MAPPING INFORMATION LEAKS IN THE ANDROID OS

By

AJOYKUMAR RAJASEKARAN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

written under the direction of

Prof. Janne Lindqvist

and approved by

_____

_____

_____

New Brunswick, New Jersey

October, 2016

ABSTRACT OF THE THESIS

Mapping Information Leaks in the Android OS

By AJOYKUMAR RAJASEKARAN

Thesis Director:

Prof. Janne Lindqvist

Android OS enforces security and privacy by means of sandboxing to isolate the execution of one application from another and by means of system permissions to restrict access to sensitive information. Android system permissions are designed to let users decide which permissions are allowed for an app. In this thesis, we show attacks to retrieve user information from Android devices by exploiting the Android framework. We also built a classifier for inferring the everyday activities of the users. This classifier has on average an accuracy of 97.9%, precision of 89.09%, specificity of 98.85%, sensitivity of 88.9%, and an F-score of 88.42%.

Acknowledgments

I would like to express my sincere gratitude to my advisor Professor Janne Lindqvist for giving me this opportunity, along with his encouragement and continual support in my pursuit for academic excellence. It was a great experience and a pleasure working under his guidance and supervision. Without his help this thesis would not have been possible. I would like to thank and appreciate Hua Deng for all the help and feedback. It certainly helped me gain a better understanding of the work at hand. I would also like to extend my gratitude to all of the lab members, who have been a constant support and made this journey an enjoyable one.

I am ever so grateful to my family and friends for all the belief and support they have showed in my work. Thank you all.

This research is based upon work supported by the National Science Foundation under Grant Numbers 1228777. Any opinions, findings, and conclusions or recommendations expressed in this research are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview of Android API

Android OS is an open source platform and application environment based on the Linux kernel. The Android OS consists of a stack of software components, which provides the application framework used by the application developers. The application framework APIs provide the functionality to communicate with the system services to access the underlying smartphone hardware. The Android applications run on the Dalvik virtual machine, and the application can be installed from a single apk file [1].

The framework APIs are updated such that the new API remains backward compatible. Every Android platform supports only one API level, with implicit support for older API levels. During installation the system checks the minSdkVersion and maxSdkVersion in which it was built to run on. The API level specified allows the system to only install a version compatible application [2]. The Android framework API consists of packages and classes, XML elements and attributes for declaring a manifest and accessing resources, broadcast intents and set of permissions that the applications can request to access the otherwise restricted system resources.

1.2 Android Permission Model

Every application has an AndroidManifest.XML file, which is used as a control file to tell the system what components such as broadcast receivers, content providers, etc. and permissions are requested by the application [1]. The Android applications are run in an application Sandbox and can only access a limited set of system resources. The Android system manages and restricts access to resources that, if used incorrectly or maliciously, will result in an

adverse impact on the user in the form of a privacy or security breach. On the Android platform the restrictions are implemented as follows:

- Intentional lack of API to access sensitive functionality. For example, there's no API to manipulate a SIM card.
- Enforcing security through separation of roles. For example, every application runs as a different UID, and has its own isolated storage which cannot be accessed by other applications.
- Sensitive APIs that provide access to sensitive resources such as Camera, are restricted to be used by only those applications that explicitly request these resources in the AndroidManifest.XML file.

Starting with Android 6.0 with API level 23 and later the users are prompted during runtime of the application to either allow or deny the access to a resource requested by the application. Once allowed by the user, the system does not prompt the user again about the use of sensitive resources. But the users can navigate to device system settings to view the permissions for the installed applications and the user can revoke previously allowed permissions.

The Android permission model implements privacy by restricting access to sensitive resources and giving the control of these sensitive resources to users. If the installed application was targeted for a lower API level, then the user will be prompted during install time to either accept and install all the applications or to not install at all. But the users have the choice to revoke any of these permissions in the device settings after the installation. Thus, even on newer Android versions, users can still be forced to grant all the permissions during the install time. The Android permissions are not completely understood by users, but a minority of users have found these permissions useful to avoid privacy-invasive applications [3].

1.3 Motivation

Android system permissions are designed to let users decide which permissions are allowed for an application. These permissions should restrict access to information that is sensitive to users. However, we show that these permissions do not compartmentalize the sensitive information. Android system permissions are divided into several protection levels. The two most important protection levels to understand are those with normal and dangerous permissions.

The normal permissions are requested by the application in the AndroidManifest.XML. The users do not have control over these permissions because Android does not identify the resources protected by these permissions as sensitive. These permissions cover areas where the application needs to access data or resources outside of the application's sandbox but where there's very little risk to the user's privacy or the operation of other applications [2]. For example, permission to turn on the flashlight by any application is a normal permission. If an application declares that it needs a normal permission, the system automatically grants the permission to the application. Some of the features requiring normal permissions are: Network State, Notification, Bluetooth, Flashlight, Internet, and Vibrate.

Dangerous permissions cover areas where the application wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other applications [2]. Some of the dangerous permissions include access to camera, contacts, location, microphone, body sensors (e.g.heart rate sensor), external storage, etc. If an application declares that it needs a dangerous permission, the user has to explicitly grant the permission to the application. The broadcast receiver APIs can be implemented in an application that requires no special permission and we can infer the state of the smartphone. The results from broadcast receiver can be used to correlate with results from activity detection to infer the lifestyle of the users. Information from files which can be accessed globally by all applications can be used to infer user's personal information. Such a private information leak will confirm that the system is vulnerable to side channel attacks on Android.

1.4 Problem Statement

If the smartphone is running Android version 5.1 or lower, or if the application's target SDK is 22 or lower, the user has to grant all of the requested dangerous permissions to install the application [2]. The user is only given the choice between installing the applications allowing all of the permissions or not installing any of the applications. If using a smartphone with Android version 6.0 or higher and if the application's target SDK is 23 or higher, the application will list all of the dangerous permissions in the manifest, and it must request each dangerous permission during the runtime [2]. In this way the user can install the application without the hassle of reading the permissions list. When prompted, the user can deny or allow each permission and the application can continue to run with reduced capabilities even if the user denies a permission [2]. If the target SDK is lower than 23, then all of the permissions requested by an application must be granted to install the application, but the users can still revoke any permission for any application at any time [2].

The hardware compatibility limitations of the devices and curtailed support from the manufacturers limits the devices from having the latest version of Android OS. Applications built using a lower SDK can be run on devices having the same or higher OS version, due to backward compatibility. While building an application, developers can set the minSdkVersion for the application. Since the developers want to target as many users as possible, they develop applications with the lowest possible SDK version.

In this work we focus on using APIs provided by the Android framework to perform side channel attacks and infer private information about the users. We use a broadcast receiver API to infer information about the state of the smartphone. To infer activities performed by the user we use the activity recognition API. These APIs were targeted for SDK level 23 with support for a minimum SDK level of 15. We built an application to read external storage to show that it is capable of inferring user's private information such as contacts, location and email conversation. The results from [4] show that users view and read the permissions, but these permissions are not understood. The users are unaware of why some permissions are being requested by the applications. Therefore, APIs that require permissions can also be exploited for side channel attacks.

Chapter 2

Related Work

Every smartphone today has embedded motion sensors that could be exploited for obtaining user's private information. These sensors include accelerometer, gravity sensor, magnetometer, orientation sensor, proximity sensor, step detector and counter. The data from these sensors are available without any special system permission. To infer any useful information from these sensor data, we need to collect the data over a period of time for a particular activity. This training data is then used to construct a predictive model for activity recognition [5][6][7][8][9]. A Trojan application could be installed that utilizes accelerometer and orientation sensor readings. These readings can be used to monitor any movement and gestures made by the users. It may also stealthily log the passwords of screen lock or numbers entered during a phone call [5]. Side channel attacks using motion sensors to track user input is possible because typing different keys on a smartphone's soft keyboard has unique associated motions of a smartphone [6]. Motion sensors can be used to detect much more than activity recognition. Researchers have shown that accelerometers can be used to extract entire sequences of text entered on the smartphone by using a training data set of about 2700 key presses [10]. This potentially allows attackers to first infer which page the user is browsing [11] and then read the passwords typed by users [10]. Activities such as walking, jogging, climbing stairs, sitting and standing can be easily detected by only collecting data from an accelerometer [8]. The authors in [12] provide an extensive review on activity recognition by only using accelerometer in which they provide information using machine learning algorithms to detect various activities. Results from [13] show that using a gyroscope sensor in combination with an accelerometer increases the recognition accuracy by 3.1-13.4%. The experimental results in [44] for a multi-sensor system, report an accuracy of 97.9% for static activities such as standing, sitting, etc. and 94.0% for dynamic and

transitional activities such as walking. The Android framework now provides an activity recognition API that allows developers to easily infer the user's activity without the need for complex algorithms to detect activities. All of the previous research is only focused on detecting activities such as walking, jogging, running etc., and did not attempt to infer the lifestyle of the user, e.g. how often does the user charge his or her phone, when does the user go to sleep and how often does the user unlock the smartphone. In our work we collect the broadcast receiver events to infer the state of the smartphone, and correlate these events with physical activities detected by the activity recognition API to have a deeper insight into the lifestyle of the user. We also present a classifier design and study its performance in classifying the activities performed by the participants.

Applications such as games with access to a microphone or camera can be used to collect image captures while the user is interacting with the application [7]. The microphone sensor can be used to detect touch events, while the camera is used to infer the PIN entered by the user using the features extracted from the image [7]. The authors in [14] show that keypad numbers can be inferred with more than 60% accuracy by only using the smartphone camera to track eye gazes. The microphone sensor can also be used to listen to the background noise to infer the environment in which the user is present. For example, we can infer if the user is traveling by listening to airport noise and in-flight announcements can be used to determine the user's destination. The drawback of this attack is that it requires the user to grant access to the microphone and camera.

As the functionality and applications of the smartphone has increased, so has access to sensitive information about the users [15]. Linux file permissions are used on Android to protect the private files to one application from being accessed by other applications. The results from work in [11] show the pervasiveness of the Android side channel attacks. This attack exploits the information from the /proc/ virtual file system, which is also known as the process information file system and it contains runtime system information like the system memory, mounted devices, hardware configuration, etc. Side channel attacks on Android can be accomplished by a concurrent application that tracks the changes in the application's memory footprint [11].

As of March 2016, Android play store has more than 2 million apps [16] and the developers

building these applications are more focused towards features and profitability by targeted advertisements. These developers do not focus on security and privacy issues of applications, and use poor programming practices which potentially leads to applications being hijacked by other malicious applications. Each application runs on Android as a unique user to limit the potential damage of programming flaws [17]. To prevent applications from leaking information to other applications, each application runs inside a sandbox [18]. To developers the range of devices provides a huge potential audience for their application [2]. Hence they support even older devices to reach as many users as possible. Due to space constraints of the internal storage on older devices, with support to expand memory using a SD card, the developers tend to store large files on external SD card storage. We use this knowledge and look into the cache files stored by the default applications in the external SD card storage. Applications can extend their functionality by dynamically loading classes during runtime using DexClassLoader API. This can be used to execute code that is not installed as part of the application [2]. But when developers use external storage to store these class files, which are also visible to other applications with access to external storage, a malicious application can inject code or modify the class files to hijack the application [19].

Chapter 3

Side Channel Attacks

3.1 Inferring Information About Users

3.1.1 Using an Application with Permission to Read External Storage

Setup

We used a Nexus 5 smartphone running Android - M version 6.0.1. The smartphone had its Wi-Fi turned on and connected to a network. The location feature was turned on, and we granted all the permissions requested by the default applications. To inspect the cache files stored by the system and user applications we used the Android debug shell to pull the files residing in the sdcard/Android/data/ directory. We then built another application to show that these files can be accessed by any user level application with permission to read external storage. To investigate further into private information stored within the cache files on external storage, we extracted the cache files from a user with a Moto G 2nd generation phone also using the Android debug shell.

