Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Monitoring smartphone users' security behaviour

Author:
Rini Banerjee

Supervisor:
Dr Soteris Demetriou

Written as part of the Undergraduate Research Opportunities Program (UROP) at Imperial College London.

August 2019

Acknowledgments

I would like to thank my UROP supervisor, Dr Soteris Demetriou, for providing me with this opportunity and taking the time to guide me through this research placement. I am also very grateful to the Department of Computing at Imperial College London for providing me with the resources that allowed me to do this placement.

Contents

Chapter 1

Introduction

It is estimated that nearly three-quarters of the worlds adult population now owns a smartphone [16]. Although smartphone technology has made the world more connected than ever before, it has also led to increased cybersecurity threats from adversaries who are able to exploit the vulnerabilities of these systems. Despite smartphone system developers' best efforts to keep their products equipped with security features, a lot of these features rely solely on the user's decisions. This in itself poses a serious danger to the security of these devices (and, ultimately, their users), since previous research suggests that users typically make minimal attempts to protect smartphone security [17]. It is, therefore, critical to understand human behaviour when it comes to keeping smartphones secure.

Previous research conducted in collaboration with the University of Illinois at Urbana-Champaign suggests that there is a correlation between certain smartphone security behaviours and mental health issues [18]. The main purpose of this research was to develop a standardised scale for measuring users' **smartphone** security behaviours, much like the well-established computer Security Behaviour Intentions Scale (SeBIS) [19], developed by Egelman et al. This research led to the creation of the **Smartphone Security Behavior Scale (SSBS)**, the finalised version of which is shown on the next page in Table 1.1.

My task was to create an Android application that could be used in the real world to track the security behaviours described in the SSBS. Using this app for field studies could improve the accuracy of the correlation between smartphone security behaviours and mental health, since field studies typically provide more accurate, real-world data points than user studies. The latter tends to suffer from issues, such as the potentially large difference between self-reported behaviours and actual behaviours, and so conducting a field study will provide more objective evidence to strengthen the research claims.

I was provided with a Google Pixel 2 XL, running Android 8.1.0 ('Oreo') and API 27, to help me work on this app during my research placement.

I reset my Advertising ID on my smartphone.
I hide device in my smartphone's Bluetooth settings.
I change my passcode/PIN for my smartphone's screen lock at a regular basis.
I manually cover my smartphone's screen when using it in the public area (e.g., bus or subway).
I use an adblocker on my smartphone.
I use an anti-virus app.
I use a Virtual Private Network (VPN) app while connected to a public network.
I turn off WiFi on my smartphone when not actively using it.
I care about the source of the app when performing financial and/or shopping tasks on that app.
When downloading an app, I check that the app is from the official/expected source.
Before downloading a smartphone app, I ensure the download is from official application stores (e.g. Apple App Store, Google Play, Amazon Appstore).
I verify the recipient/sender before sharing text messages or other information using smartphone apps.
I delete any online communications (i.e., texts, emails, social media posts) that look suspicious.
I pay attention to the pop-ups on my smartphone when connecting it to another device (e.g. laptop, desktop).

Table 1.1: Finalised version of SSBS

1.1 Applications of this research

1.1.1 Workplace security

Whether organisations provide their employees with 'work phones', or people use their personal smartphones at work, smartphones have become embedded into almost every workplace. The ever-increasing ubiquity of smartphones comes at a price, however, and organisations are particularly susceptible to cybersecurity threats - it is estimated that worldwide cybercrime costs the global economy $600bn a year [21].

Researching the correlation between users' smartphone security behaviours and their mental health conditions could improve workplace security, since being aware of employees' mental health conditions could help employers to predict which employees are inclined to use the security features on their smartphones in ways that breach their company's security protocol. It is hoped that this research will not only save organisations a significant amount of time, money and resources, but will also improve smartphone security in such workplaces, thereby improving the efficiency and security of these workplaces as a whole.

1.1.2 Healthcare

As technology continues to advance, its role in healthcare is becoming increasingly apparent. There were 40,596 apps in the Health category of the Google Play Store in the second quarter of 2019, and this number looks likely to increase over the next few years [23]. Clinicians who are interested in using health apps for the treatment of their patients may use this app for determining what kind of mental health condition their patient might be suffering from, depending on which security behaviours they exhibit.

1.1.3 Education

Technology is also proving to be an extremely useful tool when it comes to education. Not only do most children and teachers now have a smartphone of their own, but tablets are also being used extensively in classrooms across the world - in 2017, over 1.2 million schoolchildren were using iPads in school [22]. Unfortunately, having young children fully embrace mobile devices as part of their daily lives can come with big security risks, as they are particularly susceptible to cyberthreats (e.g., cyberbullying, stalking). This app can be used to assess the level of vulnerability of each child and teacher, and to improve the safety of school environments.

Chapter 2

Background

2.1 Android Components

Android applications have a very specific structure, and although my project was done using Java, writing the code for the application was very different to writing a Java program. For one thing, I came to realise that Android applications do not contain a single entry point for execution (like the `main()` function in a Java program, for example). Instead, developers are expected to design their applications in terms of components. The following section is an overview of the components I used in my application and the purpose of each component in my work.

2.1.1 Activities

In plain terms, an Android Activity is a single, focussed thing that the user can do [2]. Activities define the app's user interface (UI), and typically, there is one activity per 'screen' of the app [20]. Activities are implemented by extending the inbuilt Activity class, and every app has a MainActivity, which is an Activity that relates to the first screen the user sees when launching the app.

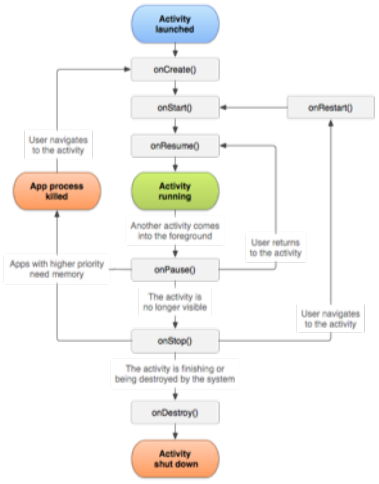

Figure 2.1: A flow diagram showing how Android Activities work.

2.1.2 Broadcast Receivers

To understand Broadcast Receivers, we must first look into Android Broadcasts, which are messages that are sent from the Android system and other Android apps whenever an event of interest occurs. These relate to the *publish-subscribe* design pattern [4] (a form of asynchronous service-to-service communication, where any message published to a topic is immediately received by all subscribers to the topic [1]). Apps can register to receive particular broadcasts so that system changes can be detected in real time.

