uploader, an android application and a cloud server. The monitoring service will keep tab of all the network connection both internal and external to the device. The filtering/firewall service will update the rules of the netfilter system firewall and enables the blocking/unblocking of the devices. This service will also keep tab of the context changes and reputation changes and changes the firewall policies accordingly. All the monitored data that will be logged by the monitoring service will be uploaded into the cloud server by the log uploader.

7.3 Process Authentication

A new process is created when an application or service is launched and the runtime systen becomes the parent process of all user application processes in Android. Android has numerous vulnerabilities that leads to failure of application sandboxes.

This Paper *DroidBarrier: Know what is executing on your Android*[Almohri et al., 2014] uses a technique called *Process Authentication* for Android applications to overcome the shortcomings of current Android security practices. A new process is created when an application or service is launched and the runtime systen becomes the parent process of all user application processes in Android. Android has numerous vulnerabilities that leads to failure of application sandboxes. For example, the Gingerbreak exploit affected the popular Android 2.3 enabled a malicious application to gain root privileges and completely bypass Android's application sandboxes. This paper observes that critical vulnerabilities in Android share a root cause: malicious applications that are installed and executed without the user's consent. Process authentication model addresses the problem by regarding application processes as individuals that must be authenticated before using system resources. In this model, legitimate applications are given credentials that are used for authentication at runtime. When enforcing process authentication, unauthorized processes that do not possess credentials fail to authenticate.

A secure application credential (SAC) is a unique secret issued to an application by a trusted process. Each SAC is associated with exactly one installed Dalvik or native application. A SAC is computationally hard to regenerate and must not be accessible by any unauthorized user process. The following are the method or rules involved in process authentication.

- User processes may not create or modify credentials, or, assign credentials to other processes and applications.
- If a process fails to authenticate itself with a valid secure application credential, then the process is potentially malicious.
- A process may not be authenticated with more than one credential.
- A process may not inherit its authentication status from parent processes or any other process. Sibling processes are authenticated with a shared application credential.
- A Dalvik/ART application process is always a child of the zygote process. Native processes must either be a child of an Android system process or a Dalvik

application process.

7.4 Android malware detection through kernel behavior analysis

Kernel-based behavior analysis for Android malware detection.[Isohara et al., 2011]

It proposes a kernel based behavior analysis for malware detection. As logcat is designed for application debugging only limited events are dumped into the log. Also, application level logging depends on the programmer of application. Malware usually tend to avoid logging. So an audit mechanism that can collect all activities of applications without relying on the implementation of the application is required to achieve a reliable malware detection system. The proposed system consists of a log collector in the Linux layer and a log analysis application. The log collector records all system calls and filters events with the target application. The log analyzer matches activities with signatures described by regular expressions to detect a malicious activity. Log collection is the first step of malicious activity analysis. As a kernel-level logging module collects all activities that occur on the operating system, a large number of log data are collected. To reduce the size of log data, this paper focuses on only collecting log information on activities such as process management an file I/O operation.

Process management - clone, execve, fork, getuid, getuid32, geteuid, geteuid32 File I/O - accept, bind, connect, mkdir, open, read, recv, rename, rmdir, send, stat, unlink, vfork, write.

Extraction of targeted process - Implements a process tree. For this, they first extract the PID of interesting applications from logcat messages. Since a parent process generate a child process by execution of fork()/clone() system call, tracing an interested process tree can be achieved by detecting lines that include fork or clone pattern from kernel-level log and analyzing a detected line to identify a PID of child process.

7.5 Malware analysis through system call monitor and analysis

Crowdroid[Burguera et al., 2011] will use a tool available in Linux called Strace to collect the system calls. The aim of hijacking these system calls, is to generate an output file with all events generated by the Android application. This file will provide useful information, like opened and accessed files, execution time stamps and the count of each system call number executed by the application. The last feature to represent the behavior of each Android application execution. System calls provide useful functions to the application. The number of request/executions have been made by a specific Android application is collected during the monitoring process. The framework creates as many datasets as possible.

