Employing Android Security Features for Enhanced Security and Privacy Preservation

by

Mike Wakim

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the Master of Applied Science degree
in Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

In this thesis, we examine the architecture and the security framework underlying the Android operating system. We explore existing Android end-to-end encrypted (E2EE) messaging applications and derive four categories of common issues that are applicable to these applications. We then provide an overview of the known issue of privilege escalation wherein a malicious privileged application can utilize inter-process communication techniques to send protected data to an unauthorized application on a user's device. We demonstrate through a proof of concept how this behavior can be achieved in real applications, and we suggest potential countermeasures that can help prevent this issue. Furthermore, in the interest of diminishing the common issues that are applicable to E2EE messaging applications, we propose a new design for such applications that employs some of the principal security features offered by the Android operating system. We explain how our design can help eliminate trust-related issues associated with such applications, as well as how it can help minimize issues in other categories. Finally, we demonstrate how our proposed design can be used in practice by implementing a proof of concept.

Acknowledgements

First, I would like to express my sincere thanks to Dr. Carlisle Adams for his unwavering support and guidance throughout my studies at the University of Ottawa. This thesis would not have been possible without Dr. Adams' feedback and encouragement. I would also like to thank my friends and colleagues at the University of Ottawa for their feedback and suggestions in our monthly meetings. Finally, I would like to thank my amazing family for their constant support and encouragement.

.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Introduction

The Android operating system has been very successful ever since it was introduced into the mobile market. The open-source operating system attracted a lot of attention since its debut. It has already been deployed on more than 4000 different products across the world [1], through 400 different mobile device manufacturers [2, 3]. Although the operating system was initially most popular for smartphone and tablet devices, it is now being used in a variety of mobile devices including smartwatches, automobiles, smart TVs, home appliances, etc. [2, 4]. As of November 2016, there are two and a half million applications on the Google Play store alone [5]. However, the Google Play store is not the only source where people can download Android applications. In fact, users have the option to enable application downloads from "Unknown Sources", i.e. anywhere they can find an Android Application Package (APK), and this makes the number of available Android applications figuratively limitless. There are Android applications for pretty much anything one can think of. The applications are typically associated with certain categories (e.g. Games, Education, Lifestyle, etc.). Regardless of what category we look into, a lot of Android applications have access to and sometimes collect some of the personal information available on their users' devices. Sometimes, the collection and use of this personal information is actually a requirement for an application's proper functioning. For example, it would make sense if a 3rd party text messaging application that a user installs to substitute the native text messaging application requires access to the user's text messages; the application simply would not function

as intended if it does not have access to the data it needs. However, a lot of times, the collection of personal information is not a requirement at all for the proper functioning of an application itself. Instead, certain companies collect user data as part of their business model, in order to generate more money. Companies can generate money from their users' data in multiple ways, one of which is through big data advertising. Essentially, they sell users' data to advertising networks which can then generate targeted ads with higher click through and conversion rates. Furthermore, user data can be used for enhanced marketing, product development, or even building giant data profiles for any imaginable use case in which the data can be required [6]. This all sounds very reasonable when looking at it from a business perspective: it is undeniable that user data can be extremely useful for business purposes. However, while the businesses are benefitting as such, it is the application users that are putting their personal information at stake, providing access of their personal information to applications in exchange for their offered services. These privacy issues are in no way limited to Android devices only. In fact, these issues apply to most smartphone and tablet devices, regardless of the involved operating system. They even apply to most web applications and websites, most commonly through the use of cookies that can build user profiles based on users' inputs and browser behavior [7]. In this thesis, we focus on the privacy and security relating to the Android operating system.

With regard to Android devices, it is important to note that 3rd party applications do not have access to all the data resources available on a user's device by default. The data resources on Android devices are typically accessible through what we refer to as "content providers", and they include elements such as text messages, phone call log information, calendar data, and much more. To gain access to these content providers, applications must request a permission from their users.

For instance, to gain access to the text messages available on a user's device, an application must request the permission known as "`android.permission.READ_SMS`". This permission, along with numerous others, are all managed by Android's permission model, which plays an essential role in the architecture of the operating system.

Although lots of applications are privacy infringing, there is a multitude of applications that aim to help users preserve their privacy on said devices. Some of these applications help users increase their awareness in relation to their privacy. They analyze the applications installed on a user's device and inform the user about how privacy infringing certain applications can potentially be, and thereafter, users can choose to keep or delete applications based on this knowledge. Other applications make use of cryptographic techniques to secure their users' data. There are applications for secure messaging, private calling, secure e-mailing, password managing, file encrypting and many more.

Over the course of the past few years, secure messaging applications have been rapidly increasing in popularity [8, 9]. There are various applications available on the Google Play store that support end-to-end encrypted (E2EE) messaging, some of which have been installed by hundreds of millions of users [10]. Although these applications share a common goal, which is to allow users to privately communicate over the internet, different applications often offer their users different security properties. Due to the important role that E2EE messaging applications play when it comes to securing users' sensitive information, several studies and research papers have been published, wherein the authors focus on analyzing and evaluating the security offered by

these applications. Unfortunately, there are issues that are often associated with E2EE messaging applications.

1.2 Thesis Contribution

To build secure applications, while at the same time preserving users' privacy, it is important to understand the architecture underlying the Android operating system. There are several features available through the Android operating system that can be applied to achieve this. These security features include the user-based permission model, process isolation, inter-process communication mechanisms, and many more. In this thesis, we focus on exploring and analyzing the architecture underlying the Android operating system in the interest of applying some the existing security features to produce applications that offer their users enhanced security and privacy preservation. Moreover, we explore existing end-to-end encrypted messaging applications available for the Android operating system, and we review the common issues that are associated with some of these applications. Our examination of the operating system and of existing E2EE messaging applications led us to two main contributions:

i. We provide an overview of the known issue of privilege escalation wherein an unprivileged application that is deployed on a user's device can gain unauthorized access to protected data (e.g. GPS location) by employing inter-process communication mechanisms to retrieve the data from a malicious privileged application. We demonstrate in a proof of concept how this unauthorized access of data can occur in practice. Furthermore, we list

and discuss potential countermeasures that can help prevent or reduce the impact caused by this issue.

ii. We identify a way to apply some of the studied security features inside end-to-end encrypted messaging applications, in the aim of rendering the security offered by these applications more transparent and trustworthy. In this thesis, we show that issues with end-to-end encrypted messaging applications fall in four principal categories: (1) trust-related issues, (2) protocol-related issues, (3) implementation-related issues and (4) metadata-related issues. We propose a design that is different from the conventional design which is typically implemented by E2EE messaging applications. We explain how our design can help reduce / eliminate some of the common issues associated with such applications, especially the trust-related issues. Furthermore, we demonstrate in a proof of concept how our design can be used in practice.

1.3 Thesis Organization

The upcoming chapters will be organized as follows:

Chapter 2: We examine the architecture underlying the Android operating system. We focus on the Android Software Stack, the principal components that constitute Android applications, and the principal security features offered by the operating system. We also present a literature review where we focus on existing research regarding some of these security features.

Chapter 3: We explain the main concept of end-to-end encryption and we list the security features that are often desired in E2EE messaging applications. We examine existing work that has to do with analyzing the security of such applications, and we present a literature review. Finally, we list and discuss four categories that represent common issues that are typically associated with E2EE messaging applications.

Chapter 4: We provide an overview of the known issue of privilege escalation wherein an unauthorized application can access protected data on a user's device by retrieving it from a malicious privileged application through the utilization of inter-process communication mechanisms. We demonstrate through a proof of concept how this unauthorized access of data is possible in practice. Furthermore, we discuss potential countermeasures that can help prevent or raise user awareness regarding occurrences of this issue.