Investigation

The Android permissions are designed to enforce user privacy, by restricting access to sensitive information to only those applications that explicitly request permissions to sensitive resources. Only when a user grants permission to these resources can an application use the corresponding APIs, provided by the Android framework, to access sensitive information. No application should be able to adversely affect other applications, the operating system or the user [2]. The application must not be able to read or write the user's private data such as contacts or emails, and one application should not be able to read or write other applications files [2]. While investigating the external storage on the Nexus 5 smartphone, we noticed

the directory /sdcard/Android/data/ which contained sub-directories whose names were the same as the package name of the application which created it. Within these directories we observed some interesting cache files stored by Google Maps and Google Camera on Nexus 5 smartphone.

We show that by having the permission to read external storage, an application can get the following information about the users from Google Maps and Google Camera respectively:

- User's Google account user name which is protected by android.permission.GET_ACCOUNTS.
- User's location information such as altitude, longitude and latitude which is protected by android.permission.ACCESS_COARSE_LOCATION and android.permission.ACCESS_FINE_LOCATION.

The Google Maps application on Nexus 5 stores cache files with the name cache_vts_psm_GMM.# in the external SD card storage directory /sdcard/Android/data/com.google.android.apps.maps/cache/. The Google Maps application creates a new cache file with an incremented number for # in the file name when the user adds a new account to the application, and all of these files are persistent between system reboot. We used the Android debug shell to pull these files into a local computer and opened the files in a text editor to view the file contents. These cache files contain the Google account name of the user in plain text. When a new account is used a new cache file is created cache_vts_psm_GMM.2 which now contains the new account name along with the older account name. These cache files are not cleared by the system and remain between system reboots. Thus we have shown that the account name of the user can be retrieved without having the permission android.permission.GET_ACCOUNTS.

We now move on to the cache the file stored by Google Camera application in the external directory /sdcard/Android/data/com.google.android.GoogleCamera/cache/. This directory contains two sub-directories "TEMP_SESSIONS" and "panorama_sessions." The directory "TEMP_SESSIONS" contains a sub-directory with the name format "PANO_YYYYMMDD_#" where # is a unique session number. These directories contain the final images from the panoramic image capture sessions. The directory "panorama_sessions"

contains a sub-directory with the name format "session_YYYYMMDD_#" where # is a unique session number. These directories contain the individual images that are used to make up the final panorama image, an orientation.txt file that is used by the camera application to determine the orientation of the smartphone between individual image captures, and a session.meta file. The session meta file contains information about the user's location's altitude, latitude and longitude. It also gives information about the time at which the user captured these images. We have thus shown from our investigation that we can infer user's Google account name, and location information from cache files stored in the external SD card storage. Now we move on to show that any application with permission to read external storage can access these cache files.

Using the Android debug shell we observed that the directories in /sdcard/Android/data/ corresponding to the Google Camera and Google Maps application belong to the group "sdcard_rw." Directories which belong to this group can be read and written to by applications that have the permission to read or write to external storage. Thus an application with permission to read external storage can read a user's account name and location information. We implemented an application which reads these cache files and displays its contents.

For our next side channel attack, we used a Moto G 2nd generation phone. The user was using the default or the stock Email application for everyday use. This Email application was using the external directory /sdcard/data/com.android.email/cache/ to cache some of the recent email threads. The files were stored with .eml extension, a common file format developed by Microsoft for use with Outlook. The eml file format is very similar to HTML and we viewed the email contents in the Chrome browser by just changing the extension to *.mht which is a web page archive file format. We found that the application had stored close to 200 such cache files.

We investigated the cache files and found that the cache files contain the synced emails encoded using Base-64 as shown in figure 1. The file contents can be viewed in decoded plain text using any browser. These cache files contain conversations that the user had in the email thread. It also gave us the senders and recipients email address. While investigating further into the email contents, we stumbled upon a link to Google sheets. Any application having access to this Google sheets link can view the file that was shared using the link.

We opened the link and found that it was a file containing emergency contacts of several students going on a trip. Such sensitive information can be accessed by any application having permission to read external storage and the information can be used for any malicious intent such as phishing, spamming and identity theft, thus posing a great privacy risk.

Figure 1: The figure shows the file contents of a cache file stored in the external storage by the default email application on Moto G 2nd generation. The cache files contain the email threads that were synced by the email application. Thus we can obtain the email contacts of the sender and the recipients. The email contacts are redacted to preserve privacy of the user who participated in this study. We can also read the email conversation in plain text by decoding the content encoded using Base-64.

Results

The permission to read the user's contacts is recognized as dangerous permission by [2], and hence an application needs to explicitly request to access the user's contact by declaring the permission android.permission.READ_CONTACTS. Also for an application to read the information about any account and also to obtain the Google account name used to sign in to Google Maps, it requires the following permission android.permission.GET_ACCOUNTS.

We built an application to read the contents of cache_vts_psm_GMM.# and used regular expression to extract and display only the email addresses. And the application also displays the contents of the session.meta which gives the user's location information. During installation the application requests the user to grant the permission to read contents of USB storage. We specified a target SDK level of 21 for our application and since the target SDK

is lower than 23, the application requests the user to grant the permission during install time [2]. But what the user does not know is that granting this permission will allow the application to access the user's Google account name and user's location information. The application that we built was successfully installed with permission to access external storage. The application is capable of accessing the user's account name and the location information as shown in figure 2. Thus we have shown that any application with permission to read external storage is capable of obtaining user's account name and the location information.

Figure 2: This figure shows the application we built to read cache files. The application has the permission to read external storage and uses a SDK level of 21, thus the permission to read external storage was granted during install time. The application reads the Google Maps cache file to display the account name, which is the email address used to sign up on Google Maps. The application also reads the Google Camera cache files, and displays the location information of the user that was stored in the session meta file.

The Android system permission is designed to prevent unrestricted access to sensitive user information. An application having permission to access information from a resource should not be able to get information protected by other permissions. But here we have shown that if the user grants permission to read external storage, then the malicious application will be able to read the user's account name and obtain a user's location information, which are explicitly protected by android.permission.GET_ACCOUNTS, android.permission.ACCESS_-COARSE_LOCATION and android.permission.ACCESS_FINE_LOCATION.

We have shown that email conversations can be read by an application with access to external SD card storage and we can learn personal information about the user and the user's contacts. The permission to read user's account name, location and contacts is listed as dangerous permission because they are highly sensitive information that could be used against the users. But having permission as simple as reading external storage, completely nullifies the importance of other dangerous permissions. Thus Android system permissions are not fool proof in compartmentalizing the sensitive information.

3.1.2 Using an Application to Infer User Activity

Setup

We designed and implemented an application on Nexus 5 that used the activity recognition API to infer physical activities performed by the user. To infer the information about the state of the smartphone, we receive and collect broadcast events from the smartphone.

Application Design

The application has a start/stop activity detection button that allows us to control the collection of the detected activity into the SQLite database. On clicking the Update Activity button the activity detected by the activity recognition API is stored into the database along with the confidence value provided by the API for each detected activity. The application collects the broadcast receiver events for the registered intents in the application and activity detected by the activity recognition API along with the timestamps. The timestamp allows us to correlate the broadcast receiver events with the detected activities. Here we have shown how an application running in the foreground can be used to infer user's activities with easily available APIs. In an actual scenario the attacker can automate data collection, and also it could be done in the background as a service without the knowledge of the user.

Activity Recognition and Broadcast Events

We implemented an application that uses the activity recognition API provided by the Android framework to infer whether the user is Still, On Foot, Walking, Running, On Bicycle,

in a Vehicle and if the phone is tilted (facing down). The physical activities detected by the API are stored in a SQLite database along with the timestamp and confidence value for each activity. The reason to store all the detected physical activities with their corresponding confidence value and not just the activity with high confidence is that, if the "On Foot" activity is detected with highest confidence, then the next highest confidence activity will tell us if the user was walking or running. The activity recognition by the API is done periodically by reading short bursts of sensors data, so that the battery consumption of the application is kept to a minimum. The update interval of the activity recognition is controlled by an update interval parameter in the methods provided by the API. If any other application uses this API and requests updates at a faster interval, then the fastest interval takes precedence [2]. After recording the activity information, we used Android debug shell pull the database to a local computer and then used SQLite browser to view the contents of the database.

The application receives the broadcast events to infer the state of the phone. The application registers for the following broadcast intents:

- Intent.ACTION_AIRPLANE_MODE_CHANGED

  By registering for this broadcast intent we can infer when the user toggles the airplane mode button. But in order to know if it was turned ON or OFF we check for the returned value of the method Settings.System.getInt(getCOntentResolver(),Settings.System.AIRPLANE_MODE_ON,0). If the return value is '0' then the Airplane mode is OFF, otherwise it is ON [2].

- Intent.ACTION_BATTERY_LOW

  The application receives this broadcast intent when the battery level of the smartphone is low [2].

- Intent.ACTION_HEADSET_PLUG

  We can detect toggling of headset plug in/plug out by registering for this broadcast intent [2]. We keep a toggle counter to keep track of plug-in and plug-out events. A

counter value of “1” is for plug-in and “0” for plug-out.

- Intent.ACTION_POWER_CONNECTED

  The application receives this broadcast intent when the user connects the smartphone to any power supply [2] - wall socket, computer or power bank.

- Intent.ACTION_POWER_DISCONNECTED

  The application receives this broadcast intent when the user disconnects the smartphone from any power supply [2] - wall socket, computer or power bank.

- Intent.ACTION_SCREEN_ON

  This broadcast events are received when the smartphone wakes up and becomes interactive [2].

- Intent.ACTION_SCREEN_OFF

  This broadcast events are received when the smartphone goes to sleep and becomes non-interactive [2].

- Intent.ACTION_USER_PRESENT

  This intent is used to determine when the user successfully unlocks a screen.

Experimental Methodology

We used five participants for an informal outside lab experiment. The participants were given the Nexus 5 smartphone. The participants were told to open our application, and turn on the activity recognition using the button provided in the application. We then informed the participants to manually click the “update activity” button shown in figure 3, every one minute interval. All the participants were from Rutgers University, and were accustomed to using an Android smartphone. The user activity study was conducted in the CORE building on the Busch campus in Piscataway, NJ. One participant was allowed to travel around Busch campus in the University bus. On completion of the study with each participant, we used the Android debug shell to pull the database and observed the detected

physical activities and broadcast events using a SQLite browser. We informed the user of the detected activities and verified the correctness of activity recognition.

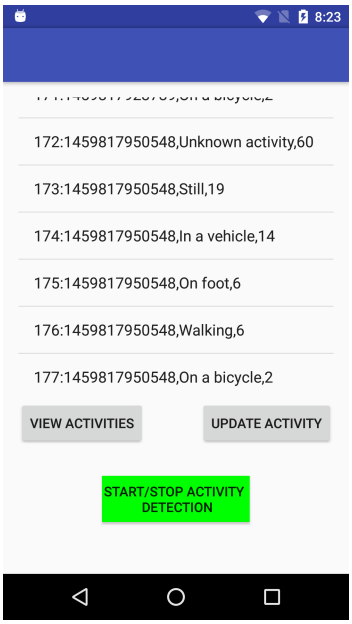

Figure 3: The user interface of the application that was used to infer user's activities using the activity recognition API is shown above. The user has to click the start/stop activity detection button in the beginning and end of an experimental session. While the user is performing the activity, the user is instructed to click the update activity button. This stores the detected activities in the database.

Results

We collected and analyzed the detected activities and broadcast receiver events in a SQL database. The application stores the confidence value of all of the possible activities that the activity recognition API is capable of detecting. This is done so that if the activity corresponding to the highest confidence value for a user is on-foot, then based on second highest confidence value, we can determine if the user was walking or running. We observed from our collected samples that when the application detected a user to be on-foot, then it is able to detect whether the user is walking or running with a very high confidence. In our experiment we were able to detect on-foot activity with a confidence value of 60% to 90%, and for the same activity session we were able to detect walking or running with a confidence value of 55% to 80%.