The framework consists of three components - Data acquisition, Data manipulation and Malware analysis and Detection. Data acquisition - Data is collected and composed by basic device information, installed applications list and the result of monitoring applications with Strace tool system calls log file. Data manipulation - This component is responsible for managing and parsing all the information collected from Android users. Malware analysis and detection - This component is responsible for analyzing and clustering the feature vectors obtained from the previous phase in order to create the normality model and detect anomalous behavior in Android applications, using K-means clustering over system call count feature vectors.

7.6 Malware Analysis - a dynamic way

DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis[Kwong and Yin, 2012]

It is a virtualization based analysis approach. Analysis runs underneath the entire virtual machine. Difficult for an attack within VM to disrupt the analysis. Losses the semantic contextual information when the analysis component is moved out of the box. Reconstruct OS-level and Java-level views. Monitors how malwares Java components communicate with Android Java Framework. Monitors how malware's native components interact with the Linux Kernel. Monitors how malwares Java components and native components communicate through the JNI interface. By adding tiny code generator(TCG) to system calls, we can get the details of how a user level process access the file system and communicates with other proesses. A list of active tasks is maintained in a task struct list which is pointed to by init task. Iterate through all active tasks by following the doubly linked task struct list. We also update our shadow list whenever the base information changes. We do this by monitoring four system calls sys fork, sys execve, sys clone and sys prctl, and updating the shadow task list when they return. To obtain the memory map of a process, we iterate through the processte'list of virtual memory areas by following the mmap pointer in the mm struct pointed to by the task struct. The following modules are used.

- API Tracer - monitors how an App (including Java and native components) interacts with the rest of the system through system and library calls.
- Native instruction tracer-Gather each instruction including the raw instruction, its operands, and their values.
- Dalvik instruction tracer-Decode instructions into dexdump format, including values and all available symbol information.
- Taint Tracker Monitor sensitive information and keep track of data propagation

7.7 System call anlaysis for malicious behavior

Identifying Android malicious repackaged applications by thread-grained system call sequences[Lin et al., 2013]

This paper is about finding malicious repacked applications. This uses the method called SCSDroid (System call Sequence Droid). In this method they extract the truly malicious common subsequence of malicious applications belonging to the same family. For this they use strace.

To record the system calls at runtime, the tool strace is used.

The boot configuration of the Android emulator was changed to make strace attach to Zygote, which is a process that focuses on processing the request for executing a new Android application.

A new application is forked from Zygote and then executed in the Dalvik virtual machine architecture .

Since all of the Android applications are forked from Zygote, strace monitors all of the executed Android applications by tracing the child processes of Zygote and individually records system calls of different processes into separate files by monitoring their process IDs.

Chapter 8

Evaluation

In this chapter, the overhead of the thesis and the better methods of solving it, are discussed.

8.1 Usage of Extracted Data

In this thesis, the log files are uploaded to the cloud and then the application download files as needed, extracts details from it and updates to database. Then the database is queried and the details are provided in the user interface of the application This causes an overhead time for the Application user. Instead, if the log files are read and details are extracted and the details are uploaded to cloud as a database, the device performance will be improved. This can be achieved by using cloud as database service like Amazon SimpleDB.

8.2 Strace of Running Process

Now when a process gets terminated, Strace saves its logged data to a text file. This thesis is not considering the strace output of currently running process. The strace log files of currently running process should also be read and interpreted, by using buffering methods, to get the provenance of currently running process. This will make the thesis, more resourceful.

8.3 Encrypting the Strace Log Files

All the generated strace files are in plain text. It may disclose some sensitive information. So encrypting the generated log files, before saving on a disk, will enhance the security of the system.

8.4 Performance

Performance of the app while getting the past process details takes more than 50seconds. One of the reason for this is that, the log file is read line by line. One pass reading of file could be used to increase performance. The size of log files can also be reduced, by using only the required system calls with strace.