Chapter 5: We examine the conventional design that is typically implemented by E2EE messaging applications, and discuss the effect that this design has with regard to the common issues applicable to these applications. We propose a new design, which employs security features that we discuss in Chapter 2 in the interest of diminishing these common issues. Furthermore, we demonstrate through a proof of concept how our design can be put to practice.

Chapter 6: We summarize and conclude our thesis. We also suggest directions for future work regarding the countermeasure techniques that we presented in Chapter 4, and the design that we proposed in Chapter 5.

Appendix A: We present details regarding the proof of concept applications that we built for Chapter 4.

Appendix B: We present details regarding the proof of concept system that we built which implements the design that we proposed in Chapter 5.

Chapter 2: Android's Software Framework

2.1 Introduction

To build a secure application on a specific platform, it is imperative to gain some understanding with regard to the platform's underlying architecture, along with its security control mechanisms and its security features. In this chapter, we provide a brief overview concerning the architecture underlying the Android platform. We list and describe the multiple components available through the Android Software Stack. We also look into the principal components that are typically available in Android applications, and we highlight the operating system's main security controls and security features. Furthermore, we examine some of the existing research relating to the security features available within the Android operating system, and we present a literature review.

2.2 Android Software Stack

The Android Software Stack, shown in figure 1 below, is composed of the following components [11]:

1. **Linux Kernel**: The base of the Android OS.
2. **Hardware Abstraction Layer (HAL)**: An interface that connects the Java API Framework with the hardware devices.
3. **Android Runtime**:

3.1. **Android runtime (ART)**: As defined in [12], it is "*the managed runtime used by applications and some system services on Android*". ART replaced the Dalvik Android runtime upon the release of Android 5.0 (Android Lollipop).

3.2. **Core Libraries**: These include Java runtime libraries, Java-based libraries specific for Android, as well as libraries specific for the Dalvik virtual machine (for the ART) [13].

4. **Native C/C++ Libraries**: Some of the code in the base Android operating system is written using C/C++ libraries. These libraries include Webkit, OpenMAX AL, OpenGL ES, and many more.
5. **Java API Framework**: A set of APIs written in Java, that provide all the tools necessary for building Android applications. The framework encloses the content providers, the view system, as well as managers for activities, resources, and many other elements**.**
6. **System Applications**: These are core applications that are typically pre-installed on user devices. System applications generally include an application for making phone calls, for text messaging, for sending emails, for organizing calendar events, etc.

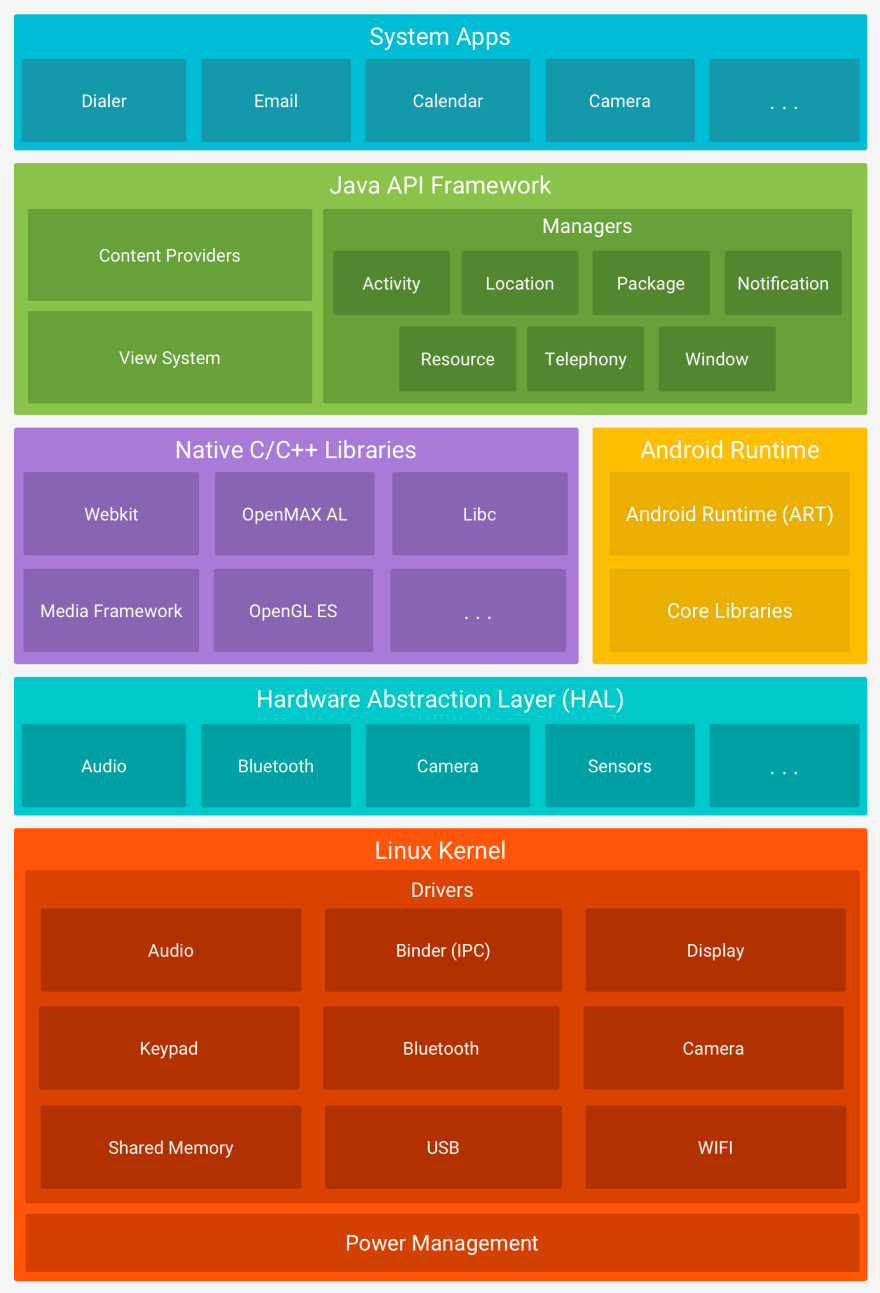

Figure 1 - Android Software Stack [11]

In Android, multi-layered security is utilized with the multiple components within the Android Software Stack, shown in figure 1. Essentially, as explained in [14], components in the Software Stack assume and rely on the fact that the components below them are properly secured.

For example, the Applications component assumes that the Android Framework component is properly secured. The Android Framework component assumes that the Native Libraries and the Android Runtime are properly secured, and so on. This security dependency between the different components implies that the security has to be carefully and well implemented at each layer. Proper security is more crucial at the lower layers, but it is still important at every layer nonetheless.

2.3 Principal Application Components

2.3.1 Introduction

Android applications typically consist of a set of components put together to achieve certain functionalities, along with a set of resources (e.g. images, audio files), and an accompanying AndroidManifest.xml file. In this section, we provide a brief overview of the principal components that generally constitute an Android application. In this research, we utilize most of these components in the proof-of-concept applications that we built for Chapter 4 and Chapter 5.

2.3.2 AndroidManifest.xml

The Android manifest file is a file that consists of XML tags, which essentially describe to the operating system the structure of the components in the corresponding application. Figure 2 displays an example of what an AndroidManifest.xml file may look like. The components generally include activities, services, broadcast receivers, and sometimes custom content providers.