In order to respond to broadcast messages from the system within our app, we need Broadcast Receivers. Broadcast Receivers are, essentially, mailboxes for broadcast messages [3], so whenever Broadcasts relating to a particular system event are sent to some implicit destination, Broadcast Receivers for that event will subscribe to that destination. One very useful property of Broadcast Receivers is that they continue to receive Broadcasts even when the app they have been registered in is not running. Broadcast Receivers are implemented by extending the `BroadcastReceiver` class.

For a complete list of Android system broadcast actions (as of API 28), please see Figures 2.2 and 2.3 on the following two pages.

2.1.3 Handlers

A Handler is an Android component which allows communication between a background thread and the main UI thread [24]. A Handler takes a Java Runnable object, and allows the same task to be scheduled or repeated at a given time interval. Handlers are particularly useful when there are no inbuilt Broadcast Receivers to listen for the variables we want to track, as they can poll the main UI thread at regular time intervals to check whether a variable is changing. The concurrent nature of Handlers makes them ideal for background processing, since potentially slow running operations in the Android app can be performed asynchronously; this improves the overall user experience[25].

2.1.4 Listeners

An Event Listener is an interface in the Android `View` class that is used to listen for changes the user had invoked in the UI (for example, an `onClickListener` could be used to detect whenever the user clicks on a specific button) [13]. Listeners contain a single call-back method, and the methods of a Listener are called when the user interacts with the `View` object that this particular Listener is registered to (where `View` is the Android class that represents the building block for UI components and is responsible for event handling [11]).

Figure 2.2: List of Android Broadcasts (i)

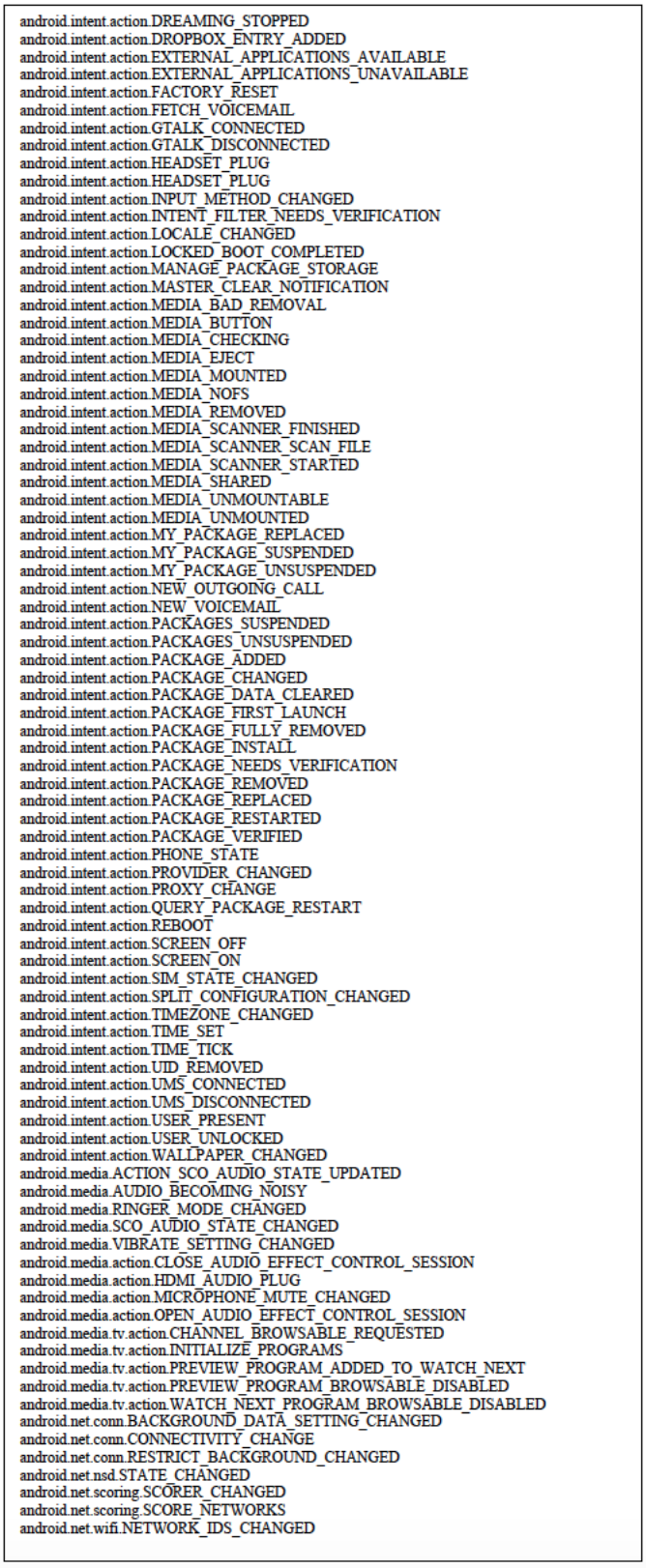

Figure 2.3: List of Android Broadcasts (ii)

2.2 Interaction between components

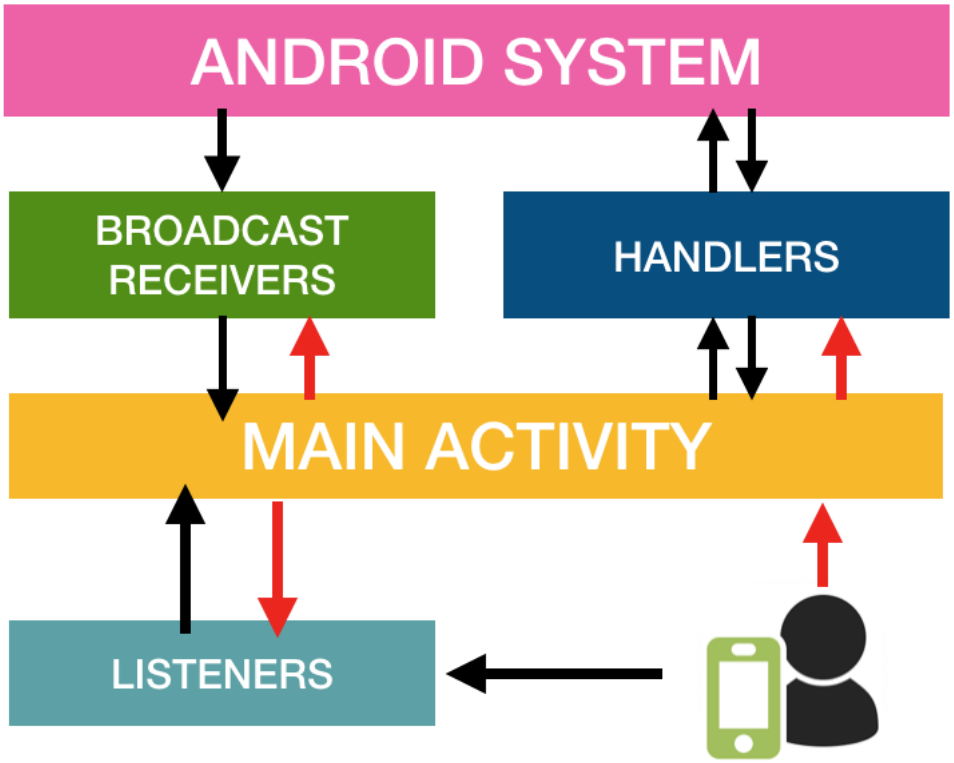

Figure 2.4: System diagram showing Android components interacting with each other

The system diagram above shows how all of the aforementioned Android components interact with each other in the app.