Chapter 9

Conclusion and Future Work

Processes are the main components, while a system is running. So when we develop a system to collect the details of the processes, we can get as much as information of the device. Knowing the activities happening inside is always helpful for analysing the device when something went wrong. Android Dashboard Application tries to provide a detailed analysis of the running and terminated processes. Provenance of a process helps in knowing the device status. It also helps in the security of the device. When the device is attacked by a malware or when a system is crashed, this helps in knowing what are all the processes running inside it. Thus helps in analysing which process created the problem. If we want to know, what all process is created, how much memory used, files opened, created and wrote this thesis will be helpful.

The source code and related documents of this thesis is available in the link https://github.com/anurp

9.1 Future Work

- Like strace, ltrace log files also should be collected and analysed to get the information related to the provenance of a process.

- When a process is running and strace is used for it, the strace log file should be opened as and when created and analysed to get the provenance of a current process. This can be achieved by using fflush() method.
- Encrypting the strace data, can help in security.

Appendix A

Creating an app console in Dropbox

A.1 Creating an app console in Dropbox

The main objective of this section is to describe the Dropbox API in Android, by providing step by step guidance and easy to understand the implementation of Dropbox API.

- Step 1 Dropbox SDK To use the Core API, download the Dropbox Android SDK and also we need to add Json Simple jar file to handle Json type request and response in your android project library(app/libs) folder. To use features of Core API, we need to compile SDK in the project. So, add Dropbox SDK and

```
dependencies {
...
compile files(  libs/dropbox-android-sdk-1.6.3.jar  )
compile files(  libs/json_simple-1.1.jar  )
...
}
```

- Step 2 Authentication Process We have to sign in and complete the authentication process in the url https://www.dropbox.com/developers/apps

- Step 3 Register App on App console Go to App Console and click on Create App button to create a new app

- Step 4 Select The Dropbox API Select the Dropbox API app
- Step 5 Set Privacy Statement Set the privacy statement access files or folders, here select App folder for accessing files which are already on Dropbox
- Step 6 Provide App Name In the textbox type your application name, if that name is already registered then you cannot create with the same name. After inputting the desired name of your Dropbox app click on Create app button
- Step 7 DashBoard We will be redirected to dashboard screen which contains all the useful parameter. We have to use that parameters in our app to access Dropbox account from our device. Here as shown in the figure below the App Key and App Secret provides unique key, which we have to pass in your application to access your Dropbox account form mobile devices
- Step 9 Enable Additional Users By default, app is in development status and has all the features available to use as production status. The only limitation of development status is that app can be used only by test users. The app can be used by up to 500 test users in development status. To make the app accessible to test users, click on Enable Additional Users button. If we want to share the app with the world, apply for production status.
- Step 9 Dropbox Implementation

– Step 9.1 Apply Secret Key

Implementation of Dropbox in our project. Apply DROPBOX APP KEY and DROPBOX APP SECRET as a constant and copy the values of both the constants from the created application previously shown in STEP 7.

```
final static public String DROPBOX_APP_KEY = " Dropbox App Key here";
final static public String DROPBOX_APP_SECRET = " Dropbox App Secret";
final static public AccessType ACCESS_TYPE = AccessType.DROPBOX;
```

– Step 9.2 Create Session which allows our application to authenticate to Dropbox API.

```
private DropboxAPI mApi;
```

Add the below lines in onCreate() method.

```
AndroidAuthSession session = buildSession();
mApi = new DropboxAPI(session);
```

Method for creating session:

```
private AndroidAuthSession buildSession() {
AppKeyPair appKeyPair = new AppKeyPair(Constants.DROPBOX_APP_KEY,
Constants.DROPBOX_APP_SECRET);
AndroidAuthSession session;
String[] stored = getKeys();
if (stored != null) {
AccessTokenPair accessToken = new AccessTokenPair(stored[0],stored[1]);
session = new AndroidAuthSession(appKeyPair,
Constants.ACCESS_TYPE,accessToken);
} else {
session = new AndroidAuthSession(appKeyPair, Constants.ACCESS_TYPE);
}
return session;
}
```

– Step 9.3 AndroidManifest.xml file

```
<intent-filter>
<!-- Change this to be db- followed by your app key -->
<data android:scheme="db-n81vuqu3mfexf6i" />
[...]
/intent-filter>
```

In the above AndroidManifest.xml file there is a line android:scheme, which contains the key. We are required to change the key value of our application which we can find form Step 7.