The Android manifest file also declares the application's unique package name, the application's title, its minimum API level, as well as all the permissions that the application may potentially need to access.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.mikewakim.privilegedapplicationthesis">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="App #1 (Privileged)"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".services.GetCurrentLocationService"/>
        <receiver android:name=".broadcasts.MessageBroadcastReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="com.mikewakim.privilegedapplicationthesis.sendLocationData" />
            </intent-filter>
        </receiver>
    </application>

</manifest>
```

Figure 2 - Example of an AndroidManifest.xml File

2.3.3 Activities

An activity, as defined in [15], is "*an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map*". Activities are fundamental when it comes to building Android applications. In general, there's an activity associated with each screen a user interacts with on an Android application. These typically hook up to declared resource files, such as XML layout files, PNG images, etc. One very important aspect of activities is their lifecycle, shown in figure 3. It is good

to understand the different states an activity can be in, as it helps developers choose at what point certain instances of code need to run, and so on.

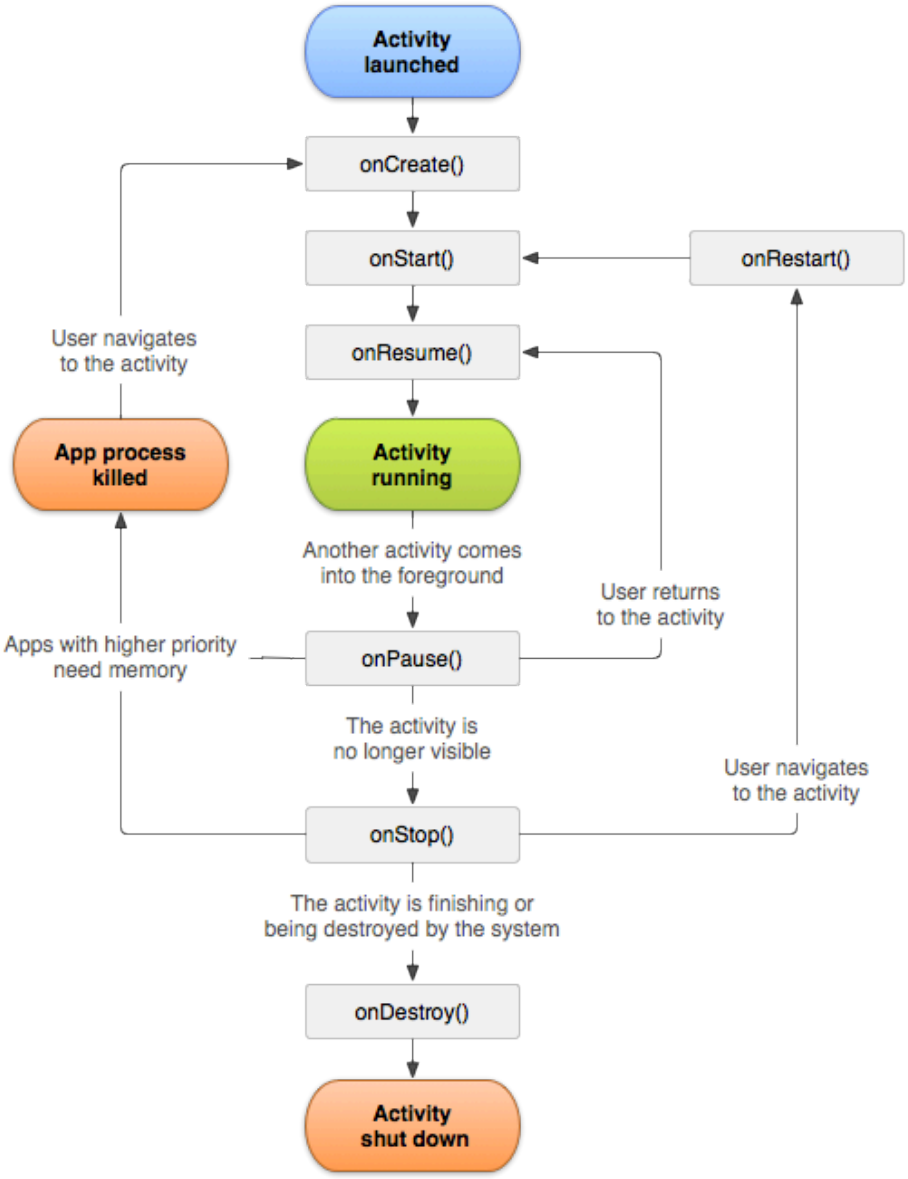

Figure 3 - Android Activity Lifecycle [16]

2.3.4 Services

A service in Android, as defined in [17], is "*an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use*". It runs on the main thread of the process to which it belongs, and it essentially has two main features [17]:

1. It is used to inform the system when a background task is required, even if an activity that started it is paused or terminated.
2. It can be used for providing some functionality to other applications. This is mainly done using the Context.bindService() function.

There are other options besides a service to run background tasks; for example, the AsyncTask class can be very useful when we need to perform background tasks for which the outcome affects our user interface in some way [18].

2.3.5 Broadcast Receivers

Broadcast receivers are very common components in Android applications: they represent "listeners" for specific types of actions [19]. For instance, if we have an application that wants to perform a certain functionality when a new incoming text message arrives to a user's device, that application would register a broadcast receiver that would listen to the action that is referred to as "Telephony.SMS_RECEIVED". Broadcast receivers may be declared statically in an AndroidManifest.xml file, or dynamically during an application's run-time. Figure 4 shows an example of what a static declaration clause of a broadcast receiver may look like.

```
<receiver android:name=".receivers.IncomingMessageReceiver">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
    </intent-filter>
```

Figure 4 - Broadcast Receiver Declaration Example

Broadcast receivers are not limited to actions that relate to resources available through the Android system, such as incoming or outgoing text messages, new calendar events, incoming or outgoing phone calls, etc. They can actually be set up for custom "actions" that applications may create. For example, if we have two applications, application A and application B, application B may create a broadcast receiver for an action that it chooses, for instance "sendData". Thereafter, whenever application A needs to send certain information to application B, application A would send a broadcast, specifying application B's package name, and specifying application B's custom action. The broadcast receiver would then amass the arriving data and perform whichever functionality its corresponding application may require. Broadcasts can be sent to multiple applications at once if needed.

2.3.6 Content Providers

As explained in the official Android documentation available in [20, 21], a content provider is a component intended to manage the access to certain data sets available in an application. For instance, in order for an application A to share data with an application B, application A may create a content provider for the data that it wants to share. Thereafter, when application B wants to access that data, application B would access application A's content provider. This concept is illustrated in figure 5 below.

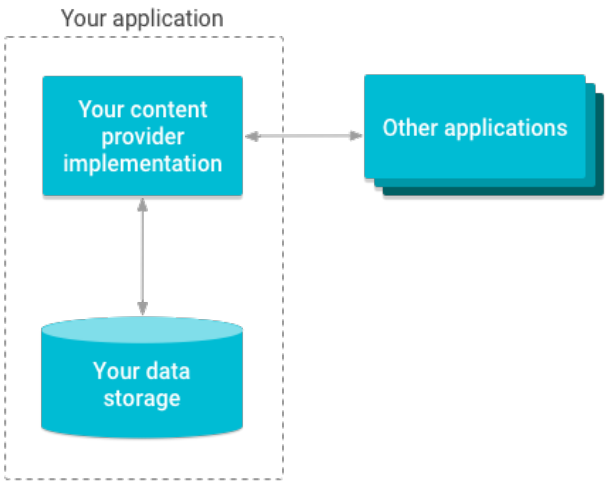

Figure 5 - Concept of Content Providers [21]

under a Creative Commons Attribution 2.5 Generic https://creativecommons.org/licenses/by/2.5/

Pre-installed native applications often utilize content providers to allow applications that are granted the appropriate permissions to access their data sets. For instance, there exists a content provider for calendar related data, for contacts stored on a user's device, for SMS messages, etc.

2.3.7 Intents

Activities, services, and broadcast receivers in Android are generally created in the same fashion. They all utilize the element known as "Intent", which also plays a fundamental role in the development of Android applications. [22] defines an intent as an "*abstract description of an operation to be performed*". Intents in Android can be used to create as well as interact with activities, services, and broadcast receivers. For instance, if we have two activities, activity A and activity B, in order to start activity A, we would typically write code that looks like the following (Java code):