Broadcast Receivers listen for messages from the Android system to inform them of any changes to the variable being tracked, and also send messages to the Main Activity, specifying which tracking behaviours have changed so that this information can be recorded.

Handlers communicate with the Main Activity in a slightly more complicated way than Broadcast Receivers. Although Handlers do not receive messages from the Android system automatically like Broadcast Receivers, the Main Activity can save the last known state of the variable being tracked and then pass it into the Handler (which takes a Java Runnable object and creates a new thread). The Handler can check whether this variable has changed by polling the Android system at regular time intervals and comparing the new value with the stored one. If anything has changed, the Handler simply sends a message back to the Main Activity so that this information can be recorded. There is one Handler in my app, which deals with all of the Handler-based tracking, and it polls the system ***every minute*** after it has been started.

```
private Handler handler;
private Runnable runnable;
private static final int ONE_MINUTE_IN_MILLISECONDS = 60 * 1000;

public void sampleHandlerActivity() {
    handler = new Handler();
    runnable = new Runnable() {
        @Override
        public void run() {
            /*
            ...
            Run your method here
            ...
            */

            /* Repeat every 1 minute */
            handler.postDelayed(runnable, ONE_MINUTE_IN_MILLISECONDS);
        }
    };
    runnable.run();
}
```

Figure 2.5: How I used `Handler` in my app

Listeners rely on user input, and in this app, they relay information back to Main Activity, just like Broadcast Receivers and Handlers.

The **Main Activity** takes input from the user on what behaviours they want the app to track, and then starts the relevant Broadcast Receivers, Handlers and Listeners needed to track these particular behaviours. The red arrows in the system diagram indicate indirect communication from the user to Broadcast Receivers, Handlers and Listeners, since the user needs to go through the Main Activity to start these tracking mechanisms.

2.3 Permissions

The Android security architecture has been designed in a way such that no app, by default, is allowed to perform operations that would have a negative effect on other apps, the operating system, or the user [10]. Android permissions were made to support this design - their purpose is to protect the user's privacy. These permissions can be divided into three broad categories: **normal** permissions, **signature** permissions and **dangerous** permissions.

A **normal** permission is one which does not pose much risk to the user's privacy or the device's operation. If that permission is listed in the Android manifest file, the system automatically grants the permission to the app, and the user is not consulted.

Next, a **signature** permission is one that the system grants at install time, but only if the app requesting the permission is signed by the same certificate as the app that defines the permission.

Finally, a **dangerous** permission covers areas that *could* potentially affect the user's privacy or the device's normal operation. In this case, the user must explicitly agree to grant such a permission for the app to be able to provide functionality that depends on the permission.

Lists of each type of permission present in Android (as of API 28) are given below (Figures 2.6, 2.7 and 2.8). All permissions that this app requires are provided in the manifest file (`AndroidManifest.xml`).