We show that the activity recognition API can be used to detect the orientation of the

phone. When the recognized activity is “still” we can infer that the user was using the phone while standing or sitting still. Since the activities are collected along with timestamps, we were able to infer that immediately after using the phone the user kept the phone on the table, with the phone facing downwards because the activity recognition API detected that the phone was tilted and still. We initially deduced that the user was not moving during this time, because no other activity was detected. Hence, the tilting action was that of placing the smartphone on the table with the smartphone display facing downwards. But the user informed us that he was just standing and flipping the smartphone in his hand.

Activity recognition API allows us to detect the physical activities performed by the user. This API requires the permission com.google.android.gms.permission.ACTIVITY_-RECOGNITION. Although we used the activity recognition API here, it is possible to recognize the various physical activities by only using the motion sensor measurements [20][21][8][9][12]. And these sensor measurements do not require any permissions and thus any application can infer the user’s physical activities without the knowledge of the users. Here we used the API since it is an easy way to detect physical activities rather than creating a learning data set from raw sensor readings and then developing predictive models to infer activity from sensor data. In section 3.1.3 we show how we can infer other activities using only the motion sensor data.

We can infer if the phone is on charging and for how long by listening to ACTION_-POWER_CONNECTED and ACTION_POWER_DISCONNECTED. If the duration of charging is longer than six hours and there is no activity detected by the API, we can infer that the user is likely to be sleeping at that time and not using the smartphone. Based on the duration of resting and active hours, we can infer the sleep cycle of that user. ACTION_BATTERY_LOW intent can be used to infer when the user is alerted of low battery. If we detect the user to be walking or running before and after power connected broadcast events, then we can conclude that the user has a power bank.

One of the participants was asked to perform any actions on the smartphone. The participant was provided with a headset and USB charging cable. Based on the time-stamped broadcast receiver events, we inferred that the participant performed the following sequence of actions - unplugged a headset, plugged in the headset, disconnected power, turned

on the airplane mode and then connected the power supply again. Using the timestamp, we looked into the activities detected during this period. We observed that the activity recognition API detected, with 100% confidence, that the participant was still during this time. Later, the participant confirmed that he was siting and standing during the experiment session and only walked inside the room for a short duration. Since the participant did not perform walking for a significant duration of time the activity recognition API was not able to detect this activity.

During every broadcast receiver event we collect information based on whether the smartphone is in vibrate mode, whether vibrate on ring is turned on, and also the brightness level of the phone. This information is obtained by using the following methods:

- Settings.System.getInt(getContentResolver(),Settings.System.VIBRATE_ON, 0)
- Settings.System.getInt(getContentResolver(),Settings.System.VIBRATE_WHEN_RINGING, 0)
- Settings.System.getInt(getContentResolver(),Settings.System.SCREEN_BRIGHTNESS, 0)

These methods need to be invoked every time to get the information, thus they are placed inside broadcast receiver methods. Whenever a broadcast event occurs the above methods will be invoked. From the sequence of broadcast receiver events for one of the participants, we were able to infer that the participant was constantly locking and unlocking the while using the smartphone. By using the timestamp of these events we looked into the physical activities detected during this time and observed that the user was detected to be on foot and walking. The participant confirmed that he had set the display to sleep after 15 seconds, and that he had to constantly unlock the smartphone to update the activities in the application while walking in the hallway.

The final participant for this experiment was allowed to go around the campus. From the broadcast receiver events during his trip we observed that the participant had unlocked the device, and then connected the device to a power supply to charge the smartphone. The participant must have connected the phone to a power bank or some other source of

power supply. During the time period this broadcast event was received, the participant was detected to be traveling in a vehicle.

We were able to detect physical activities performed by the participants and correlate these physical activities with the broadcast receiver events if any, to infer detailed information about the activities performed by the users. By using broadcast receiver events we were able to infer the activities performed by the user on the phone as well as motion activity.

The activity recognition API allows us to infer only activities such as on foot, walking, on a bicycle, running and still. When the user is performing any other activity the API will recognize the activity as unknown activity. Thus we cannot rely only on the API to infer user activities. For this reason we created another application as described in the following section, that allows us to collect labeled motion sensor data for various activities, and then use the collected data to build a classifier for activity detection.

The broadcast receiver events allow us collect various events that occur on the smartphone. These broadcast events alone are very trivial, such as whether the headset is plugged in, power connected, and if the airplane mode is on or off, etc. But when these events are correlated with activity detected from the activity recognition API, we can infer private information about the user such as when the user connects the phone to charge, if the user uses the phone during this period of time, and if the user constantly walks or runs with headset plugged in. Such data correlation gives us a deep insight into the lifestyle of a user.

3.1.3 Using Labeled Sensor Data

The drawback of using activity recognition API is that, it can only detect a small set of activities. To infer more about the activities performed by the users, we collected labeled sensor data for activities listed in table 2. For each of the above activities we collected the raw sensor data from the accelerometer sensor, gravity sensor, gyroscope and linear accelerometer and stored them in a SQLite database under the corresponding sensor table. The motion sensors detect and return the sensor event values of the device along x, y and z axes in a static frame relative to the phone as shown in figure 4 [22]. The sensor measurements are always made along these static axes direction irrespective of the device's screen orientation changes.

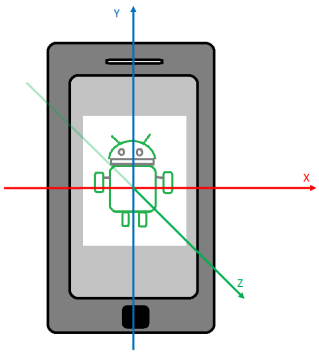

Figure 4: X, Y and Z axes are static relative to the smartphone. The motion sensors on the smartphone measure the sensor values along these axes. If the smartphone moves along the direction pointed by these axes, then the sensor values will be positive. For example, when the device is still, and kept facing upwards on a table then the acceleration read along z axis will be positive and equal to the value of gravity.

The accelerometer sensor measures the acceleration along the three axes. The acceleration is measured in m/s^2. The measured value includes both the rate of change of velocity of the device and the force of gravity on the device along that axis. So when the device is in free fall, acceleration along x, y and z should be close to zero. When the device lies flat on a table facing upwards and is pushed right, the x acceleration value is positive and the z acceleration is +9.81. When the device is lifted up the z acceleration is greater than +9.81 [22]. The linear accelerometer measures the linear acceleration of the device along the axes in m/s^2, but the measured value does not include gravity. When the device is at rest the output along all the axes should be zero [22]. The gravity sensor measures the direction and magnitude of gravity along the x, y and z axes. It is also measured in m/s^2. When the device is at rest the sensor and accelerometer sensor output should be identical [22]. The gyroscope sensor reports the rate of rotation or the angular speed around the sensor axes in rad/s. We collected labeled and time stamped sensor measurements for each of the activities to build a classifier.

Setup

We built an application to collect labeled sensor data from users and to build a classifier in order to detect activities that cannot be recognized by the activity recognition API. The application had a drop down list of the above mentioned activities as shown in figure 5. The

start/stop sensor recording button will allow us to collect labeled sensor data for activities segregated into trials. A start and stop of sensor recording will give us one trial of labeled sensor data for that activity.

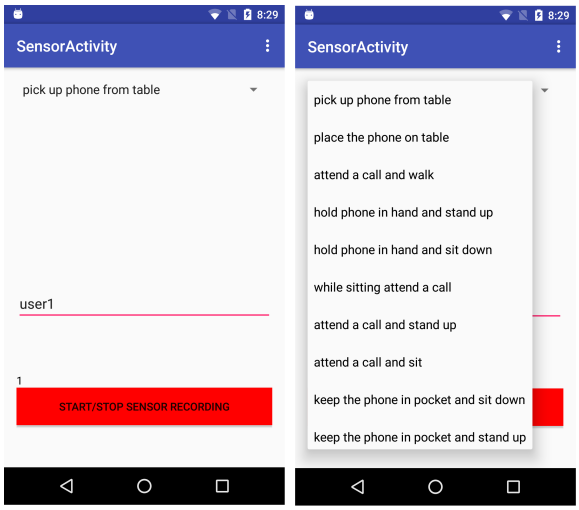

Figure 5: The application that we built to collect labeled sensor data. The participants will first select an activity to perform from the drop down menu. Then the participants will enter his/her name in the text field provided. The participants were instructed to start the sensor recording before beginning the activity, and stop the sensor recording after completing the activity by using the start/stop sensor recording button. The participants were then instructed to perform each activity for five times.

Experimental Methodology

We used six participants, for an informal outside lab experiment. The participants were between the ages 23 to 26. The participants were given the Nexus 5 smartphone and told to open the application, and enter their name in the text field provided at the bottom as shown in figure 5. They were told to choose an activity from the drop down list, and then start the sensor recording before they begin to perform the activity, and then stop the sensor recording after they have completed the activity. Participants were asked to perform each activity for five times. An incremental number was shown on the screen to indicate the number of trials performed. This number allowed us to uniquely distinguish the samples of each activity. After each participant had performed all the activities, we used the Android debug shell to pull the database into a local computer for analyzing and running our classifier on the labeled sensor data.

_id	timestamp	x	y	z	activity	trial	user
342	1464242500450	-2.2027588	2.8791351	7.3841705	0	3	participant1
343	1464242500470	-3.016449	2.543274	6.697647	0	3	participant1
344	1464242500489	-2.9492798	2.811798	8.627319	0	3	participant1

Table 1: Labeled accelerometer sensor data for one of the participants. The trial entry uniquely identifies one sample from another for each activity. Each activity is assigned a unique identifier.

ACTIVITY	IDENTIFIER
Picking up the phone from the table	0
Placing the phone on the table	1
Attending a call and then walking	2
Holding the phone in hand and standing up	3
Holding the phone in hand and sitting down	4
Attending a call while sitting	5
Attending a call and then stand up	6
Attending a call and then sit down	7
Keep the phone in pocket and sit down	8
Keep the phone in pocket and stand up	9
Keep the phone in pocket and walk	10

Table 2: Activities and their corresponding identifiers used in explaining the sensor data plotting, performance metric plots, and the confusion matrix.

Sensor Data Analysis

In this section we discuss some of the features that can be observed from the activities that the participants performed during the informal study. Each participant performed the activities in their own style. For example, one participant always used their right hand to pick up the phone, while another participant used their left hand for some trials and right hand for other trials. A labeled sensor data for the accelerometer is shown in table 1 for one of the participants. The activity column has the identifier number for the activity performed, and the trial number distinguishes one sample from another. The identifier for each activity is shown in table 2, and from now onwards we will refer to activities by their identifier.

Before we move on to the design of the classifier for activity recognition, we will discuss some of the information that can be inferred directly from the raw sensor data. Consider the gravity sensor data for activity 0 plotted against the normalized time stamp for user 1 and user 2 shown in figure 6. The x, y and z components of the gravity sensor corresponds to a static frame reference as shown in figure in 4. We can infer from the figure that, the spikes

and transitions of y and z-gravity at around 1000 ms for both the users occurs while the phone is just lifted of the table. The positive transition of y-gravity and negative transition of z-gravity indicates that the phone is being lifted off the table. The small positive spike, followed by a negative spike, for x-gravity for user 1, indicates that the phone was slightly tilted to the left then to the right while it was lifted. The small duration of transition, followed by a smooth x, y and z positioning from 2000 ms onwards, indicates that user 1 lifted the phone rather quickly to its normal reading position. Whereas, for user 2, there is a large positive spike for x-gravity, indicating that the phone almost completely titled to its left while being lifted, before holding the phone in reading position from 2500 ms onwards. While the phone is held in reading position, the x-gravity is due to either noise or slight tilting of the phone to left or right.