```
Intent i = new Intent(ActivityA.this, ActivityB.class);
i.putExtra("variable_name", value);
startActivity(i);
```

The code above essentially does three things:

1. Starting from activity A, it creates an "intent" to start activity B.
2. It passes information to activity B through the use of "extras". Extras require a variable name as well as a value associated with that variable name. The values can be of primitive types, or of custom objects that are serializable in some way.
3. It makes a function call to start activity B, using the newly created intent as input.

The same type of code applies to starting new services, as well as sending or receiving broadcasts. It is important to note that there are two types of intents: (1) explicit intents and (2) implicit intents. Explicit intents are ones that specify the exact class name of the intended recipient. On the other hand, implicit intents do not specify the class name of the recipient; instead they specify an action, and any application that can handle that action may pick up that intent [23]. We demonstrate how we put to use all this functionality in Chapters 4 and 5.

2.4 Principal Security Features

2.4.1 Introduction

The Android operating system is built on top of the Linux Kernel partly due to the fact that the Linux Kernel has undergone a lot of security attacks and a lot of research since its creation, with the involvement of thousands of developers, and because it is highly modifiable and considered very stable and secure. Android puts to use multiple security features offered by the

Linux kernel, these primarily include a user-based permissions model, the concept of process isolation, and mechanisms for inter-process communication [24]. In addition to these security features, there are various utilities that developers can take advantage of to increase applications' overall security. For instance, there are utilities for securely storing cryptographic keys (e.g. Android KeyStore System), for full-disk encryption, for authentication, for application signing, etc. In this section, we describe some of these security features. In Chapter 4, we outline a negative effect that the inter-process communication has on the security imposed by the user-based permissions model. In Chapter 5, we explain how we apply the mechanisms of inter-process communication in our proposed end-to-end encrypted messaging model. And finally, in Appendix B, we demonstrate how some of these security features can be put together in our end-to-end encrypted messaging application proof-of-concept.

2.4.2 User-Based Permissions Model and Process Isolation

The user-based permissions model that is inherited from the Linux kernel is one of the most important security features available in the Android operating system. In Linux, there are basically three user-based permission groups and three permission types. The permission groups, as defined in [25], are the following:

1. *"**Owner**: The Owner permissions apply only the owner of the file or directory; they will not impact the actions of other users."*
2. *"**Group**: The Group permissions apply only to the group that has been assigned to the file or directory; they will not affect the actions of other users."*
3. *"**All Users**: The All Users permissions apply to all other users on the system; this is the permission group that you want to watch the most."*

On the other hand, the three permission types, as defined in [25], are the following:

1. *"**Read**: The Read permission refers to a user's capability to read the contents of a file."*
2. *"**Write**: The Write permissions refer to a user's capability to write or modify a file or directory."*
3. *"**Execute**: The Execute permission affects a user's capability to execute a file or view the contents of a directory."*

Basically, a user may perform "read", "write", or "execute" operations on a certain file only if he has the required permissions to do so. When it comes to Android, in a similar fashion to the Linux operating system, a unique user ID (UID) is attached to each individual application that is installed on an Android device. The operating system is built in such a way that each application runs on its own separate process as a "user". This is exceptionally important, as it brings into being the concept of the Android application sandbox. The Android application sandbox, as explained in [24], makes it so that applications by default cannot interact with each other, unless they have the proper "user" privileges. Moreover, each application has its own private process, along with its own (optional) internal storage space. Accordingly, the operating system was built in such a manner that if an application wants to access sensitive data outside of its sandbox, that application must request the permission from the user of the device to do so [26].

The Android permissions model controls who has access to certain content providers and resources available on an Android device. The model accomplishes that by securing protected APIs through the usage of "permissions". These permissions are essentially split into three main categories: normal, dangerous and custom permissions. Normal permissions manage data that pose "*very little risk to the user's privacy or the operation of other apps*" [26]. These include permissions such as "INTERNET", "ACCESS_WIFI_STATE", "SET_WALLPAPER",

"SET_TIMEZONE", and many more [27]. Such permissions are automatically granted by the operating system when declared in an application's manifest file (i.e. AndroidManifest.xml). Dangerous permissions, on the other hand, manage data or resources that "*involve the user's private information, or could potentially affect the user's stored data or the operation of other apps*" [26]. These include permissions such as "READ_CALENDAR", "READ_CONTACTS", "READ_CALL_LOG", "SEND_SMS", "READ_SMS", etc. [26]. When an application declares dangerous permissions in its manifest file, a user must explicitly approve and grant these permissions to the application. It is worth noting that dangerous permissions are often grouped together in "Permission Groups". Table 1 below lists all the dangerous permission groups and their associated permissions. Finally, custom permissions are permissions that are created by application developers. These are typically used for securing functionality available through the inter-process communication mechanisms.

Table 1 - Dangerous Permissions, data from [26]

Permission Group	**Permissions**
CALENDAR	• READ_CALENDAR • WRITE_CALENDAR
CAMERA	• CAMERA
CONTACTS	• READ_CONTACTS • WRITE_CONTACTS • GET_ACCOUNTS
LOCATION	• ACCESS_FINE_LOCATION • ACCESS_COARSE_LOCATION
MICROPHONE	• RECORD_AUDIO
PHONE	• READ_PHONE_STATE • CALL_PHONE • READ_CALL_LOG • WRITE_CALL_LOG • ADD_VOICEMAIL

Permission Group	Permissions
	• USE_SIP • PROCESS_OUTGOING_CALLS
SENSORS	• BODY_SENSORS
SMS	• SEND_SMS • RECEIVE_SMS • READ_SMS • RECEIVE_WAP_PUSH • RECEIVE_MMS
STORAGE	• READ_EXTERNAL_STORAGE • WRITE_EXTERNAL_STORAGE

The permission model has been updated a few times with the releases of new versions of the operating system. It used to be that users had to accept all the permissions required by an application at the time of installation. However, as of Android 6.0 (Android Marshmallow), this mechanism changed. Instead of having to accept all permissions at installation, applications got the capability of requesting permissions at run-time, whenever they need access to certain data, and the user can accept or refuse the data access [28]. It is important to note that regardless of whether permissions are set at installation time or at run-time, the permissions that an application seeks to obtain must always be listed in the application's AndroidManifest.xml file.

2.4.3 Inter-Process Communication (IPC)

We have mentioned in the previous section that the Android application sandbox makes it so that applications by default cannot interact with each other. However, when it comes to data sharing, applications clearly need to have the means to communicate with each other. Android provides several mechanisms for inter-process communication; these mechanisms primarily include the following [29]:

1. **Intents**: We have already explained the concept of intents in section 2.3.7. When it comes to IPC, intents are typically utilized to send a "broadcast" from one application to another. This is generally done via the sendBroadcast() or sendOrderedBroadcast() methods. In this thesis, we refer to such intents as "broadcast sender intents".
2. **Services**: We have also already explained the main idea behind services in section 2.3.4. Services in Android have multiple usages, one of which is to supply functionalities for other applications to use. For instance, if we had application A and application B, and application A wanted to trigger some functionality in application B, application B can supply a service for that functionality, and application A would trigger that service whenever it needs.
3. **Binder and Messenger Interfaces**: Binder and Messenger are two classes that can be utilized to achieve IPC [29]. These two classes are generally more effective than the intents and services in terms of latency and synchronization, however they are much more complex to use.