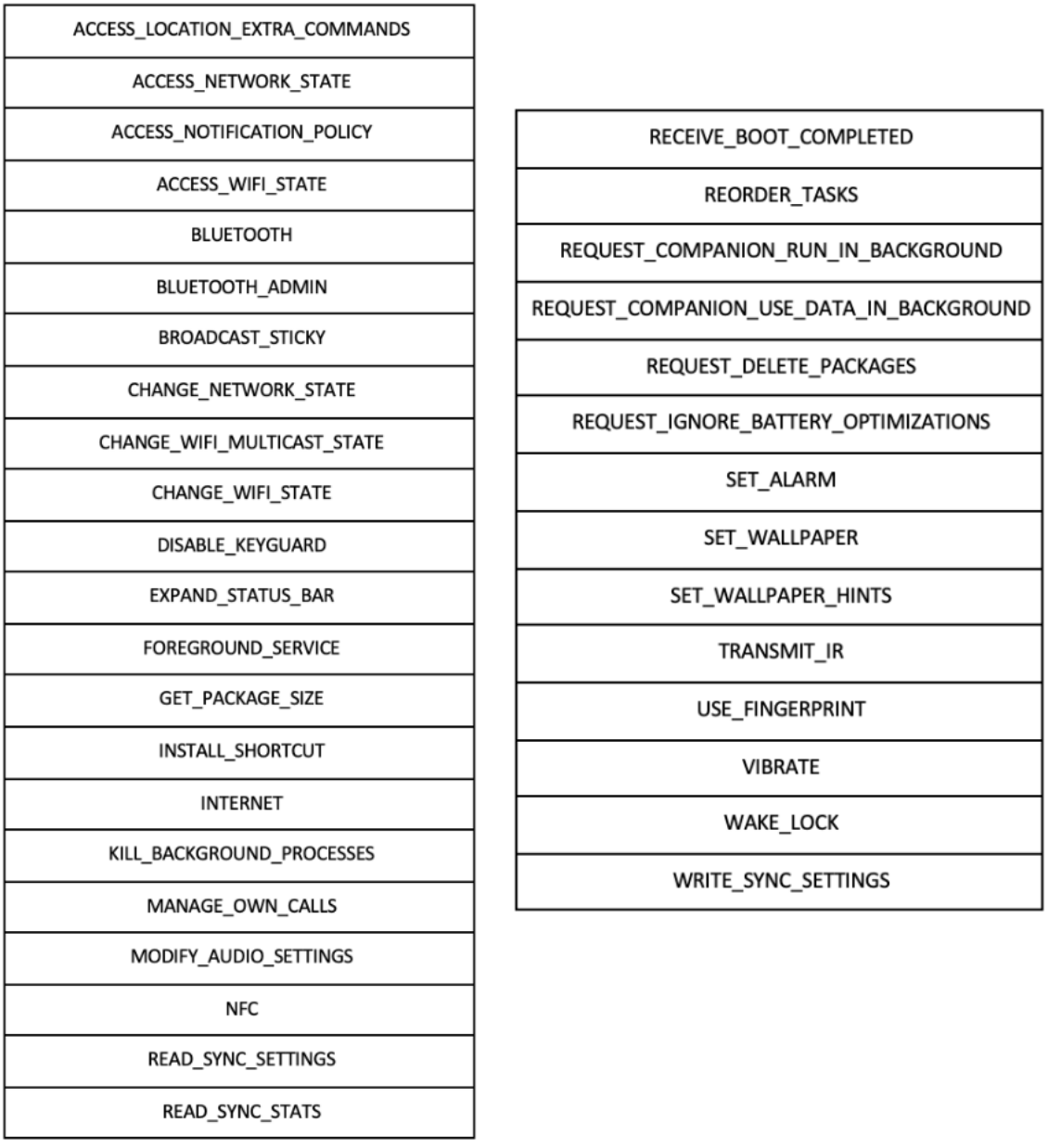

Figure 2.6: Normal permissions

BIND_ACCESSIBILITY_SERVICE
BIND_AUTOFILL_SERVICE
BIND_CARRIER_SERVICES
BIND_CHOOSER_TARGET_SERVICE
BIND_CONDITION_PROVIDER_SERVICE
BIND_DEVICE_ADMIN
BIND_DREAM_SERVICE
BIND_INCALL_SERVICE
BIND_INPUT_METHOD
BIND_MIDI_DEVICE_SERVICE
BIND_NFC_SERVICE
BIND_NOTIFICATION_LISTENER_SERVICE
BIND_PRINT_SERVICE
BIND_SCREENING_SERVICE
BIND_TELECOM_CONNECTION_SERVICE
BIND_TEXT_SERVICE
BIND_TV_INPUT
BIND_VISUAL_VOICEMAIL_SERVICE
BIND_VOICE_INTERACTION
BIND_VPN_SERVICE
BIND_VR_LISTENER_SERVICE
BIND_WALLPAPER
CLEAR_APP_CACHE
MANAGE_DOCUMENTS
READ_VOICEMAIL
REQUEST_INSTALL_PACKAGES
SYSTEM_ALERT_WINDOW
WRITE_SETTINGS
WRITE_VOICEMAIL

Figure 2.7: Signature permissions

PERMISSION GROUP	PERMISSIONS
CALENDAR	READ_CALENDAR
	WRITE_CALENDAR
CALL_LOG	READ_CALL_LOG
	WRITE_CALL_LOG
	PROCESS_OUTGOING_CALLS
CAMERA	CAMERA
CONTACTS	READ_CONTACTS
	WRITE_CONTACTS
	GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION
	ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE
	READ_PHONE_NUMBERS
	CALL_PHONE
	ANSWER_PHONE_CALLS
	ADD_VOICEMAIL
	USE_SIP
SENSORS	BODY_SENSORS
SMS	SEND_SMS
	RECEIVE_SMS
	READ_SMS
	RECEIVE_WAP_PUSH
	RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE
	WRITE_EXTERNAL_STORAGE

Figure 2.8: Dangerous permissions

Chapter 3

Design Overview

I was given a set of milestones to complete for my app. All of the milestones involved tracking smartphone security behaviours. These are based on the SSBS behaviours mentioned in Table 1.1 of the Introduction section. The security behaviours were divided into two groups: **technical** configuration actions and **social** configuration actions.

The **technical** configuration milestones were as follows:

1.1 Track changes in the configuration of Advertising ID

1.2 Track changes in the configuration of hide device in Bluetooth settings

1.3 Track changes of passcode/PIN for the smartphone's screen lock

1.4 Detect if the user physically/manually covers their smartphone's screen when in public spaces

1.5 Detect if the user uses adblocker(s)

1.6 Detect if the user uses anti-virus app(s)

1.7 Detect if the user uses VPN app(s) when connected to a public network

1.8 Detect if the user turns off WiFi when not actively being used

Meanwhile, the **social** configuration milestones were as follows:

2.1 Track the source of the app when the user performs financial and/or shopping tasks

2.2 Determine when downloading an app, if the user checks (or not) that the app is from the official/expected source (e.g. developer name)

2.3 Determine when downloading an app, if the user checks the source of apps (e.g. if they come from Google Play, Amazon App Store or other third party stores)

2.4 Determine if the user verifies the recipient/sender before sharing text messages or other information using smartphone apps

2.5 Determine if the user deletes any online communications (i.e., texts, emails, social media posts) that look suspicious

2.6 Determine if the user pays attention to the pop-ups on her smartphone when connecting it to another device (e.g. laptop, desktop)