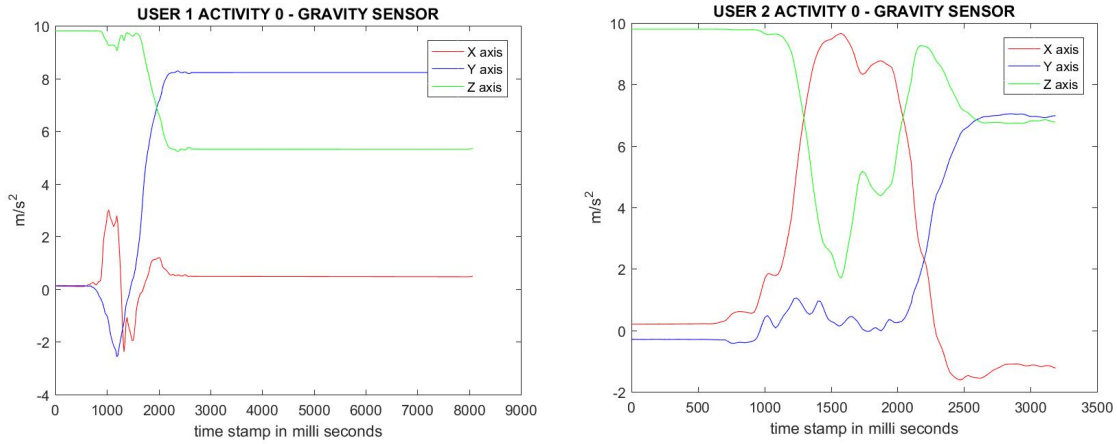

Figure 6: Gravity sensor data for activity-0 (picking up the phone from the table) is plotted against the timestamp for two users. The plot on the left shows that the user-1 picked up the phone quickly and then the user was holding the phone steadily for some time before stopping the sensor data collection. The plot on the right shows that user-2 took as much time as user-1 to pick up the phone, but stopped the sensor data collection immediately after picking up the phone. The x-axis plot on the left figure shows that phone was tilted to the left, then to right, while picking it up. The x-axis plot on the right figure shows the phone was completely tilted to its left while being picked up.

From figure 7 we can easily calculate the position in which the phone was held based on the final resting values of either y and z. The angle theta at which the phone is held is given by $sin^{-1}(y - gravity/9.81)$ or $cos^{-1}(z - gravity/9.81)$. Here, for user 1 the magnitude of y-gravity is 8 and the magnitude of z-gravity is 5.5 at the final holding position. Thus, user

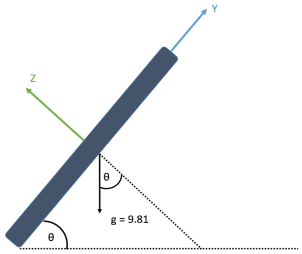

Figure 7: Phone orientation while holding it in reading position. We can calculate the angle at which it is held based on the gravity sensor reading along the y-axis and z-axis.

1 is holding the phone at approximately 55° from the ground. While user 2 is holding the phone at 45°. Based on the gravity sensor we can only tell the orientation of the phone while it is held and during motion. But to understand the motion of the user, whether he was walking or being still, we need to look into the accelerometer sensor data.

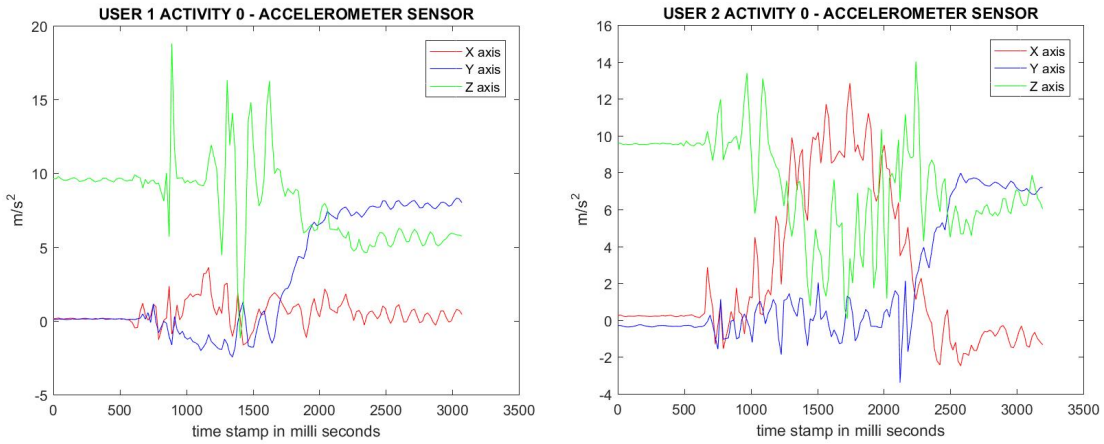

Figure 8: Accelerometer sensor data for activity 0 is plotted against timestamp for two users. The accelerometer sensor reading is similar to the gravity sensor, but it has linear acceleration superimposed on it. The above figure shows the accelerometer sensor data for activity-0 (picking up the phone from the table). The size of the spikes corresponds to how fast the phone was picked up. Since the linear acceleration was the highest along the z-axis, we observe large spikes along the z-axis.

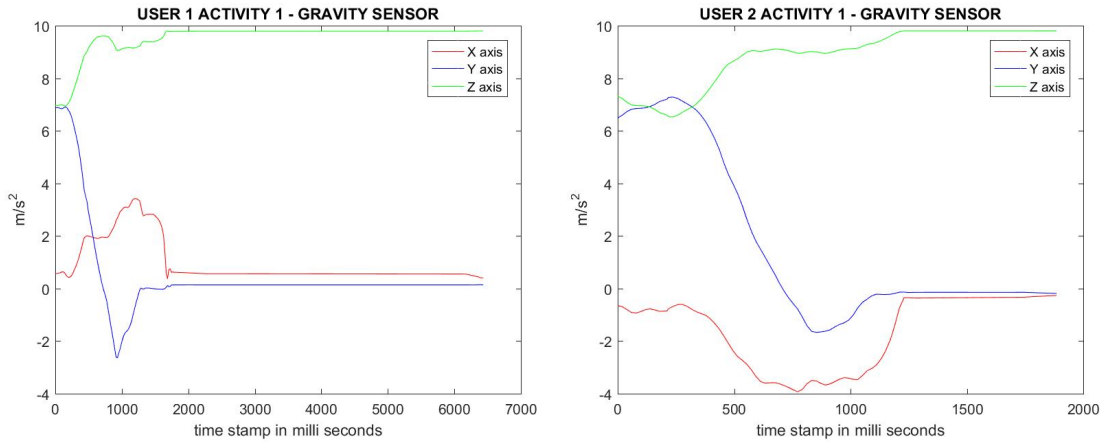

Figure 9: Gravity sensor data for activity-1 (placing the phone on the table) is shown for two users. The negative transition of the gravity sensor along the y-axis and positive transition of the gravity sensor reading along the z-axis indicates that the phone is being placed on the table.

The accelerometer sensor data for activity 0 is shown in figure 8 in which gravity sensor data is displayed with linear acceleration superimposed on it. The overall shape of the x, y and z accelerometer is similar to that of the gravity sensor after the transition has occurred, except for a small noise like linear transition after 2500 ms which indicates that the phone was held relatively steady in hand, and the user was not moving.

Gravity sensor data for activity 1 is shown in figure 9. Here the negative transition y-gravity towards zero and positive transition of z-gravity towards 9.81 indicates that the phone is being placed on the table. Based on x-gravity we can infer that user 1 tilted the phone slightly to the left, while user 2 tilted the phone to the right. The tilt of phone, while being placed on the table, indicates that user 1 probably used his right hand to do so, and user 2 used his left hand.

The accelerometer sensor data for activity 1 is shown in figure 10. The overall shape of x, y and z accelerometer is similar to that of the gravity sensor after the transition has occurred, and we can observe that there is almost no linear acceleration indicating that the phone was kept on a flat, steady surface facing upwards.

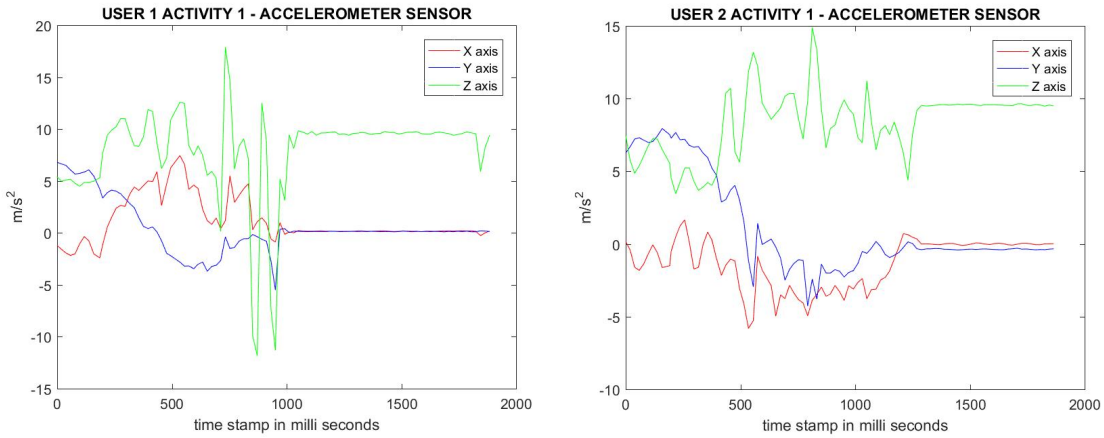

Figure 10: The shape of accelerometer sensor data for activity-1 (placing the phone on the table) for the two users is similar to the gravity sensor data, but it has linear acceleration superimposed on it. We observe large spikes along z-axis, since acceleration is greatest along the z-axis for this activity.

Gravity sensor data for activity-2 is shown in figure 11. The transition at around 1000 ms indicates the instant the user attended the call and moved the phone towards his ear. We can observe from y-gravity that both the users were holding the phone at about the same angle and from the z-gravity data we can infer that the phone was slightly facing downwards. But negative x-gravity indicates that user 1 held the phone in his right hand, while the positive x-gravity indicates that user 2 held the phone in his left hand.

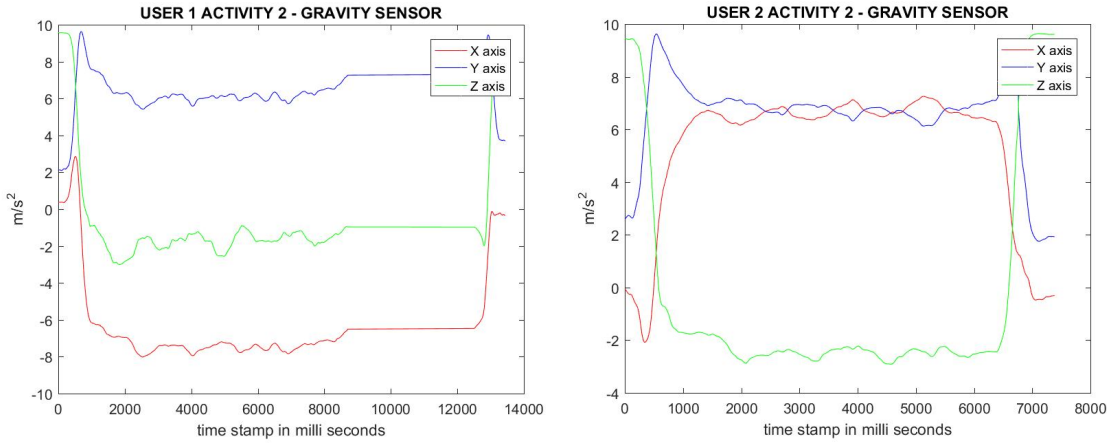

Figure 11: Gravity sensor data for activity-2 (attending a call and then walking). The peaks at around 1000 ms in both plots indicates the instant the call was attended by the participants. The peak at around 13000 ms in the left figure and the peak at 7000 ms in the right figure indicates the instant the call was disconnected by the participants. The gravity sensor data between the instants of attending the call and disconnecting the call shows the sensor data for walking. The negative transition of x-axis gravity in the left figure shows that user-1 held the phone in his right hand. The positive transition of x-axis gravity shows that user-2 held the phone in his left hand.