4. **Broadcast Receivers**: We explained the concept of broadcast receivers in section 2.3.5. Basically, in order for an application A to listen to messages from another application B, application A can set up a broadcast receiver which listens to broadcast sender intents. When application B sends a broadcast sender intent, applications A's broadcast receiver can pick up the event and trigger any required functionality.

2.4.4 Android KeyStore System

When it comes to security-related applications, securely managing (e.g. storing, extracting) cryptographic keys is not a simple task. The Android operating system provides a key storing system that offers application developers the means to securely store cryptographic keys on a

user's device. The Android KeyStore system has two principal security features: (1) "*Extraction Prevention*" and (2) "*Key Use Authorizations*" [30].

In terms of extraction prevention, the key storing system makes it so that a utilized key never enters an application's process. Basically, instead of performing cryptographic computations on the application's process, these computations get executed on a system process. [30] explains that as a result of this, if an application's process were to be compromised, an attacker would be capable of using the applicable key; however, that attacker will never be able to extract that key. [30] also explains that the key material can be stored on secure hardware, such as the Secure Element (SE) or Trusted Execution Environment (TEE).

In terms of key use authorizations, the Android KeyStore system requires that applications define the authorized uses of their corresponding keys. This authorization is set upon the addition of new keys to the key store system, whether it be through key generation or through importing, and it is unmodifiable. [30] explains that there are three categories for key use authorizations: (1) "*cryptography*", (2) "*temporal validity interval*", and (3) "*user authentication*". The "*cryptography*" key use authorizations specify how the key can be used (i.e. with which algorithms, for what operations, etc.). The "*temporal validity interval*" key use authorizations specify a time interval for when the key can be used. Finally, the "*user authentication*" key use authorizations require that a user is authenticated using the lock screen credentials (e.g. password, fingerprint) in order to allow him to use the corresponding stored keys [30].

2.4.5 Encryption

The Android operating system offers its users two types of device encryption: (1) full-disk encryption (for Android 4.4, 5.0 and up) and (2) file-based encryption (for Android 7.0 and up) [31, 32]. Both mechanisms put to use cryptographic primitives to encrypt user data available on a user's device. The full-disk encryption essentially encrypts all the available user data by applying one master key (AES, 128 bits or more) [32]. On the other hand, the file-based encryption separately encrypts files available on a user's device, using different keys for different files. As a result, files can be unlocked independently [33]. More details regarding full-disk encryption and file-based encryption can be found in [32] and [33], respectively.

2.4.6 Authentication

There are two principal authentication components supported by the Android operating system: (1) Gatekeeper and (2) Fingerprint. As explained in [34], the Gatekeeper component deals with PIN, pattern, and password authentication, while the Fingerprint component strictly deals with fingerprint authentication. More details regarding Gatekeeper and Fingerprint authentication can be found in [35] and [36], respectively.

2.4.7 Application Signing

The Google Play store requires that developers sign their Android Application Packages (APKs) prior to uploading them. Accordingly, when a user downloads an application on his

Android device, the Package Manager on his device would extract the corresponding certificate from the APK and verify the signature. If the signature is valid, the installation proceeds as intended. Otherwise, the application gets rejected. This concept of application signing is very important. As mentioned in [37]: "*On Google Play, application signing bridges the trust Google has with the developer and the trust the developer has with their application. Developers know their application is provided, unmodified, to the Android device; and developers can be held accountable for behavior of their application.*"

2.5 Literature Review

2.5.1 Introduction

There are various research papers that relate to the security features available in the Android operating system. Some of these papers go over the entire security underlying the operating system, while others focus on specific security features. In fact, some security features have attracted the attention of researchers more than others; a good example of that would be the user-based permissions model. Due to its importance when it comes to securing users' personal information on their mobile devices, the Android permission model and the security that is associated with it have been the main subject of various research papers. Some papers focus on the usability aspect of the permission model [38], while others study the coarse-granularity of permissions and over-privileged applications [39, 40]. Moreover, some papers look into privilege escalation issues [41, 42, 43, 44] and some analyze the security [45] that is associated with the Android IPC mechanisms and IPC channels. In this section, we review some of the existing work

relating the security of the Android operating system. We aim our attention on papers that have to do with the Android permission model and with the inter-process communication mechanisms.

2.5.2 Usability Aspect of the Android Permission Model

One of the main studies that focused on evaluating the usability aspect of the Android permission model is the research paper [38], wherein Felt et al. analyze how effective the Android permission model is when it comes to warning users about the extent to which newly installed applications may be invading their privacy. To do so, the authors implemented two usability studies, one that involved 308 Android users through an Internet survey, and another one that involved 25 Android users in a laboratory study.

Their Internet survey aimed to discover the level of understanding users have when it comes to Android permissions. It consisted of nine different pages that were meant to be completed on an Android device. A few of the questions had to do with the users' usage statistics, e.g. how long have they had an Android phone for, while the rest had to do with specific permissions, such as the READ_SMS, READ_CALENDAR or CAMERA permissions and so on. On the other hand, their laboratory study consisted of a 30 to 60-minute interview with each user, in which they studied the attention, comprehension, and behavior that users had with regards to the permission model.

Upon analyzing their results, Felt et al. concluded that the permission model failed to properly inform the majority of the survey participants about the resources that applications are

indeed capable of accessing with their requested permissions. They mentioned however that a small set of the participants proved to have good awareness and understanding of the permissions. In fact, the permission model did help some users stay away from applications that could potentially invade their privacy. The authors also concluded that since the statistics that they got with regards to user attention and comprehension of the permission model were at fairly low rates, the model could use some enhancements in order to become more accessible [38]. Moreover, Felt et al. also investigated some potential reasons that caused the permissions' details to be somewhat vague. For example, they suggested that the category headings were confusing, and that better headings could lead to less ambiguity. Based on their findings, they recommended a few actions that could be performed to address some of these discovered issues.

The issues addressed in this paper are of utmost importance, due to the fact that user attention and comprehension are key for helping users preserve their privacy. The experiments that were conducted by the authors were implemented on Android 2.2 devices. It is important to note that since this paper was published in 2012, the Android operating system has undergone several updates that affected the permission model. For instance, as of Android 6.0 (Android Marshmallow), the permission model no longer forced users to accept the permissions requested by an application at installation time. On demand run-time permissions were implemented, in which applications could request permissions when they actually need them.

2.5.3 Coarse-Granularity of Permissions and Over-Privileged Applications