3.1 Main Activity UI

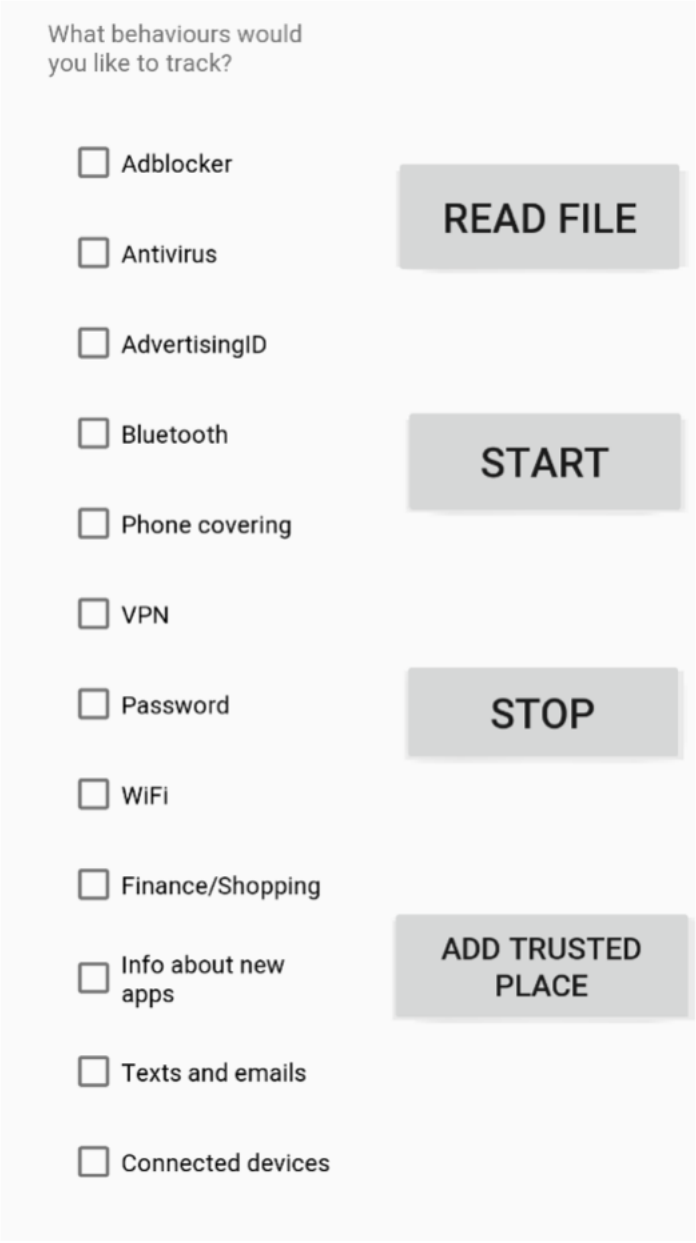

Figure 3.1: Main Activity UI

The user interface for this app's Main Activity is shown in Figure 3.1 above. The user ticks checkboxes in the Main Activity to specify which behaviours they want the app to track. They then click the ”Start” button to start tracking changes in these behaviours, and click the ”Stop” button to stop tracking changes. In general, every time a security behaviour that is being tracked changes, a message is written to a local file in the app, which records the timestamp, a unique id depending on the type of behaviour and a brief description of what has occurred. To read this file, the user can click on the ”Read file” button. To understand the purpose of the ”Add trusted place” button, please see the explanation for Problem 1.4 below.

3.2 Technical configuration milestones

In this section, I will provide details on how I implemented each technical configuration milestone.

3.2.1 Problem 1.1: Advertising ID

Problem 1.1 requires the app to track changes in the device's Advertising ID, which is a unique, user-resettable ID for advertising provided by Google Play Services [12]. This section relied on the Handler component.

My app uses an `AdvertisingClient`, as well as the `getAdvertisingIdInfo()` and `getId()` methods from that class, to get the devices Advertising ID. When the Handler is first started, the first-known value of the Advertising ID is written to the tracking file and saved into a variable called **id**. When the Handler next runs 1 minute later, it checks whether the polled Advertising ID is different from the saved value in **id**, and a message is written to the tracking file if this is the case. The Handler runs every minute, and this process is repeated to track changes in the Advertising ID.

Since the `getAdvertisingIdInfo()` method is a *blocking call* method [12] (i.e., a method that blocks the executing thread until the operation has finished [14]), it cannot be called on the main (UI) thread. Therefore, I made an `AdvertisingIdRunnable` class which starts a background thread to check for the Advertising ID, so as not to interfere with the main thread.

3.2.2 Problem 1.2: Bluetooth

I used a Broadcast Receiver for Problem 1.2, which required the app to track changes in the configuration of the hide device functionality in Bluetooth settings. However, in the newest versions of Android, I found that the device is *always* discoverable to other devices when connected to Bluetooth, so I adapted the milestone to just check whenever Bluetooth was switched on or off. I created a class called `BluetoothBroadcastReceiver`, which extends the `BroadcastReceiver` class, and I overrided the `onReceive()` method to check whether the message being received

from the system (also known as the intent action) says that the Bluetooth state has changed; in this case, the intent action to look out for would be `BluetoothAdapter.ACTION_STATE_CHANGED`.

Additional information can be gathered about the intent action by calling the `getIntExtra()` method and storing this extra value in an integer variable called `state`. `state` could be one of four values: `BluetoothAdapter.STATE_OFF`, `BluetoothAdapter.STATE_TURNING_OFF`, `BluetoothAdapter.STATE_ON` and `BluetoothAdapter.STATE_TURNING_ON`. The value of the `state` variable determines what message is written to the tracking file.

For this milestone, the app required two normal permissions: `android.permission.BLUETOOTH` and `android.permission.BLUETOOTH_ADMIN`.

3.2.3 Problem 1.3: Password

Problem 1.3 required the app to track changes in the device password. For this problem, I created a class called `PasswordReceiver`, which extended the class `DeviceAdminReceiver` - this is a subclass of `BroadcastReceiver` and is part of the Android Device Administration API. The Device Administration API is designed to support enterprise apps by providing device administration features at the system level [6]. I then simply overrided the inbuilt method `onPasswordChanged()` to write to the tracking file whenever the phones password is changed.

For this problem, the protected permission `android.permission.BIND_DEVICE_ADMIN` was required, as mentioned in the official documentation [7].

3.2.4 Problem 1.4: Phone covering

For Problem 1.4, which required checking whether physically covers their phone screen when in public spaces, I needed to check for three things before writing a message to the tracking file.

First, I checked whether the phone screen was being covered (by, for example, the user's hand). In order to check this, I created a `SensorEventListener` (a type of **Listener**) and overrided the `onSensorChanged()` method to detect when something was close to the proximity sensor (which is normally located at the top of an Android phone).

After checking whether the phone was being covered, the second condition I checked was whether the user was using their phone. I did this by checking whether the screen was unlocked. I created a method called `isScreenLocked()` and if the result of this was false, I assumed the phone screen to be unlocked, implying that the user was using their phone. In this case, I continued to check the third condition.

The third and final condition to check is whether the user is in a public space. In order to check this, I implemented **geofences**. Geofences are circular areas of a specified radius surrounding a location of interest, and they combine awareness of the users current location and awareness of the users proximity to locations of interest [5]. In my app, I used these to set up **trusted places** of 100m radius that the user can input into the app, using the ”Add trusted place” button mentioned in the Main Activity UI section. Clicking on this button leads the user to a screen (controlled by an Activity called **MapsActivity**) which implements the Google Maps API. Here, the user can either set the current location as a trusted place, or pick a point on the map and add that as a trusted place. Whenever the smartphone is outside one of these trusted places, the user is presumed to be in a public place.

Figure 3.2: Google Maps API in my app. Accessible from the ”Add trusted place” button (see Main Activity UI section)

I extended the `BroadcastReceiver` class to make a new class called `GeofenceBroadcastReceiver`. This checks for a `GeofencingEvent`, and gets the transition type if there is one. It then checks whether the geofence transition was *GEOFENCE_TRANSITION_DWELL* (meaning the user has been inside the radius of one of their trusted places for some time) or *GEOFENCE_TRANSITION_EXIT* (which means the user has left a trusted place). In the case of the latter, it now presumes the user is in a public place. At this stage, the three conditions have been satisfied, and so a message is written to the tracking file indicating that the phone is being covered while being used in a public space.