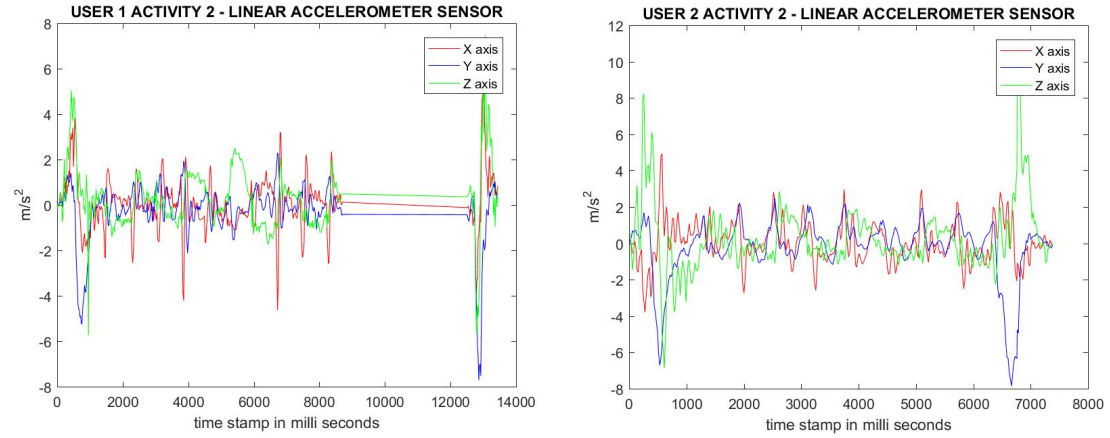

Figure 12: Linear accelerometer sensor data for activity-2 (attending a call and walking) is shown for two participants. The large spikes at 1000 ms and 13000 ms in the left figure shows the instant the call was attended and disconnected respectively. The large spikes at 500 ms and around 7000 ms in the right figure shows the instant the call was attended and disconnected respectively. The small spikes between the instants of attending and disconnecting the call indicates the steps the participant took while walking. So by counting the number of spikes, we can calculate the number of steps the participant took.

To determine whether the user was in motion or sitting during the call can be inferred from the linear accelerometer sensor data for activity 2 as shown in figure 12. For user 1 the spikes between 2000 ms and 8000 ms indicates the number of steps that the user took while on the call. For example, we can infer from the graph that user 1 took 6 steps. Similarly, for user 2 the spikes between 1000 ms and 6000 ms indicates the number of steps that the user took while on call in which case we can infer that user 2 took 8 steps.

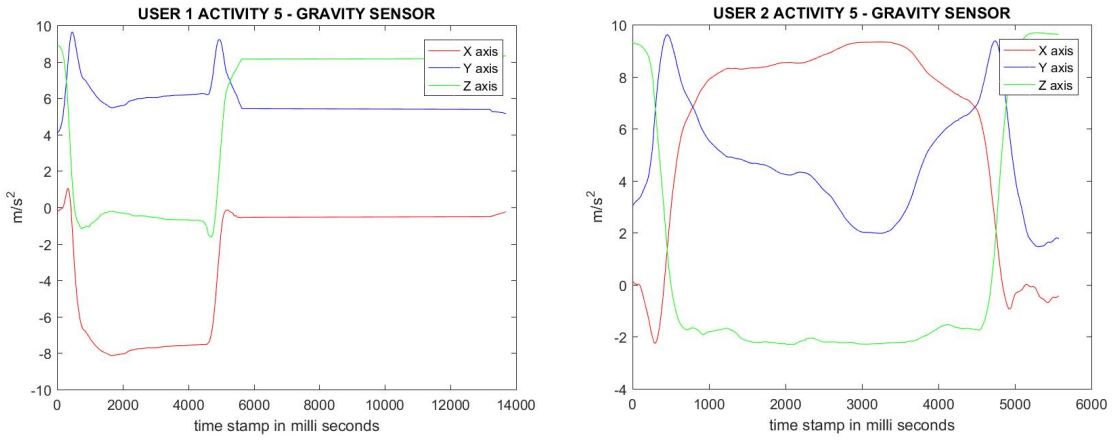

Figure 13: Gravity sensor data for activity-5 (attending a call while sitting). The first spike indicates the instants when the call was attended, and the second spike indicates when the call was disconnected. The negative value of x-gravity in the left figure indicates that user-1 was holding the phone in his left hand, while the positive value of x-gravity in the right figure indicates user-2 was holding the phone in his right hand.

Gravity sensor data for activity 5 is shown in figure 13. Between 0 ms and 6000 ms the data plot looks similar to figure 11. Thus the user attended a call and moved the phone towards his ear. Also based on x-gravity values we can infer that user 1 was holding the phone using his left hand and user 2 was holding the phone using his right hand. To determine if the user was in motion or still during the call, we will use the linear accelerometer data shown in figure 14. The first large spikes around 1000 ms indicates the instant when the users attended the call and moved the phone towards their ear. The second large spikes around 5000 ms indicates the instant when the users moved the phone away from their ear and ended the call. The sensor data between these two spikes is relatively steady around 0 m/s^2 with some noise like pattern. Thus both the users were still during the call.

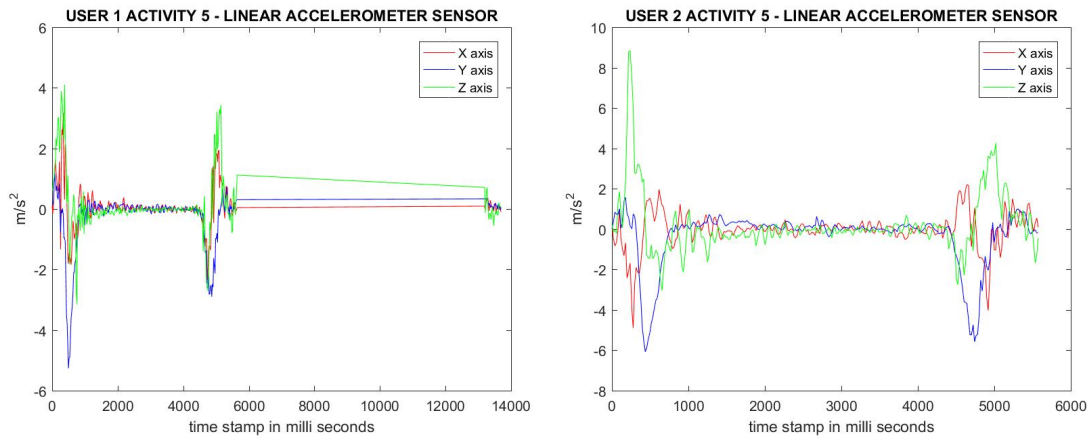

Figure 14: Linear accelerometer sensor data for activity-5 (attending a call while sitting). The first large spike indicates the instants when the call was attended, and the second spike indicates when the call was disconnected. In between the two spikes there is almost no acceleration, and there's only noise indicating that there was no motion during this time.

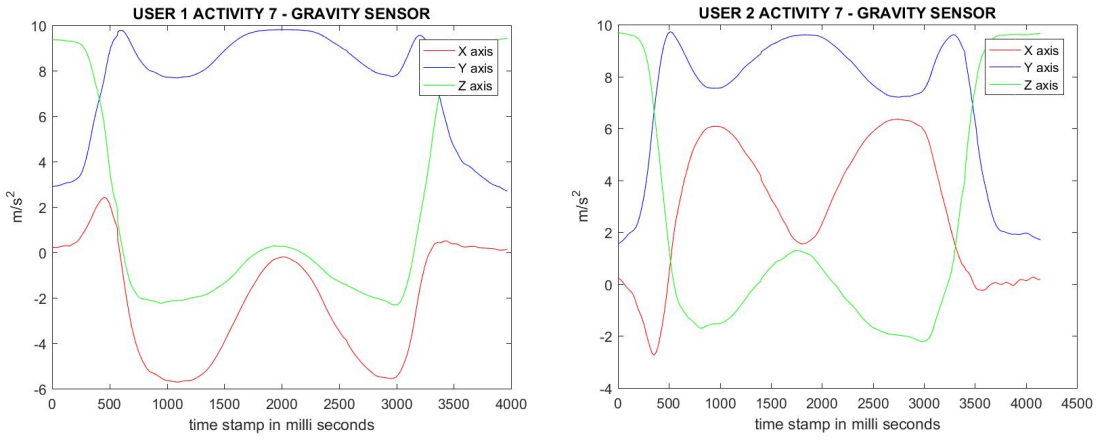

Figure 15: Gravity sensor data for activity-7 (attending a call and sitting down) is similar to figure 11. There is a peak in the left figure and a trough in the right figure at around 2000 ms. This corresponds to the sitting motion while on a call. The negative x-axis gravity in the left figure shows that user-1 was holding the phone in his right hand. The positive x-axis gravity in the right figure shows that user-2 was holding the phone in his left hand.

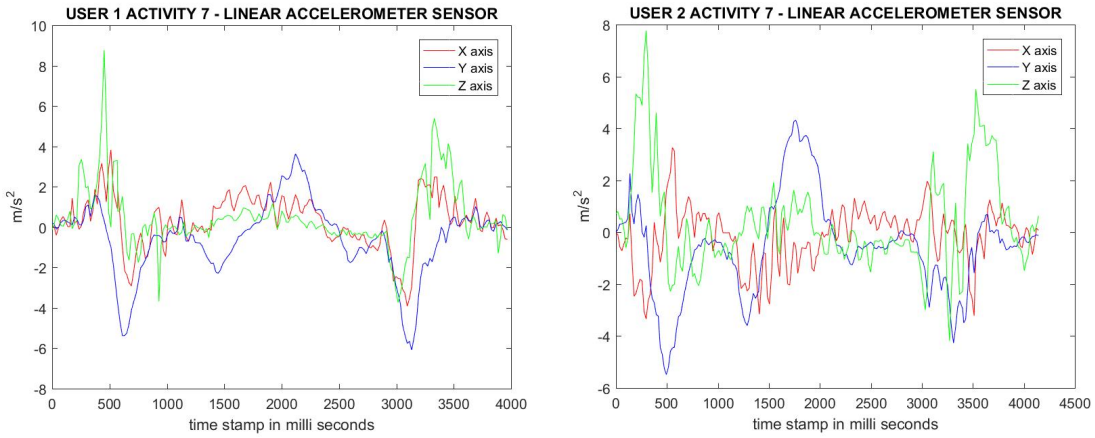

Figure 16: Linear accelerometer sensor data for activity-7 (attending a call and sitting down). The positive spike at around 2000 ms corresponds to the sitting activity performed by the participants. The negative spikes at 500 ms and 3000 ms corresponds to attending the call and disconnecting the call respectively.

Based on the gravity sensor data from figure 15 we can infer that both the users were on call since the gravity sensor data plot is similar to figure 11. From the x-gravity we can infer that user-1 was using his right hand and user-2 was using his left hand to attend the call. At around 2000 ms we can observe a peak and trough in the x-gravity, in the left plot and right plot of figure 15 respectively. This peak and trough corresponds to the sitting activity performed by the participants during the call. There is a peak and a trough for the same sitting motion because the participants were holding the phone in different hands during the call. Holding the phone to the ear using the right hand and then sitting generates a peak, while holding the phone in the left hand generates a trough. The positive spike on the y-axis at 2000 ms according to the linear accelerometer data, as shown in figure 16, corresponds to this sitting activity. The negatives spikes at 500 ms and 3000 ms in figure 16 corresponds to attending the call and disconnecting the call receptively.