One of the main problems that affect the Android permission model is the concept of applications being over-privileged. Over-privileged applications are applications that have access to more data than they actually require. Sometimes this is done intentionally by application developers, e.g. by requesting permissions that they don't necessarily need; however, other times, this is a result of the granularity and sometimes ambiguity corresponding to the permissions available through the permission model. In terms of granularity, depending on the context, some of the Android permissions can grant more privileges to an application than the application actually requires. For example, the READ_SMS permission is one that allows an application to access all available SMS messages on a user's device. In some cases, applications may actually need to access all this data. For example, a 3rd party text messaging application that acts as an alternative to the native text messaging application undoubtedly would need the data provided through the permission. On the other hand, another application may need this permission for a one-time use, in which it may require to read one specific text message. An example of this would be an application that wants to validate a user's phone number. It would typically send a PIN to a user via an SMS/MMS message, and the user would input the received PIN into the corresponding application to prove that he indeed owns that phone number. Instead of requiring the user to input that PIN manually, some of these applications request the permission READ_SMS, wait for the incoming text message that holds the PIN, and thereafter validate the user's phone number by automatically inputting that PIN. In this case, these applications are gaining access to all the text messages available on a user's device, simply to read one of them, for a specific functionality.

There are certain research papers wherein the authors focused on detecting and informing the user about applications that may be over-privileged on their device [39, 40]. In [39], Felt et al.

developed a tool known as Stowaway, which performs a static analysis in regards to what API calls an application makes, and which also maps the API calls that are made to the permissions required to make them. Upon applying the tool to 940 Android applications, their results demonstrated that one-third of these applications were over-privileged. In [40], Au et al. developed a tool known as PScout, which also performs a static analysis that relates API calls with permissions; however, for this tool, the permission specification is extracted directly from the Android operating system, as opposed to using a published specification. These tools were developed prior to the update that the Android permission model underwent to allow for run-time permission granting; however, since the permissions must be declared in the AndroidManifest.xml file regardless, the results indicated by these tools can be extremely useful.

2.5.4 Privilege Escalation

In section 2.4.2, we explained that the Android operating system is built in such a way that each application has its own sandbox. Essentially, in order for an application to interact with other applications or to access certain protected resources, that application must have the appropriate privileges. This is where the user-based permission model comes into play. In the research paper [41], Davi et al. explained that although the Android permission model has a good, well-structured permission system, the model fails to protect users from privilege escalation attacks. Basically, they defined that a privilege escalation attack occurs when "*an application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee)*" [41]. The attack was described by Davi et al. through a scenario that

implicates three applications: application A, application B, and application C. These applications are designed such that:

- Application A has no granted permissions. It contains two components: C_{A1} and C_{A2}. These components are not protected by any permissions.
- Application B has a granted permission, p_1. It contains two components: C_{B1} and C_{B2}. These components are not protected by any permissions.
- Application C has no granted permissions. It contains two components: C_{C1} and C_{C2}. The C_{C1} component requires the permission p_1, and the C_{C2} component requires the permission p_2.

Given this structure, components C_{B1} and C_{B2} in application B are expected to have access to component C_{C1} in application C. Components C_{A1} and C_{A2} are not expected to have access to any of the components in application C as they lack the required permissions. However, due to the fact that the components in application B do not require any permissions, application A can access components in application B, which in turn can access components in application C. In other words, the non-privileged application A is capable of accessing protected components available through the privileged application B. In their paper [41], Davi et al. also described a proof of concept wherein they demonstrated a privileged escalation attack. In their attack, they exploited an at the time known vulnerability of the Android Scripting Environment (ASE). The exploit allowed them to send SMS messages from a non-privileged application to a premium-rate number.

In [42], Felt et al. explain that systems that associate permissions for each application, and that provide IPC capabilities, may lead to the threat of "permission re-delegation": an issue that occurs "*when an application with a user controlled permission makes an API call on behalf of a*

less privileged application without user involvement" [42]. In their study, the authors consider the permission re-delegation problem as a special instance of the "confused deputy problem", whereby a non-malicious privileged application that has been granted permissions by the user is considered a "deputy". In the interest of demonstrating that permission re-delegation is an important problem that is applicable to real-world applications, Felt et al. conducted a case study in which they surveyed 872 Android applications, of which 16 were core system applications. The surveyed applications were deployed on Android 2.2 devices. Their study demonstrated that approximately 37% of the applications that they surveyed were (1) granted at least one permission by the user, and (2) they provided access to a public component (e.g. service or receiver) [42]. Moreover, Felt et al. discovered vulnerabilities in 5 out of the 16 surveyed system applications. For instance, they were capable of toggling on or off certain settings, such as GPS or Wi-Fi, by sending IPC requests to the main Settings application.

In the interest of preventing permission re-delegation from occurring, Felt et al. proposed and implemented a defence mechanism that they referred to as IPC Inspection [42]. The main idea behind IPC Inspection was to reduce the permissions that correspond to applications that are communicating with each other. Essentially, if an application that is less privileged communicates through IPC with an application that is more privileged, IPC Inspection would reduce the permissions of the more privileged application to that of the less privileged application in order to prevent permission re-delegation. There are a few other research papers in which researchers attempt to tackle the confused deputy problem. For instance, in [43], Dietz et al. present QUIRE: a security mechanism that tracks the full call chain that is associated with IPC instances, and that can add extra security to ongoing IPC by putting to use message authentication codes. On a similar

note, in [44], Bugiel et al. present XManDroid: a security framework that permits run-time monitoring of IPC instances, and that verifies ongoing IPC with accordance to centralized system policy rules.

Privilege escalation issues relating to the Android operating system are extremely important, as they provide means to bypass the principal security features put in place to secure users' data on their mobile devices. Although some research papers suggested countermeasures to protect against the confused deputy problem, the issue of privilege escalation still persists. It is still possible for an unprivileged application to gain access to protected data from a malicious privileged application by employing IPC mechanisms. We will discuss this in detail in Chapter 4.

2.5.5 Security of the Inter-Process Communication Mechanisms

As we have already seen, there are several inter-process communication mechanisms available within the Android operating system. These mechanisms often deal with sensitive information and thus, it is important to examine their security. In the research paper [45], Chin et al. evaluated the security of different inter-process communication techniques, and provided a static analysis tool called ComDroid, which searches for potential vulnerabilities that have to do with IPC channels in Android applications. The authors explained that there are a basically two intent-based attack surfaces: (1) unauthorized intent receipt and (2) intent spoofing.

In terms of unauthorized intent receipt, Chin et al. explained that when an application wants to send a message to another application, there are no guarantees that the message will indeed