For this problem, the normal permission `android.hardware.sensor.proximity` was needed for the `SensorEventListener` to work. In addition, the **dangerous** permission `android.permission.ACCESS_FINE_LOCATION` was required for the app to access the user's current location, both when displaying this location on screen as part of the Google Maps API, and also when checking where the user is in relation to any geofences that have been set up. Since this is a dangerous permission, the user needs to explicitly give their permission for the app to track their current location.

3.2.5 Problem 1.5: Adblocker

I implemented Problems 1.5 and 1.6 in extremely similar ways since the milestones themselves are nearly identical. Therefore, I thought it would be best to discuss the implementation of these milestones in one section, since they are so closely linked.

These two problems both rely on the Handler in my app, since, at the time of writing this report, there are no Broadcast Receivers that receive messages from the system whenever a particular app is running.

I decided to manually create two whitelists: one containing the package names of the 'top' 15 adblocking apps on the Google Play Store, and one containing the 'top' 15 antivirus apps on the Google Play Store. To find the 'top' apps for each milestone, I searched 'adblocker' into the store for Problem 1.5 and 'antivirus' for Problem 1.6; I then took the first 15 results that appeared on the Google Play Store for these searches and added them to the respective whitelist.

I wanted to create two new Java classes for the adblocking app and antivirus app whitelists, and since I knew these two classes would be very similar, I decided to make a Java Interface called `Whitelist`, which the two new classes would implement. This interface contains one method - `getSet()` - which returns the relevant Java Set of package names. I decided to store the package names in a Java Set because I knew I would need to use its `contains()` method for these milestones, and the time complexity of this method is O(1) [15].

For Problem 1.5, I created a class called `AdBlockerWhitelist`, which implements the `Whitelist` interface. `AdBlockerWhitelist` includes a Java HashSet of Strings called

`adBlockerSet`, and this contains the package names of the top 15 adblocking apps on the Google Play Store. The class also overrides the `Whitelist` method `getSet()`, and simply returns `adBlockerSet` when this method is called.

Similarly, for Problem 1.6, I created a class called `AntivirusWhitelist`, which also implements the `Whitelist` interface. `AntivirusWhitelist` also includes a Java HashSet of Strings, this time called `antivirusSet`, which contains the package names of the top 15 antivirus apps on the Google Play Store. Once again, this class also overrides the `Whitelist` method `getSet()`, and simply returns `antivirusSet` when this method is called.

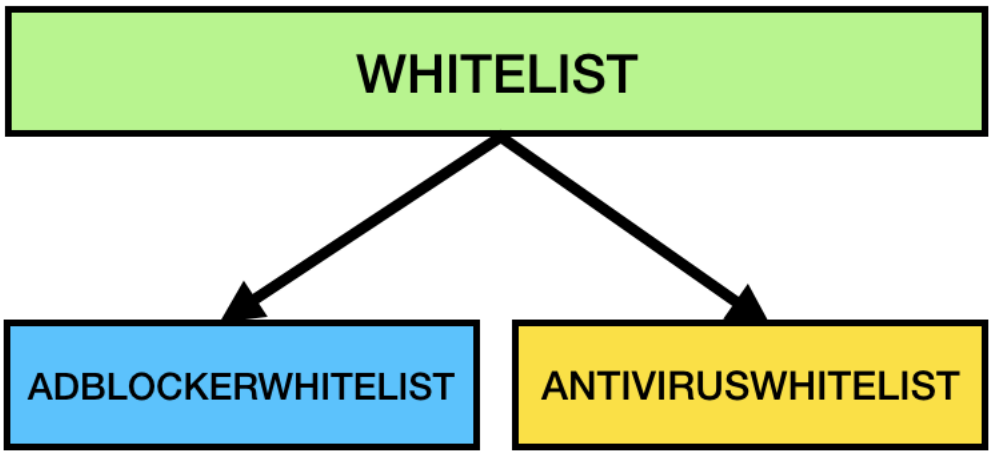

Figure 3.3: Relationship between `Whitelist`, `AdBlockerWhitelist` and `AntivirusWhitelist`

For both problems, I needed a method which would check if any of the apps in the relevant whitelist were running at a certain point in time. To get this information, I created a method called `isAppRunning()`, which takes a `Whitelist`, goes through the list of currently running app processes and checks if any of these processes are part of the `Whitelist`. If there is a running process that is part of the whitelist, then an adblocking/antivirus app has been found to be running on the phone, so the package name of this app is returned. Otherwise, the empty string (“”) is returned.

```
public String isAppRunning(Set<String> whitelist) {
    final ActivityManager activityManager =
            (ActivityManager) getApplicationContext().getSystemService(Context.ACTIVITY_SERVICE);
    final List<ActivityManager.RunningAppProcessInfo> procInfos =
            activityManager != null ? activityManager.getRunningAppProcesses() : null;
    if (procInfos != null) {
        for (final ActivityManager.RunningAppProcessInfo processInfo : procInfos) {
            if (whitelist.contains(processInfo.processName)) {
                return processInfo.processName;
            }
        }
    }
    return "";
}
```

Figure 3.4: `isAppRunning()` method

I then created two methods for my Handler so that Problems 1.5 and 1.6 could be tracked effectively: `adBlockerHandlerActivity()` and `antivirusHandlerActivity()`. Both of these methods call the `isAppRunning()` method on the relevant whitelist, and if the String returned from this method is non-empty, a message is written to the tracking file stating that an adblocker/antivirus app is running. Once the Handler has started, if the user has ticked the Adblocker checkbox in MainActivity, `adBlockerHandlerActivity()` is called. Similarly, if the user has ticked the Antivirus checkbox in MainActivity, `antivirusHandlerActivity()` is called.

3.2.6 Problem 1.6: Antivirus

See Problem 1.5 above.

3.2.7 Problem 1.7: VPN

Next, Problem 1.7 requires the app to detect if the user uses VPN app(s) when connected to a public network. There are two layers to this problem: detecting whether a VPN is running and checking if the device is connected to a public network.

For the first issue of checking whether a VPN is running, I created a method called `isVPNOn()`, which goes through the list of open network connections and checks if any of them starts with 'tun' (e.g., 'tun0', 'tun1'). This is checked because the Android system automatically routes VPN connections to these 'tun'networks.

The app then checks whether the device is connected to a public network. First, the app checks if the phone is connected to WiFi at all, using Androids `WifiManager` class, and then it checks for two things: whether the WiFi has no password, and whether the WiFi is a captive portal. If either of these are true, then the network is presumed to be public. I created a method called `isWifiNotPasswordProtected()`, which returns true if the current WiFi network is not a WEP, WPA or WPA2 network, in which case the WiFi is presumed to not be password protected. I also created a method called `isCaptivePortal()` to check whether the WiFi is a captive portal,

and it does so by checking if the currently active network has the network capability `NetworkCapabilities.NET_CAPABILITY_CAPTIVE_PORTAL`.