The Classifier Design

In this section we discuss the design of our classifier, that is capable of classifying activities of users with an accuracy of 97.9% and sensitivity of 88.9%. The classifier utilizes the data from all of the motion sensors for classification. The classifier was designed using Matlab on a local computer. We study the performance using the k nearest neighbor classification method, which uses dynamic time warping (DTW) as the distance metric. The DTW algorithm is best suited for time series analysis such as the sensor data, which is usually time shifted. DTW is useful is measuring the similarity between two temporal sensor data sequences. To understand how DTW works and how it can be used to find the similarity between two time shifted signals consider figure 17 and figure 18.

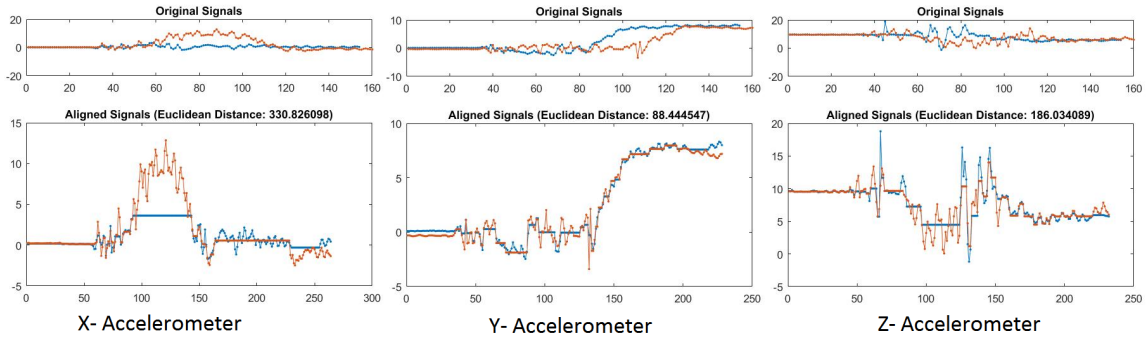

Figure 17: DTW distance of accelerometer sensor data of activity-0 for two users.

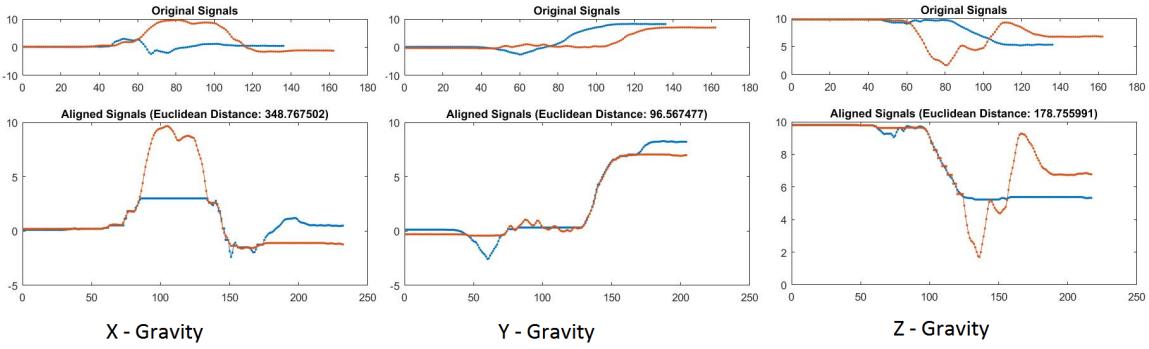

Figure 18: DTW distance of gravity sensor data of activity-0 for two users.

The signals shown in figure 17 and figure 18 are for activity 0, which is lifting the phone off from a table. The x component of the signal depends on how the phone was titled while it was lifted, i.e. whether it was tilted to the left or right. Since different users tilt the phone differently while lifting, the x components of the signal has the maximum DTW distance as compared to y and z components. As the phone is lifted, the sequence of data for y and z components of the signal are very similar for the users. The y component of signals here has the minimum DTW distance compared to the other two components.

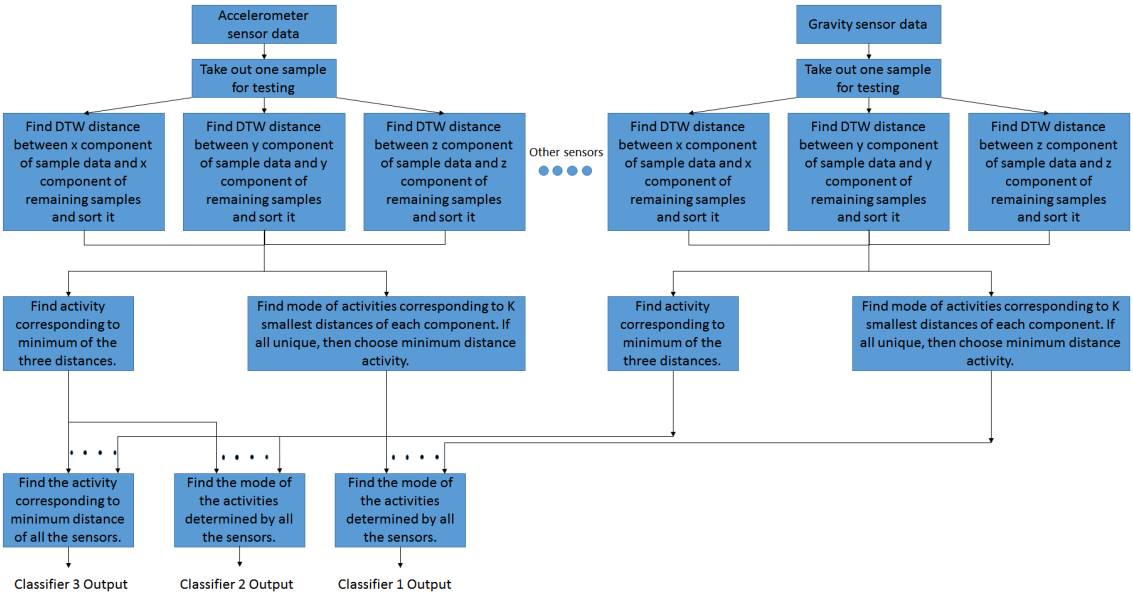

Figure 19: Classifier design that considers the raw sensor data. The motion sensors available on the Nexus 5 device - accelerometer, gravity sensor, gyroscope, and linear accelerometer are used to classify an activity. The classifier uses DTW as the distance metric for a knn based classifier.

Based on these observations, we built a classifier that calculates the DTW distance between the x components of all of the samples of one motion sensor. We then repeated this for the components between the y-axis and the z-axis. From among the three signal components, we can find the component that has the minimum distance, and then match the activity corresponding to that distance in the labeled sensor data. For k nearest neighbors, we sort the distances in ascending order and take the statistical mode of k activities corresponding to k minimum DTW distances for each signal component. We then take the mode of the activities as recognized by the signal components. If the activities recognized are all unique then we consider the activity corresponding to the minimum distance. The

classifier design is shown in figure 19. In the following section we evaluate and compare the performance of the three classifiers we designed.

Results

We used leave-one-out cross validation to evaluate the performance of our classifier. We report and compare the confusion matrix, accuracy, Fscore, precision, sensitivity, and specificity for our classifier design, as shown in figure 19, for different values of K. The definition of these performance metrics is given below.

- Confusion matrix is the error matrix, that visualizes the performance of our classifier design. The columns of the confusion matrix represent the predicted activities, and the rows represent the actual activity.
- Accuracy is the measure of fraction of all the activities that are correctly classified; it is the ratio of the total number of correct classifications to the total number of correct or incorrect classification.
- Precision is the fraction of classified activities that are relevant or true positive classification.
- Sensitivity is a measure of proportion of positives that are correctly classified. Sensitivity is the ratio of the total number of true positive classification to the total number of false negatives and true positives.
- Specificity is a measure of proportion of true negatives that are correctly classified. Sensitivity is the ratio of the total number of true negative classification to the total number of false negatives and true positives.
- Fscore is a measure of the classifier's accuracy, that considers both the precision and sensitivity of the classifier.

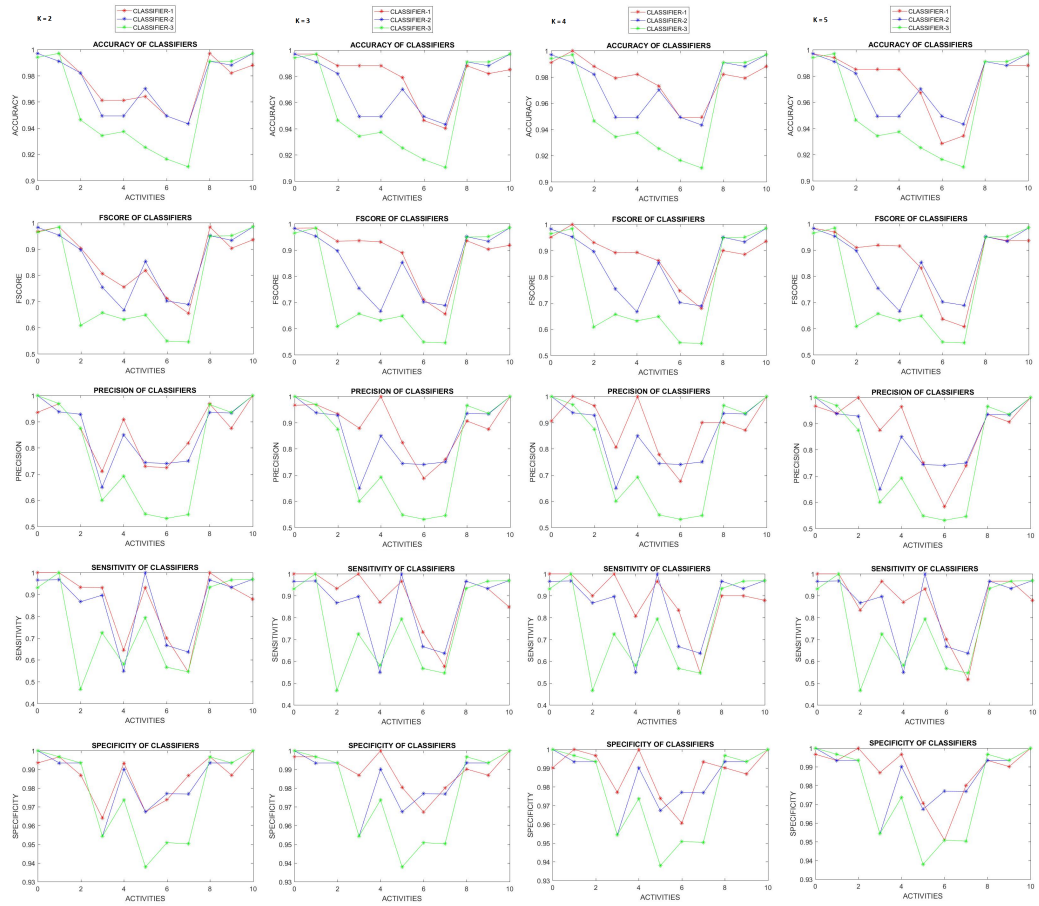

Figure 20: The performance of the three classifiers is compared for k = 2,3,4,5. Classifier-2 and Classifier-3 shows a steady performance for all values of k. Classifier-1 has the best performance with k = 3 and k = 4, and an average accuracy of 97.9 % and sensitivity of 88.9 %. DTW distances is calculated between the testing sensor data and the training sensor data of corresponding motion sensors. The DTW distances are sorted in increasing order for each sensor. Classifier-1 compares the first k DTW distances from all the motion sensors and classifies an activity based on the minimum DTW distance. Classifier-2 classifies based on the mode of the activities corresponding to the first k DTW distances of all of the motion sensors. Classifier-3 takes the mode of activities corresponding to first k DTW distances of each motion sensor, and then takes the mode of the activities detected by each sensor to obtain the classifier output.