reach its intended recipient. This is due to the fact that unless the intent is protected with permissions, a malicious application can intercept the intent in traffic and read all its content. Moreover, malicious applications that intercept such intents may also perform phishing (e.g. by spoofing an expected activity in order to steal user supplied data [45]) or denial of service (i.e. prevent the intent from reaching its correct destination). Chin et al. explained that the interception of intents can be done through broadcast theft, activity hijacking and service hijacking. In terms of intent spoofing, they explained that if an application has an exported component (i.e. a component accessible to other applications) that does not require any permissions, it is possible to send intents to that component in the hope of triggering certain functionalities. Spoofing attacks as such can target broadcast receivers, exported activities, and exported services.

Chin et al. utilized ComDroid to analyze 20 Android applications, and they determined that 12 of the studied applications contained at least one vulnerability [45]. To protect against the unauthorized intent receipt and intent spoofing attacks, Chin et al. explained that application developers should (1) be careful about the usage of implicit intents (i.e. intents without a specific intended recipient) and exported components, (2) utilize explicit intents (i.e. intents with a specific intended recipient) when sending private data from one application to another, and (3) associate strong permissions with the selected intents for good access control [45].

2.6 Conclusion

In this chapter, we provided background information regarding the Android operating system. More specifically, we examined the Android Software Stack, we listed and described the

principal components that typically constitute Android applications, and we listed and described the principal security features that are offered by the operating system. Furthermore, we presented a literature review wherein we focused on research papers that study some of these security features. In our literature review, we focused on papers that examine (1) the usability aspect of the permission model, (2) the coarse-granularity of permissions within the permission model, (3) the issue of privilege escalation, and (4) the security associated with the inter-process communication mechanisms available within the operating system.

In this thesis, we will explain and provide an overview of how some of the concepts that we discussed in this chapter can be applied together to allow an unauthorized application to gain access to protected data by retrieving it from a malicious privileged application on a user's device. Furthermore, we will propose a new architectural design for E2EE messaging applications which applies some of these concepts in the interest of diminishing the common issues that are generally applicable to such applications.

Chapter 3: End-to-End Encrypted (E2EE) Messaging Applications

3.1 Introduction

In this chapter, we first list and describe the security properties that are often desired in end-to-end encrypted (E2EE) messaging applications. We then examine some of the existing research that has to do with evaluating the security of such applications, and we present a literature review. Moreover, we derive and describe four categories of issues that are commonly associated with E2EE messaging applications.

Communication online, whether it be through e-mails, instant messaging, or any other means, can be extremely convenient, cost-effective, and flexible [46]. There are numerous applications available for Android devices, that allow users to communicate over the Internet. These applications may differ greatly in terms of how they are designed and how they are implemented; however, one thing that they share in common, is the fact that they all deal with users' sensitive information to a certain extent. Considering that the majority of the Internet traffic is accessible to anyone with the proper tools, applications often resort to encryption to help them secure the users' data. Essentially, encryption is applied to protect the data in transit between a client and a server. Depending on the utilized scheme, the server may or may not have the means to decrypt the cipher text.

End-to-end encrypted (E2EE) messaging applications have been increasing in popularity over the course of the past few years [8, 47]. These types of applications allow their users to securely communicate with each other in such a way that the exchanged data is inaccessible to anyone except for the senders and receivers of the encrypted messages. Essentially, the data gets encrypted on the sender's device, and only the intended receiver is capable of decrypting it. Applications can offer end-to-end encrypted messaging by employing combinations of cryptographic primitives, which include techniques for authentication, key management, encryption, integrity verification, and digital signatures. There are various applications available on the Google Play store that provide end-to-end encryption to their users; table 2 below lists some of the most popular ones.

Table 2 - Most popular Android messaging applications that support E2EE

Application Name	**Number of Installs** *Note that these numbers only account for installs from the Google Play Store; they do not account for installs from other marketplaces.*
WhatsApp	1 to 5 billion [48]
Messenger *(Secret Conversations)*	1 to 5 billion [49]
Telegram	100 to 500 million [50]
Google Allo *(Incognito Mode)*	10 to 50 million [51]
Signal Private Messenger	1 to 5 million [52]
Wickr Me – Secure Messenger	1 to 5 million [53]
ChatSecure	500 thousand to 1 million [54]

3.2 Desired Security Properties

There are multiple security properties that end-to-end encrypted messaging applications often provide. In this section, we list some of these security properties and provide a brief description for each one.

- **Confidentiality:** Confidentiality is one of the most important properties that users typically seek in security related applications. It is defined by [55] as "*the ability to hide information from those people unauthorised to view it*". In other words, when we want to protect the confidentiality of users' data, we want to ensure that that data cannot be disclosed to any unauthorized parties [56]. Typically, we apply access control and encryption techniques to prevent the loss of confidentiality [57].
- **Integrity:** Data integrity is another property that is crucial in most secure applications. Data integrity is defined in [58] as a "*property whereby data has not been altered in an unauthorized manner since it was created, transmitted or stored*". Generally, integrity is ensured by employing cryptographic hash functions along with digital signatures and/or message authentication codes.
- **Availability:** The availability property has to do with ensuring that the data and the systems required in a security application are reliable and operational whenever they are needed. Protection against loss of availability is typically done by having fault tolerant systems, redundancies, and backups [57].
- **Authenticity:** Authenticity is defined in [59] as "*the property that ensures that the identity of a subject or resource is the identity claimed*". Authentication may be required by applications at multiple different levels (e.g. at the operating system, at the network, at the application, etc.).

There are various user authentication techniques that applications can put to use, the most common techniques include biometrics (e.g. fingerprint), text-based passwords, graphical passwords, digital signatures, etc. Authentication techniques can be combined to achieve multi-factor authentication for increased security.

- **Forward Secrecy:** Forward secrecy, sometimes referred to as perfect forward secrecy (PFS), is a property which assures that the compromise of long-term private keys does not lead to the compromise of previous sessions in which session keys were utilized [60].
- **Future Security:** The future security property was specified by the authors of the TextSecure application, an end-to-end encrypted messaging application that we will be discussing in the upcoming section 3.3.3.1. Essentially, as explained in [61, 62], the future security property guarantees that if long-term keys remain secret, and a compromise of short-term keys occurs, then the applicable protocol will eventually self-heal and future messages will stay secure.
- **Deniability:** Deniability is a property that can be associated with authentication or encryption techniques. In encryption techniques, the deniability property essentially allows users to deny their capability of encrypting or decrypting a certain piece of data [63]. On the other hand, in authentication techniques, the deniability property refers to the scenario in which the participants involved in a specific setup can authenticate the messages exchanged between each other, however a participant cannot convince a third party that these messages were indeed associated with another participant.

3.3 Existing Work

3.3.1 Introduction