- Step 9.4 Start Authenticating Process Apply the code below in onActivityResult() method to complete the authentication process.

```
mApi.getSession().startAuthentication(Main.this)
onResume = true;
@Override
protected void onResume() {
AndroidAuthSession session = mApi.getSession();
if (session.authenticationSuccessful()) {
try {
session.finishAuthentication();
TokenPair tokens = session.getAccessTokenPair();
storeKeys(tokens.key, tokens.secret);
setLoggedIn(onResume);
} catch (IllegalStateException e) {
showToast("Couldn t authenticate with Dropbox:"
+ e.getLocalizedMessage());
}
}
super.onResume();
}
private void storeKeys(String key, String secret) {
SharedPreferences prefs = getSharedPreferences(
Constants.ACCOUNT_PREFS_NAME, 0);
Editor edit = prefs.edit();
edit.putString(Constants.ACCESS_KEY_NAME, key);
edit.putString(Constants.ACCESS_SECRET_NAME, secret);
edit.commit();
}
```

References

ALMOHRI, H. M., YAO, D. D., AND KAFURA, D. 2014. DroidBarrier: Know what is executing on your Android. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 257–264. `http://people.cs.vt.edu/danfeng/papers/spy008-almohri.pdf`.

BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. 2011. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 15–26.

ELENKOV, N. 2014. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press.

EOLWRAL. 2015. Os monitor. `https://github.com/eolwral/OSMonitor`.

GEHANI, A. AND TARIQ, D. 2012. Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 101–120.

George, N. 2015. Network ombudsman for Android. M.S. thesis, Amrita Vishwa Vidyapeetham, Ettimadai, Tamil Nadu 641112, India. Advisor: Prabhaker Mateti.

Google. 2016a. Connectivity manager. `https://developer.android.com/reference/android/net/ConnectivityManager.html`.

Google. 2016b. Wifi manager. `https://developer.android.com/reference/android/net/wifi/WifiManager.html`.

Gurung, P. 2015. Android's architecture diagram. `http://newtechinfo.net/android-system-architecture/`.

Inc, D. 2015. Dropbox cloud upload. `https://www.dropbox.com/developers-v1/core/start/android`.

Isohara, T., Takemori, K., and Kubota, A. 2011. Kernel-based behavior analysis for android malware detection. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*. IEEE, 1011–1015.

Kerrisk, M. 2016. System calls. `http://man7.org/linux/man-pages/man2/syscalls.2.html`.

Kwong, L. and Yin, Y. H. 2012. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis.

Lin, Y.-D., Lai, Y.-C., Chen, C.-H., and Tsai, H.-C. 2013. Identifying Android malicious repackaged applications by thread-grained system call sequences. *Computers and Security 39*, 340–350.

Linux Kernel Organization, I. 2016. The linux kernel archives. `https://www.kernel.org/`.

Mateti, P., Aiyyappan, P., George, N., Kamardeen, J., Sahadevan, A. K., and Shetti, P. 2015. Design and construction of a new highly secure Android ROM. Tech. rep., Amrita Vishwa Vidyapeetham, Ettimadai, Tamil Nadu 641112, India. 6. Advisor: Prabhaker Mateti; `http://cecs.wright.edu/~pmateti/GradStudents/index.html`.

Rusling, D. A. 1996 - 1999. Linux processes. `http://www.tldp.org/LDP/tlk/kernel/processes.html`.