Therefore, if the WiFi is connected and either not password protected or a captive portal, and if a VPN is running, a relevant message is written to the tracking file.

For this milestone, the app required three normal permissions: `android.permission.ACCESS_NETWORK_STATE`, `android.permission.ACCESS_WIFI_STATE` and `android.permission.INTERNET`. All of these permissions were needed to access information about the WiFi.

3.2.8 Problem 1.8: WiFi

Problem 1.8 requires that the app detects if the user turns off WiFi when not actively being used. To check whether the internet is in use, I created a method called `checkForActiveConnections()`, which checks if the list of TCP and TCP6 connections on the device is empty by reading the files `/proc/net/tcp` and `/proc/net/tcp6`. If this list is empty, the device is presumed to not be actively using the internet. I then created a class called `WifiBroadcastReceiver` to extend the `BroadcastReceiver` class. This checks for the action `WifiManager.WIFI_STATE_CHANGED_ACTION`, and only if the state of the WiFi is `WifiManager.WIFI_STATE_DISABLING` (i.e., the WiFi is in the process of switching off) does the app read the list of TCP and TCP6 connections. If the WiFi is being switched off and the TCP and TCP6 connection list is empty, a message is written to the tracking file indicating this.

Since this milestone also accesses information about the WiFi, it requires the same three permissions as Problem 1.7 above: `android.permission.ACCESS_NETWORK_STATE`, `android.permission.ACCESS_WIFI_STATE` and `android.permission.INTERNET`.

3.3 Social configuration milestones

In this section, I will provide details on how I implemented each social configuration milestone.

3.3.1 Problem 2.1: Finance/Shopping

For Problem 2.1, the app was required to track the source of the app when the user performs financial and shopping tasks. I implemented this in a very similar way to the way I implemented adblocking and antivirus app milestones (Problems 1.5 and 1.6) – I created a new `Whitelist` called `FinanceShoppingWhiteList` and used the `isAppRunning()` method once again (see Figure 3.4) to see if the current foreground activity was one of the apps from this whitelist. However, in this case, instead of manually adding the top 15 finance and shopping apps on the Google Play Store to my whitelists, I was able to use a Google Play crawler created by George Hage. This

provided me with the top 100 finance and shopping apps on the Google Play Store, organised by number of reviews.

3.3.2 Problem 2.2: Developer name

For Problem 2.2, my task was to determine when downloading an app, if the user checks that the app is from the official/expected source (e.g. developer name). Unfortunately, I was not able to find any inbuilt methods that allowed me to access the developer name of an app, so I was not able to complete this milestone.

3.3.3 Problem 2.3: Store name

My task for Problem 2.3 was to determine when downloading an app, if the user checks the source of apps (e.g. if they come from Google Play, Amazon Appstore or other third-party stores). Firstly, I created an `AppInstallBroadcastReceiver` (a sub-class of `BroadcastReceiver`), which checks for the action `Intent.`*`ACTION_PACKAGE_ADDED`* - this message is sent out by the system whenever an app is downloaded. I then used `PackageManager` to determine which app store the app had been downloaded from and saved this into a String called `storeName`.

I decided that the best way to find out if the user had checked the source of the apps they were downloading was to ask them directly, using a small pop-up box on screen. Therefore, inside the `AppInstallBroadcastReceiver`, once the `Intent.`*`ACTION_PACKAGE_ADDED`* message has been received, an `AlertDialog` (a small window that asks the user to make a decision or input some information [8]) is displayed on the app screen asking the user if they know which store they just downloaded this app from. They can choose either 'Yes'or 'No', and if they pick 'Yes', they are asked to input the name of the store they think it came from. If the user input matches with the value in `storeName`, a message is written to the tracking file indicating that the user has checked the source of the app they just downloaded.

3.3.4 Problem 2.4: Text verification

Due to the time constraints of my research placement, I was not able to complete this milestone.

3.3.5 Problem 2.5: Suspicious online communications

Due to the time constraints of my research placement, I was not able to complete this milestone.

3.3.6 Problem 2.6: Pop-ups

Due to the time constraints of my research placement, I was not able to complete this milestone.

Chapter 4

Testing and Evaluation

When it came to testing and evaluating my project, I took a different approach for each milestone. For some milestones, testing the milestone repeatedly was enough to virtually guarantee its reliability, whereas for others, I needed to consider other factors as well in order to assess whether my implementation of the milestone was sufficient for the purposes of this research. I have outlined how I went about testing each milestone below.

4.1 Simple repeated testing

I repeatedly tested some of the milestones to see how many times out of a total number of attempts the app was able to identify the target action for each milestone.

For each milestone, I decided to test the relevant action by printing a message to the `Logcat` (a tool which allows the developer to see system messages [9]) if the appropriate change had occurred. I did this **50** times for each milestone. I have provided details on the methodology I used to test each of the aforementioned milestones, as well as a table of results (Table 4.1) for this stage of the testing, below.

4.1.1 Methodology

- **Problem 1.1** - I manually reset the Advertising ID.
- **Problem 1.2** - I switched Bluetooth on and off.
- **Problem 1.3** - I changed the phone password.
- **Problem 1.4** * - I moved my hand close to the phone's proximity sensor.
- **Problem 1.7** - I ran a VPN while the phone was connected to a public captive network.
- **Problem 1.8** - I switched WiFi off.
- **Problem 2.3** - I downloaded an app.

[* I only tested one part of this milestone using repeated testing.]

4.1.2 Results

Problem	No. of times app tracked changes	No. of times app did not track changes
1.1	50	0
1.2	50	0
1.3	50	0
1.4	50	0
1.7	50	0
1.8	50	0
2.3	50	0

Table 4.1: Table of results for repeated testing

4.2 Other types of testing

4.2.1 Adblocking, antivirus, finance and shopping apps

I implemented Problems 1.5, 1.6 and 2.1 in very similar ways (as discussed in Section 3.2) and so I decided to test them in similar ways too. For Problem 1.5, I downloaded the 15 apps listed in `AdBlockerWhiteList` and tested each app 10 times to see if my app detected that it was running. For Problem 1.6, I did the same thing but downloaded the 15 apps listed in `AntivirusWhiteList` and tested each of these apps 10 times too. For Problem 2.1, I downloaded the top 15 apps (out of 100) for both Finance and Shopping categories (taken from `FinanceShoppingWhitelist`) and tested each of these apps 10 times as well.

