Monitoring smartphone users’ security behaviour

# Introduction

It is estimated that nearly three-quarters of the world’s adult population now owns a smartphone **(1).** 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 **(26).** 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 **(27).** 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) (**6),** 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 below.

|  |
| --- |
| 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). |

My task was to create an Android application that could be used in the real world to track the security behaviours 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.

**Applications of this research**

**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 **(5).**

Researching the correlation between users’ smartphone security behaviours and 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.

**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 **(28).** 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.

**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 (**29**). 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, therefore, improve the safety of the school environment as a whole.

# (App) Background

**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.

*Activities*

In plain terms, an AndroidActivityis a single, focussed thing that the user can do (**8**). Activities define the app’s user interface (UI), and typically, there is one activity per “screen” of the app (**9**). Activities are implemented by extending the inbuilt *Activity* class, and every app has a MainActivity, which is an Activity that related to the first screen the user sees when launching the app.

*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 (**11**) (a form of asynchronous service-to-service communication, where any message published to a topic is immediately received by all subscribers to the topic (**12**)). Apps can register to receive particular broadcasts so that system changes can be detected in real time.

**Complete list of system broadcast actions, as of API 28**

**Accessed from Android SDK file: sdk/platforms/android-28/data/broadcast\_actions.txt**

**BROADCAST ACTIONS LIST TABLES – INSERT HERE FROM OTHER DOCUMENT**

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 (**9**), 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.

*Handlers*

A Handler is an Android component which allows communication between a background thread and the main UI thread (**14**). 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 (**15**).

*Event 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) (**16**). 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 (**17**)).

**Interaction between components**

<System Diagram>

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.

**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.

**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 (**18**). 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. All permissions that this app requires are provided in the manifest file (AndroidManifest.xml).

<LISTS OF PERMISSIONS – NORMAL, SIGNATURE, DANGEROUS>

# Design Overview

I was given a set of milestones to complete for my app. All of the milestones involved tracking smartphone security behaviours. These security behaviours were divided into two groups: **technical** configuration actions and **social** configuration actions.

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

**Problem 1.1** Track changes in the configuration of Advertising ID.

**Problem 1.2** Track changes in the configuration of hide device in Bluetooth settings.

**Problem 1.3** Track changes of passcode/PIN for the smartphones screen lock .

**Problem 1.4** Detect if the user physically/manually covers her smartphone’s screen when in public spaces.

**Problem 1.5** Detect if the user uses adblocker(s).

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

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

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

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

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

**Problem 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).

**Problem 2.3** 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).

**Problem 2.4** Determine if the user verifies the recipient/sender before sharing text messages

or other information using smartphone apps.

**Problem 2.5** Determine if the user deletes any online communications (i.e., texts, emails,

social media posts) that look suspicious.

**Problem 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).

I have provided details on how I implemented each milestone in sections **3.2** (Technical configuration milestones) and **3.3** (Social configuration milestones) below.

**Main Activity UI**

<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.

**Technical Configuration Actions**

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

**Problem 1.1**

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 (**19**). 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 device’s 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 (**19)** (i.e., a method that blocks the executing thread until the operation has finished (**20**)), 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.

**Problem 1.2**

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.

**Problem 1.3**

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** andis 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 (**21**). I then simply overrided the inbuilt method **onPasswordChanged()** to write to the tracking file whenever the phone’s password is changed.

For this problem, the normal permission android.permission.BIND\_DEVICE\_ADMIN was required, as mentioned in the official documentation (**22**).

**Problem 1.4**

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 there was any foreground activity running on the phone. The foreground activity at any moment in time is the activity on the screen that the user is currently interacting with, and is considered the most important (**8**). By checking whether a foreground activity was running, I was able to determine if the user was covering their phone while actually using it for something, or whether the phone was idle, in which case there would be nothing to track. To check whether there was a running foreground activity, I created a method called **isThereAnyForegroundActivity()**, which goes through the list of running processes on the phone and checks whether the importance of any of these processes is **ActivityManager.RunningAppProcessInfo.*IMPORTANCE\_FOREGROUND***. If there is an activity with this level of importance, that means a foreground activity *is* currently running, and so the third condition can be checked.

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 user’s current location and awareness of the user’s proximity to locations of interest (**23**). 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 section 3.1. Clicking on this button leads the user to a screen implementing the Google Maps API, where they 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.

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 using this permission.

**Problems 1.5 and 1.6**

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. 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) (**24**).

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.

<Diagram of interfaces>

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.

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.

**Problem 1.7**

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 Android’s **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.

**Problem 1.8**

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 set up a **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.

**Social Configuration Actions**

**Problem 2.1**

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 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.

**Problem 2.2**

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.

**Problem 2.3**

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 **AppInstallBroadcastReceiver** (a subclass 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, a …… is displayed on 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.

<PICTURE OF ALERTDIALOG>

**TALK ABOUT PERMISSIONS AT THE END OF EVERY MILESTONE WHERE THEY WERE NEEDED**

# Evaluation

Some things to improve:

* 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 (like the one used for **Problem 2.1**)
* Hand covering ambiguous – only works if hand is covering the proximity sensor at the top of the phone

# Conclusion

# Bibliography

1. <https://www.economist.com/leaders/2019/01/12/the-maturing-of-the-smartphone-industry-is-cause-for-celebration>
2. <https://www.ca.com/en/blog-highlight/cybersecurity-in-the-workplace-its-everyones-business.html>
3. <https://enterprise.verizon.com/resources/reports/dbir/>
4. <https://www.siliconrepublic.com/careers/cybersecurity-culture-workplace>
5. <https://www.internetsociety.org/blog/2018/02/the-cost-of-cybercrime/>
6. Scaling the Security Wall: Developing a Security Behavior Intentions Scale (SeBIS)
7. Counter Measuring Conceivable Security Threats on Smart Healthcare Devices
8. <https://developer.android.com/reference/android/app/Activity>
9. Understanding Android Security
10. <https://developer.android.com/guide/components/activities/intro-activities>
11. <https://developer.android.com/guide/components/broadcasts>
12. <https://aws.amazon.com/pub-sub-messaging/>
13. <https://developer.android.com/guide/components/fundamentals>
14. <https://medium.com/@ankit.sinhal/handler-in-android-d138c1f4980e>
15. <https://www.vogella.com/tutorials/AndroidBackgroundProcessing/article.html#why-using-concurrency>
16. <https://www.tutorialspoint.com/android/android_event_handling.htm>
17. <https://developer.android.com/reference/android/view/View>
18. <https://developer.android.com/guide/topics/permissions/overview>
19. <http://www.androiddocs.com/google/play-services/id.html>
20. <https://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html?m=1>
21. <https://developer.android.com/guide/topics/admin/device-admin.html>
22. <https://developer.android.com/reference/android/app/admin/DeviceAdminReceiver>
23. <https://developer.android.com/training/location/geofencing>
24. <https://www.baeldung.com/java-collections-complexity>
25. <https://stackoverflow.com/questions/54954779/how-does-android-know-that-it-has-to-direct-allowed-vpn-application-to-tun0>
26. Security behaviors of smartphone users (Das and Khan, 2015)
27. Smartphone Security Behavioral Scale: A New Psychometric Measurement for Smartphone Security (**unpublished**)
28. <https://www.statista.com/statistics/779919/health-apps-available-google-play-worldwide/>
29. <https://www.simbainformation.com/Content/Blog/2017/08/07/Apples-iPad-is-Most-Popular-Tablet-in-Schools>