Based on the classifier performance metrics defined above, we can observe from figure 20 that classifier-1 has the best performance over the other two classifier designs. Classifier-2 and classifier-3 show a steady performance throughout. This is because their design does not consider the value of K, since these two classifiers only consider the activity corresponding to the minimum DTW distance. We observe that for k = 3 and k = 4, classifier-1 has the best overall performance. Thus taking the mode of activities determined by the motion sensors gives the best overall classification results. For k = 3 and k = 4, the classifier-1 on an average has an accuracy of 97.9%, precision of 89.09%, specificity of 98.85%, sensitivity of 88.9%, and an Fscore of 88.42%. The confusion matrix for Classifier-1 for values of k = 3 and k = 4 and is shown in figure 21.

k = 3
Classifier-1 Confusion Matrix

Actual Activity \ Predicted Activity	0	1	2	3	4	5	6	7	8	9	10
0	29	0	0	0	0	0	0	0	0	0	0
1	0	31	0	0	0	0	0	0	0	0	0
2	0	0	28	0	0	1	0	1	0	0	0
3	0	0	0	29	0	0	0	0	0	0	0
4	0	0	0	4	27	0	0	0	0	0	0
5	0	1	0	0	0	28	0	0	0	0	0
6	0	0	0	0	0	3	22	5	0	0	0
7	0	0	2	0	0	2	10	19	0	0	0
8	1	0	0	0	0	0	0	0	29	0	0
9	0	0	0	0	0	0	0	0	2	28	0
10	0	0	0	0	0	0	0	0	1	4	28

k = 4
Classifier-1 Confusion Matrix

Actual Activity \ Predicted Activity	0	1	2	3	4	5	6	7	8	9	10
0	29	0	0	0	0	0	0	0	0	0	0
1	0	31	0	0	0	0	0	0	0	0	0
2	1	0	27	0	0	1	0	1	0	0	0
3	0	0	0	29	0	0	0	0	0	0	0
4	0	0	0	6	25	0	0	0	0	0	0
5	0	0	1	0	0	28	0	0	0	0	0
6	0	0	0	0	0	4	25	1	0	0	0
7	0	0	0	0	0	3	12	18	0	0	0
8	2	0	0	1	0	0	0	0	27	0	0
9	0	0	0	0	0	0	0	0	3	27	0
10	0	0	0	0	0	0	0	0	0	4	29

Figure 21: The confusion matrix for Classifier-1 for k values indicating the best performance.

Chapter 4

Discussion

In this section, we first discuss the work that we pursued in the beginning towards exploiting the system-wide information. We then discuss the implications of our work in inferring the lifestyle of a user using results from motion sensor and broadcast receiver API.

Investigating system-wide information files

Android OS is based on the Linux kernel, and it inherits the /proc virtual filesystem from Linux. The /proc filesystem is also referred to as process information pseudo-file system, which contains runtime system information such as system memory, hardware configuration, etc. We investigated the /proc/locks file which displays the files that are currently locked by the kernel. The lock file contents are as shown in figure 22.

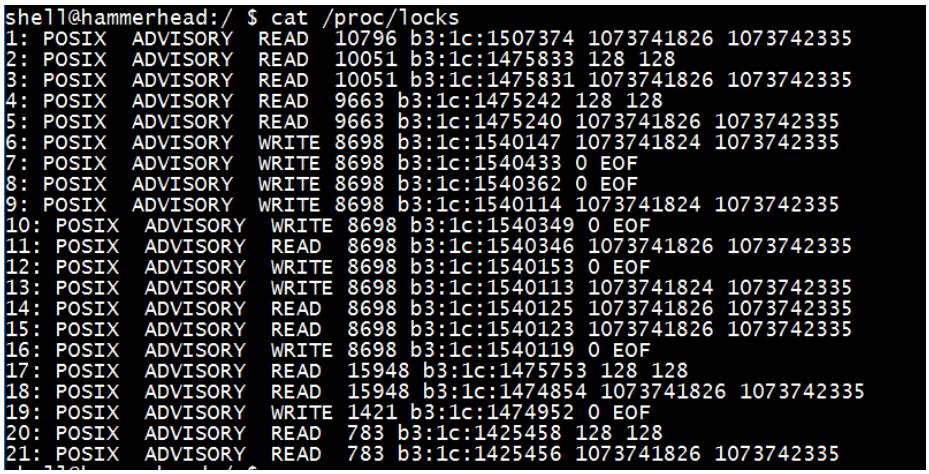

Figure 22: Screen shot of /proc/locks file

Column 1: Each lock on a file has a unique number. 1, 2 etc. Column 2: POSIX represents the new POSIX locks from the lockf system call. Column 3: Advisory means that the lock on that file does not prevent other processes from accessing the data, but it prevents

attempts to lock onto that file. Column 4: Indicates whether the lock has Read or Write access to the file. Column 5: indicates the process ID that holds the lock. Column 6: This is the ID of the locked file. The format is as follows: Major Device:Minor Device:Outnumber. The inum or inode number is an integer associated with a file. Whenever a new file is created, a unique integer number is generated in sequence and associated with the file. This number is a pointer to the inode structure which contains the meta data of the file. An inode number is used by the operating system to refer to the file during all the processes involving the file. Column 7 and 8: indicates the start and end of the files locked region. The following commands were used in the Android debug shell to log the /proc/locks and also the output from the "top" command was logged so that we can get the process name that corresponds to the process ID from column 5 in /proc/locks.

watch -n 1 'adb shell cat proc/locks — sed -e "s/ ^ / $(date -R) /" ⟩⟩ locks.log'

watch -n 1 'adb shell top — sed -e "s/^/$(date -R) /" ⟩⟩ top.log'

```
1: POSIX  ADVISORY  WRITE 7254 b3:1c:1540147 1073741824 1073742335
2: POSIX  ADVISORY  WRITE 7254 b3:1c:1540362 0 EOF
3: POSIX  ADVISORY  WRITE 7254 b3:1c:1540114 1073741825 1073741825
4: POSIX  ADVISORY  WRITE 7254 b3:1c:1540349 0 EOF
5: POSIX  ADVISORY  READ  7254 b3:1c:1540346 1073741826 1073742335
6: POSIX  ADVISORY  WRITE 7254 b3:1c:1540153 0 EOF
7: POSIX  ADVISORY  WRITE 7254 b3:1c:1540113 1073741824 1073742335
8: POSIX  ADVISORY  READ  7254 b3:1c:1540114 1073741826 1073742335
9: POSIX  ADVISORY  READ  7254 b3:1c:1540125 1073741826 1073742335
10: POSIX  ADVISORY  READ  7254 b3:1c:1540123 1073741826 1073742335
11: POSIX  ADVISORY  WRITE 7254 b3:1c:1540119 0 EOF
1: POSIX  ADVISORY  WRITE 7254 b3:1c:1540147 1073741824 1073742335
2: POSIX  ADVISORY  WRITE 7254 b3:1c:1540362 0 EOF
3: POSIX  ADVISORY  WRITE 7254 b3:1c:1540114 1073741825 1073741825
4: POSIX  ADVISORY  WRITE 7254 b3:1c:1540349 0 EOF
5: POSIX  ADVISORY  READ  7254 b3:1c:1540346 1073741826 1073742335
6: POSIX  ADVISORY  WRITE 7254 b3:1c:1540153 0 EOF
7: POSIX  ADVISORY  WRITE 7254 b3:1c:1540113 1073741824 1073742335
8: POSIX  ADVISORY  READ  7254 b3:1c:1540114 1073741826 1073742335
9: POSIX  ADVISORY  READ  7254 b3:1c:1540125 1073741826 1073742335
10: POSIX  ADVISORY  READ  7254 b3:1c:1540123 1073741826 1073742335
11: POSIX  ADVISORY  WRITE 7254 b3:1c:1540119 0 EOF
```

Figure 23: Screen shot of locks.log file when Chrome browser was first started

A part of locks.log is shown in figure 23, which was obtained when the Chrome browser was started. Logs shown in figure 24 were obtained after closing and restarting the Chrome browser. The process ID 7254 and 8317 corresponds to the Chrome browser. The blue box in figure 24 shows some of the write locked files and the red box shows the read locked files. After the Chrome browser was re-spawned with a new PID, we observe from the inode

```
3: POSIX  ADVISORY  WRITE 8317 b3:1c:1540349 0 EOF
4: POSIX  ADVISORY  READ  8317 b3:1c:1540346 1073741826 1073742335
5: POSIX  ADVISORY  WRITE 8317 b3:1c:1540153 0 EOF
6: POSIX  ADVISORY  WRITE 8317 b3:1c:1540113 1073741824 1073742335
7: POSIX  ADVISORY  READ  8317 b3:1c:1540114 1073741826 1073742335
8: POSIX  ADVISORY  READ  8317 b3:1c:1540125 1073741826 1073742335
9: POSIX  ADVISORY  READ  8317 b3:1c:1540123 1073741826 1073742335
10: POSIX  ADVISORY  WRITE 8317 b3:1c:1540119 0 EOF
1: POSIX  ADVISORY  WRITE 8317 b3:1c:1540349 0 EOF
2: POSIX  ADVISORY  READ  8317 b3:1c:1540346 1073741826 1073742335
3: POSIX  ADVISORY  WRITE 8317 b3:1c:1540153 0 EOF
4: POSIX  ADVISORY  WRITE 8317 b3:1c:1540113 1073741824 1073742335
5: POSIX  ADVISORY  READ  8317 b3:1c:1540114 1073741826 1073742335
6: POSIX  ADVISORY  READ  8317 b3:1c:1540125 1073741826 1073742335
7: POSIX  ADVISORY  READ  8317 b3:1c:1540123 1073741826 1073742335
8: POSIX  ADVISORY  WRITE 8317 b3:1c:1540119 0 EOF
1: POSIX  ADVISORY  WRITE 8317 b3:1c:1540362 0 EOF
2: POSIX  ADVISORY  READ  8317 b3:1c:1540147 1073741826 1073742335
3: POSIX  ADVISORY  WRITE 8317 b3:1c:1540114 1073741825 1073741825
4: POSIX  ADVISORY  WRITE 8317 b3:1c:1540349 0 EOF
5: POSIX  ADVISORY  READ  8317 b3:1c:1540346 1073741826 1073742335
6: POSIX  ADVISORY  WRITE 8317 b3:1c:1540153 0 EOF
7: POSIX  ADVISORY  WRITE 8317 b3:1c:1540113 1073741824 1073742335
8: POSIX  ADVISORY  READ  8317 b3:1c:1540114 1073741826 1073742335
9: POSIX  ADVISORY  READ  8317 b3:1c:1540125 1073741826 1073742335
10: POSIX  ADVISORY  READ  8317 b3:1c:1540123 1073741826 1073742335
11: POSIX  ADVISORY  WRITE 8317 b3:1c:1540119 0 EOF
```

Figure 24: Screen shot of locks.log file after restarting the Chrome browser

number that it obtains a lock on same files. It should be the same file because no two files can have the same inode number and the inode numbers are generated in sequence. Based on the files accessed by a process we can infer the behavior of an application and what operation it is performing at any given time. Also, if the files that are accessed by the application are from its own internal directory, then those files would be private to that application alone and obtaining the locks on files would be redundant. Thus most of these files must be residing in external storage. If any of these files are residing in external storage and are classes that the application loads using DexClassLoader API, then these classes could be modified or injected with malicious code by other applications since files in external storage are public to all the applications [19].

We only have the knowledge of an inode number of the files that are being accessed. Research of this section to determine where these files are actually residing, whether internal or external storage, is left for future work.