Unfortunately, after I started testing these milestones, I came to realise that the method `getRunningAppProcesses` from `ActivityManager` (which my `isAppRunning` method relies on heavily) has become severely restricted in the latest versions of Android, and that it is now much more difficult to get information on apps running in the foreground and background. Therefore, instead of checking whether the app is running, I check whether it is installed using a method I created called `isPackageInstalled` (see Figure 4.1 below).

The testing results of the three milestones using `isPackageInstalled` rather than `isAppRunning` are shown below.
[NB: The number of test results given above in Table 4.2 is the total: for each milestone, the 15 apps were tested 10 times each, so 150 results are shown above]

```
private boolean isPackageInstalled(String packageName) {
    PackageManager packageManager = getPackageManager();
    boolean found = true;
    try {
        packageManager.getPackageInfo(packageName, i: 0);
    } catch (PackageManager.NameNotFoundException e) {
        found = false;
    }
    return found;
}
```

Figure 4.1: `isPackageInstalled()` method

Problem	No. of times app tracked changes	No. of times app did not track changes
1.5	150	0
1.6	150	0
2.1	150	0

Table 4.2: Table of results for Problems 1.5, 1.6 and 2.1

4.2.2 Problem 1.4: Phone covering

Although I tested the phone's proximity sensor for this problem using repeated testing (as shown in Section 4.1 above), I did not test the geofences in this way. I was, in fact, unable to test geofences very well, and this was as a result of a number of factors. Firstly, the phone I was provided with did not have a SIM card, so it had no access to mobile networks. In addition, the testing was done in a rural area, and so there were very few public WiFi networks. Both of these factors meant that it was very difficult to detect when the phone had left a geofenced trusted place, since location accuracy is improved significantly when the device is connected to the internet. However, I was able to check that the geofences had been set up by printing a message to the logcat when this was done, and I also believe that the Google Maps API displays the correct current location when connected to WiFi, since I tested it in 3 public places with WiFi.

4.3 Evaluation

Overall, I believe the testing for this app was satisfactory. Nonetheless, I still think were a few parts of my testing strategy that could be improved upon, such as the following:

- The adblocker and antivirus whitelists (**Problems 1.5** and **1.6**) were found manually and are only accurate as of July 2019. I hope that in the future, the

app could be updated to use data from a Google Play crawler (such as the one used for **Problem 2.1**).

- In **Problem 1.4**, besides the issue of not having tested the geofences properly (as outlined above in Section 4.2.2), the phone–covering aspect of the milestone is slightly ambiguous. For example, if the user decided to cover the bottom half of the phone, the proximity sensor (which is located towards the top of the phone) would not detect any changes.
- Not being able to use the `isAppRunning()` method for **Problems 1.5, 1.6** and **2.1** was frustrating, since having any of these apps installed does not necessarily imply that the user actively uses them.
- With regards to **Problem 1.4**, geofences are not the ideal way to measure whether the user is in a public place. If the user does not register a location that they trust as a 'trusted place' on their phone, the phone may incorrectly assume they are in a public place when they are actually not (for example, when visiting the homes of friends and family).

Chapter 5

Conclusion

This app has been created in order to strengthen the claim made by the University of Urbana-Champaign that there is a correlation between certain smartphone security behaviours and mental health issues. This claim was based on the self-reported responses of 487 participants of an online survey [18]. By tracking these security behaviours in real-time, the app will be able to provide objective evidence about how people use their smartphones in everyday life, thus avoiding the large differences that typically arise when comparing self-reported behaviour and actual behaviour.

I hope that the use of this app in a future field study will help to further corroborate this claim, and that this research will not only lead to improved smartphone security in real-world environments (such as the workplace, schools and healthcare institutions), but also save these organisations a lot of time, money and effort in their attempts to protect themselves from cybercrime and online adversaries.

Bibliography

[1] Amazon: What is pub/sub messaging? https://aws.amazon.com/pub-sub-messaging/.

[2] Android developer guide: Activity. https://developer.android.com/reference/android/app/Activity.

[3] Android developer guide: BroadcastReceiver. https://developer.android.com/reference/android/content/BroadcastReceiver.

[4] Android developer guide: Broadcasts. https://developer.android.com/guide/components/broadcasts.

[5] Android developer guide: Create and monitor geofences. https://developer.android.com/training/location/geofencing.

[6] Android developer guide: Device administration overview. https://developer.android.com/guide/topics/admin/device-admin.html.

[7] Android developer guide: DeviceAdminReceiver. https://developer.android.com/reference/android/app/admin/DeviceAdminReceiver.

[8] Android developer guide: Dialogs. https://developer.android.com/guide/topics/ui/dialogs.

[9] Android developer guide: Logcat command-line tool. https://developer.android.com/studio/command-line/logcat.

[10] Android developer guide: Permissions. https://developer.android.com/guide/topics/permissions/overview.

[11] Android developer guide: View. https://developer.android.com/reference/android/view/View.

[12] Android docs: Advertising ID. http://www.androiddocs.com/google/play-services/id.html.

[13] Android event handling. https://www.tutorialspoint.com/android/android_event_handling.htm.

[14] What is blocking methods in java and how do deal with it?, February 2017. https://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html?m=1.

[15] Time complexity of Java collections, July 2019. https://www.baeldung.com/java-collections-complexity.

[16] Anonymous. The maturing of the smartphone industry is cause for celebration. *The Economist*, 2019. https://www.economist.com/leaders/2019/01/12/the-maturing-of-the-smartphone-industry-is-cause-for-celebration.

[17] A. Das and H. U. Khan. Security behaviors of smartphone users. *Information & Computer Security*, 24(1):116–134, 2016.

[18] S. Demetriou et al. Smartphone security behavioral scale: A new psychometric measurement for smartphone security (unpublished).

[19] S. Egelman and E. Peer. Scaling the security wall: Developing a security behavior intentions scale (sebis). In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2873–2882. ACM, 2015.

[20] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE security & privacy*, 7(1):50–57, 2009.

[21] G. Gross. The cost of cybercrime, February 2018. https://www.internetsociety.org/blog/2018/02/the-cost-of-cybercrime/.

[22] K. Meaney. Apple's iPad is most popular tablet in schools, August 2017. https://www.simbainformation.com/Content/Blog/2017/08/07/Apples-iPad-is-Most-Popular-Tablet-in-Schools.

[23] M. Mikulic. Number of mHealth apps available at Google Play from 1st quarter 2015 to 2nd quarter 2019, August 2019. https://www.statista.com/statistics/779919/health-apps-available-google-play-worldwide/.

[24] A. Sinhal. Handler in android. *Medium*, January 2017.

[25] L. Vogel, 2017. https://www.vogella.com/tutorials/AndroidBackgroundProcessing/article.html#why-using-concurrency.