Inferring lifestyle of a user

By combining broadcast events with results from activity recognition, we can tell how often the user charges his smartphone and what the user does during that time. We can infer if the phone was kept tilted on the table during charging, and if the user constantly unlocked his phone during this time. If the smartphone is on charge and undisturbed for long hours we can infer that the user is sleeping. Based on the intervals of timestamps, we can infer if any of these events or users activities are routine events. During the study participants expressed concern that Android applications can potentially invade their privacy, if any application is using sensors to infer their daily activities. They were surprised to know that Android applications do not need any special permission to utilize the smartphone sensors and also the applications do not disclose their use of sensors during installation. Thus sensors on smartphones, that do not require any special permission, can be used as a side channel for inferring the daily activities of the smartphone users.

Chapter 5

Conclusions

We have shown here that by having permission to read external storage we were able to obtain information about the Google account used by the user and the user's location. Similarly, while investigating the default applications on Moto G, we found that the default email application stores email conversations in the external storage. The email conversations can be used to extract contact information and other personal information about the users. Since the exploited applications were the default applications on these devices, all of the users owning these devices are prone to this side channel attack.

Previous research on activity recognition does not tell much about users lifestyle or surroundings other than simple activities like walking, running, sitting or standing. We show that by using broadcast receivers we can infer events such as whether Airplane mode is on or off, when a user successfully unlocks the screen and interacts with the smartphone, when the smartphone is going to sleep due to inactivity, when the smartphone battery is low, when the user is plugging the smartphone in for charging and for how long by listening to the plug out event and when the user plugs in headset. We can use the timestamp of these broadcast events to look into the database where we collected the detected physical activities, to infer if the user was walking, or being still during that time. Thus, by combining results of broadcast receivers with activity recognition we can infer the lifestyle of users.

We then present a classifier that utilizes the sensors data from all of the motion sensors available on the Nexus 5 smartphone. The classifier was evaluated using leave-one-out cross validation, and we were able to achieve a 98% classification accuracy on an average, thus we show that by having a labeled sensor data, we can build sensors models for recognizing complex activities performed by other users.

References

[1] Android Source, https://source.android.com

[2] Android Developer, http://developer.android.com/index.html

[3] Felt, Adrienne Porter, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. "Android permissions: User attention, comprehension, and behavior." In Proceedings of the Eighth Symposium on Usable Privacy and Security, p. 3. ACM, 2012.

[4] Kelley, Patrick Gage, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. "A conundrum of permissions: installing applications on an Android smartphone." In Financial Cryptography and Data Security, pp. 68-79. Springer Berlin Heidelberg, 2012.

[5] Xu, Zhi, Kun Bai, and Sencun Zhu. "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors." In Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks, pp. 113-124. ACM, 2012.

[6] Cai, Liang, and Hao Chen. "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion." HotSec 11 (2011): 9-9.

[7] Simon, Laurent, and Ross Anderson. "PIN skimmer: inferring PINs through the camera and microphone." In Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices, pp. 67-78. ACM, 2013.

[8] Kwapisz, Jennifer R., Gary M. Weiss, and Samuel A. Moore. "Activity recognition using cell phone accelerometers." ACM SigKDD Explorations Newsletter 12, no. 2 (2011): 74-82.

[9] Shoaib, Mohammed, Hans Scholten, and Paul JM Havinga. "Towards physical activity recognition using smartphone sensors." In Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC), pp. 80-87. IEEE, 2013.

[10] Owusu, Emmanuel, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. "ACCessory: password inference using accelerometers on smartphones." In Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications, p. 9. ACM, 2012.

[11] Jana, Suman, and Vitaly Shmatikov. "Memento: Learning secrets from process footprints." In Security and Privacy (SP), 2012 IEEE Symposium on, pp. 143-157. IEEE, 2012.

[12] Incel, Ozlem Durmaz, Mustafa Kose, and Cem Ersoy. "A review and taxonomy of activity recognition on mobile phones." BioNanoScience 3, no. 2 (2013): 145-171.

[13] Wu, Wanmin, Sanjoy Dasgupta, Ernesto E. Ramirez, Carlyn Peterson, and Gregory J. Norman. "Classification accuracies of physical activities using smartphone motion sensors." Journal of medical Internet research 14, no. 5 (2012): e130.

[14] Al-Haiqi, Ahmed, Mahamod Ismail, and Rosdiadee Nordin. "The eye as a new side channel threat on smartphones." In Research and Development (SCOReD), 2013 IEEE Student Conference on, pp. 475-479. IEEE, 2013.

[15] Kenworthy, Gary, and Pankaj Rohatgi. "Mobile Device Security: The case for side channel resistance." (2012).

[16] Kenworthy, Gary, and Pankaj Rohatgi. "Mobile Device Security: The case for side channel resistance." (2012).

[17] Enck, William, Machigar Ongtang, and Patrick McDaniel. "Understanding Android security." IEEE security & privacy 1 (2009): 50-57.

[18] Miller, Charlie. "Mobile attacks and defense." Security & Privacy, IEEE 9, no. 4 (2011): 68-70.

[19] Fernandes, Earlence, Bruno Crispo, and Marco Conti. "Fm 99.9, radio virus: Exploiting fm radio broadcasts for malware deployment." Information Forensics and Security, IEEE Transactions on 8, no. 6 (2013): 1027-1037.

[20] Anjum, Ashiq, and Muhammad Usman Ilyas. "Activity recognition using smartphone sensors." In Consumer Communications and Networking Conference (CCNC), 2013 IEEE, pp. 914-919. IEEE, 2013.

[21] Viet, Vo Quang, Gueesang Lee, and Deokjai Choi. "Fall detection based on movement and smartphone technology." In Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2012 IEEE RIVF International Conference on, pp. 1-4. IEEE, 2012.

[22] https://source.android.com/devices/sensors/sensor-types.html

[23] Lin, Jialiu, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing." In Proceedings of the 2012 ACM Conference on Ubiquitous Computing, pp. 501-510. ACM, 2012.

[24] Zhang, Nan, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng Wang. "Leave me alone: App-level protection against runtime information gathering on Android." In Security and Privacy (SP), 2015 IEEE Symposium on, pp. 915-930. IEEE, 2015.

[25] Markmann, Tobias, David Gessner, and Dirk Westhoff. "Quantdroid: Quantitative approach towards mitigating privilege escalation on Android." In Communications (ICC), 2013 IEEE International Conference on, pp. 2144-2149. IEEE, 2013.

[26] Zhou, YongBin, and DengGuo Feng. "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing." IACR Cryptology ePrint Archive 2005 (2005): 388.

[27] Sarwar, Golam, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. "On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices." In SECRYPT, pp. 461-468. 2013.

[28] Chen, Qi Alfred, Zhiyun Qian, and Z. Morley Mao. "Peeking into your app without actually seeing it: Ui state inference and novel Android attacks." In 23rd USENIX Security Symposium (USENIX Security 14), pp. 1037-1052. 2014.

[29] Hutter, Michael, and Jrn-Marc Schmidt. "The temperature side channel and heating fault attacks." In Smart Card Research and Advanced Applications, pp. 219-235. Springer International Publishing, 2013.

[30] Genkin, Daniel, Adi Shamir, and Eran Tromer. "RSA key extraction via low-bandwidth acoustic cryptanalysis." In Advances in CryptologyCRYPTO 2014, pp. 444-461. Springer Berlin Heidelberg, 2014.

[31] Wei, Michael, Benedikt Heinz, and Frederic Stumpf. "A cache timing attack on AES in virtualization environments." In Financial Cryptography and Data Security, pp. 314-328. Springer Berlin Heidelberg, 2012.

[32] Cai, Liang, and Hao Chen. "On the practicality of motion based keystroke inference attack." Springer Berlin Heidelberg, 2012.

[33] Orthacker, Clemens, Peter Teufl, Stefan Kraxberger, Gnther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber. "Android security permissionscan we trust them?." In Security and Privacy in Mobile Information and Communication Systems, pp. 40-51. Springer Berlin Heidelberg, 2011.

[34] Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android permissions demystified." In Proceedings of the 18th ACM conference on Computer and communications security, pp. 627-638. ACM, 2011.

[35] Ferreira, Denzil, Vassilis Kostakos, Alastair R. Beresford, Janne Lindqvist, and Anind K. Dey. "Securacy: an empirical investigation of Android applications' network usage, privacy and security." In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, p. 11. ACM, 2015.

[36] Lei, Lingguang, Yuewu Wang, Jian Zhou, Daren Zha, and Zhongwen Zhang. "A threat to mobile cyber-physical systems: Sensor-based privacy theft attacks on Android smartphones." In Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on, pp. 126-133. IEEE, 2013.

[37] Xu, Zhi, and Sencun Zhu. "SemaDroid: A Privacy-Aware Sensor Management Framework for Smartphones." In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 61-72. ACM, 2015.

[38] Wang, Qinglong, Amir Yahyavi, Bettina Kemme, and Wenbo He. "I know what you did on your smartphone: Inferring app usage over encrypted data traffic." In Communications and Network Security (CNS), 2015 IEEE Conference on, pp. 433-441. IEEE, 2015.

[39] Ohmura, Ren, and Wataru Takasaki. "Response time improvement in accelerometer-based activity recognition by activity change detection." In Proceedings of the 13th international conference on Ubiquitous computing, pp. 589-590. ACM, 2011.

[40] http://www.appbrain.com/stats/number-of-android-apps

[41] bin Abdullah, Mohd Fikri Azli, Ali Fahmi Perwira Negara, Md Shohel Sayeed, Deok-Jai Choi, and Kalaiarasi Sonai Muthu. "Classification algorithms in human activity recognition using smartphones." International Journal of Computer and Information Engineering 6 (2012): 77-84.

[42] Guiry, John J., Pepijn van de Ven, John Nelson, Lisanne Warmerdam, and Heleen Riper. "Activity recognition with smartphone support." Medical engineering & physics 36, no. 6 (2014): 670-675.

[43] Wu, Wanmin, Sanjoy Dasgupta, Ernesto E. Ramirez, Carlyn Peterson, and Gregory J. Norman. "Classification accuracies of physical activities using smartphone motion sensors." Journal of medical Internet research 14, no. 5 (2012): e130.

[44] Gao, Lei, A. K. Bourke, and John Nelson. "Evaluation of accelerometer based multi-sensor versus single-sensor activity recognition systems." Medical engineering & physics 36, no. 6 (2014): 779-785.

[45] Krishnan, Narayanan C., and Diane J. Cook. "Activity recognition on streaming sensor data." Pervasive and mobile computing 10 (2014): 138-154.

[46] Al-Haiqi, Ahmed, Mahamod Ismail, and Rosdiadee Nordin. "Keystrokes Inference Attack on Android: A Comparative Evaluation of Sensors and Their Fusion." Journal of ICT Research and Applications 7, no. 2 (2013): 117-136.

[47] He, Yi, and Ye Li. "Physical activity recognition utilizing the built-in kinematic sensors of a smartphone." International Journal of Distributed Sensor Networks 2013 (2013).

[48] Al-Haiqi, Ahmed, Mahamod Ismail, and Rosdiadee Nordin. "A new sensors-based covert channel on android." The Scientific World Journal 2014 (2014).

[49] Zhu, Jiang, Pang Wu, Xiao Wang, and Juyong Zhang. "Sensec: Mobile security through passive sensing." In Computing, Networking and Communications (ICNC), 2013 International Conference on, pp. 1128-1133. IEEE, 2013.

[50] Su, Xing, Hanghang Tong, and Ping Ji. "Activity recognition with smartphone sensors." Tsinghua Science and Technology 19, no. 3 (2014): 235-249.

[51] Subramanian, Venkatachalam, Selcuk Uluagac, Hasan Cam, and Raheem Beyah. "Examining the characteristics and implications of sensor side channels." In Communications (ICC), 2013 IEEE International Conference on, pp. 2205-2210. IEEE, 2013.