Although end-to-end encrypted messaging applications aim to help users secure their data and preserve their privacy, the security offered by these applications typically depends on the design and implementation details selected by the developers. Due to the fact that there is a wide range of applications that claim to offer end-to-end encrypted messaging, it is extremely important to verify and validate the security of these applications. The Electronic Frontier Foundation (EFF) published a scorecard [64] in which they evaluated existing secure messaging applications and tools in accordance with a set of seven main criteria. The authors emphasized in [65] that although the criteria that they examined were necessary for a tool to be secure, the selected criteria were not sufficient to guarantee the tool's security, and as a result, more detailed analyses are required. Nevertheless, the results that were published in the scorecard were very interesting and extremely important. In other respects, several researchers targeted end-to-end encryption applications, whether it be for the Android or other operating systems. Some of the research papers focused on analyzing the protocols utilized by specific applications, and on determining whether these applications indeed offer the security properties that they promise [61, 66]. Other research papers focused on performing and publishing the results of forensic analyses with regard to specific applications, in the interest of searching for potential bugs or exploits [67, 68]. In sections 3.3.2 and 3.3.3, we will discuss the EFF's Secure Messaging Scorecard, and we will go over some of the existing literature regarding end-to-end encrypted messaging applications, respectively.

3.3.2 EFF's Secure Messaging Scorecard

The seven criteria selected by the EFF for the scorecard were the following [64]:

Criterion #1: "*Is your communication encrypted in transit?*"
Criterion #2: "*Is your communication encrypted with a key the provider doesn't have access to?*"
Criterion #3: "*Can you independently verify your correspondent's identity?*"
Criterion #4: "*Are past communications secure if your keys are stolen?*"
Criterion #5: "*Is the code open to independent review?*"
Criterion #6: "*Is the crypto design well-documented?*"
Criterion #7: "*Has there been an independent security audit?*"

The EFF analyzed various tools and applications according to these seven criteria. Some of these projects were built for an individual operating system (e.g. Mac OS X, Windows, Android or iOS), others were built for multiple platforms. The results that were published in the scorecard are shown in table 3 below. It is important to note that the scorecard was initially launched on November 6th, 2014. Some of the results were updated by the EFF; according to the Changelog available in [64], the latest update occurred on April 5th, 2016.

Table 3 - Results of EFF scorecard 1.0, data from [64]

Application/Tool Name	Criterion						
	#1	#2	#3	#4	#5	#6	#7
AIM	✓	☒	☒	☒	☒	☒	☒
BlackBerry Messenger	✓	☒	☒	☒	☒	☒	☒
BlackBerry Protected	✓	✓	✓	☒	☒	✓	✓
ChatSecure + Orbot	✓	✓	✓	✓	✓	✓	✓

Application/Tool Name	Criterion						
	#1	#2	#3	#4	#5	#6	#7
Ebuddy XMS	✓	☒	☒	☒	☒	☒	☒
Facebook chat	✓	☒	☒	☒	☒	☒	✓
FaceTime	✓	✓	☒	✓	☒	✓	✓
Google Hangouts/Chat “off the record”	✓	☒	☒	☒	☒	☒	✓
Hushmail	✓	☒	☒	☒	☒	☒	☒
iMessage	✓	✓	☒	✓	☒	✓	✓
iPGMail	✓	✓	✓	☒	☒	✓	☒
Jitsi + Ostel	✓	✓	✓	✓	✓	✓	☒
Kik Messenger	✓	☒	☒	☒	☒	☒	☒
Mailvelope	✓	✓	✓	☒	✓	✓	✓
Mxit	☒	☒	☒	☒	☒	☒	☒
Off-The-Record Messaging for Mac (Adium)	✓	✓	✓	✓	✓	✓	☒
Off-The-Record Messaging for Windows (Pidgin)	✓	✓	✓	✓	✓	✓	✓
PGP for Mac (GPGTools)	✓	✓	✓	☒	✓	✓	☒
PGP for Windows Gpg4win	✓	✓	✓	☒	✓	✓	☒
QQ	✓	☒	☒	☒	☒	☒	✓

Application/Tool Name	Criterion						
	#1	#2	#3	#4	#5	#6	#7
RetroShare	✓	✓	✓	✓	✓	✓	☒
Signal / RedPhone	✓	✓	✓	✓	✓	✓	✓
Silent Phone	✓	✓	✓	✓	✓	✓	✓
Silent Text	✓	✓	✓	✓	✓	✓	✓
Skype	✓	☒	☒	☒	☒	☒	☒
SnapChat	✓	☒	☒	☒	☒	☒	✓
StartMail	✓	☒	✓	☒	☒	✓	☒
SureSpot	✓	✓	✓	☒	✓	✓	☒
Telegram	✓	☒	☒	☒	✓	✓	✓
Telegram (secret chats)	✓	✓	✓	✓	✓	✓	✓
TextSecure	✓	✓	✓	✓	✓	✓	✓
Threema	✓	✓	✓	✓	☒	✓	✓
Viber	✓	☒	☒	☒	☒	☒	✓
Virtru	✓	☒	☒	☒	☒	✓	✓
WhatsApp	✓	✓	✓	✓	☒	✓	✓
Wickr	✓	✓	✓	✓	☒	☒	✓
Yahoo! Messenger	✓	☒	☒	☒	☒	☒	☒

From table 3 above, we can see that not all the applications the EFF listed on their scorecard conformed with the criteria that they selected. Essentially, 36 out of the 37 applications listed in the scorecard indeed encrypted the data communications in transit. Out of these applications, only 21 applications encrypted the communications with a key that the provider did not have access to. Basically, these 21 applications are the ones that support end-to-end encrypted communications. When it comes to criterion #3, 20 out of the 37 applications provided means for their users to independently verify their correspondent's identity. In other words, these applications offered their users a way to verify the identities of the parties that they are communicating with. Only 15 out of the 37 applications conformed with the criterion that required that past communications be secure if private keys were stolen. This criterion is essentially the same thing as the "forward secrecy" security property that we defined in section 3.2. With regard to the code being open to independent review, only 15 out of the 37 applications satisfied criterion #5. This criterion is extremely important due to the fact that independent review of code can help detect bugs, back doors, and other problems that may have to do with the structure of an application [64]. On the other hand, 23 out of the 37 applications provided detailed documentation and explanations in regards to how the cryptography-related techniques were applied. Finally, for criterion #7, the results from [64] state that 23 of the 37 applications have had an independent security audit. A summary of the results is shown in table 4 below.

Table 4 - Summary of scorecard results, data adapted from [64]

#	Criterion	# of applications that satisfy the criterion *(out of 37)*
1	"Is your communication encrypted in transit?"	**36** (~ 97%)
2	"Is your communication encrypted with a key the provider doesn't have access to?"	**21** (~57 %)
3	"Can you independently verify your correspondent's identity?"	**20** (~ 54%)
4	"Are past communications secure if your keys are stolen"?	**15** (~ 41%)
5	"Is the code open to independent review?"	**15** (~ 41%)
6	"Is the crypto design well-documented?"	**23** (~ 62%)
7	"Has there been an independent security audit?"	**21** (~ 57%)

3.3.3 Literature Review

3.3.3.1 Security Analysis of Existing Android E2EE Messaging Applications

In the paper [61], Frosch et al. explore and analyze the security of one of the most popular secure messaging applications that used to be available for mobile devices, that is known as TextSecure. TextSecure is one of the many products developed by OpenWhisperSystems, and it is an application that was built to support encrypted end-to-end messaging for mobile devices, whether it be over the usage of SMS or data channels. TextSecure was replaced by Signal Private Messenger, a new application that implements the same protocol used by it for encrypted end-to-end messaging. In [61], Frosch et al. reveal the details regarding the protocol utilized by TextSecure for secure push-messaging. They discuss the building blocks upon which the main Protocol is based. They also provide a rundown in which they list some of the flaws that they discovered relating to the protocol. Furthermore, they examine the extent to which the TextSecure application satisfies the properties that it claims to have, which consist of end-to-end security, deniability, perfect forward secrecy, and future security [61]. We defined all of these properties in section 3.2.

The TextSecure protocol essentially utilizes cryptographic primitives to permit end-to-end encryption between different users. As described in [61], these cryptographic primitives include the Curve25519 algorithm, for Elliptic curve Diffie-Hellman, along with AES (counter mode and cipher block chaining mode) for symmetric encryption. It also takes advantage of HMAC-SHA256 for integrity verification purposes. The application has its own central server, which it employs along with the Google Cloud Messaging (GCM) utility in order to relay messages and manage users' public keys. A typical end-to-end encrypted message exchange between two users through the data channel would employ both the central server and the GCM.

Upon analyzing the protocol, Frosch et al. discovered a couple of issues. One of these issues had to do with the password-based key registration. For instance, they were capable of retrieving an unencrypted password utilized for key registration and sending messages. They did so by simply calling a certain export function. Another one of these issues involved an Unknown Key-Share Attack (UKS), an attack in which Eve intervenes with Alice and Bob's key sharing, in such a way that Alice would indeed share a key with Bob, however Bob would believe that the key is actually shared with Eve instead of Alice [69]. Aside from the issues, Frosch et al. determined that if we assume that the key registration part is properly secured, the TextSecure application indeed meets the security goals that it claims to have in terms of the end-to-end encryption, deniability, perfect forward secrecy, and future security properties.

In the case of the TextSecure application, the fact that the code was open for independent review led to the discovery of issues that were otherwise undetected by the TextSecure team. After auditing and analyzing the security offered by this application, Frosch et al. informed the TextSecure developers about the issues that they discovered within the TextSecure protocol, and they only published the results that they found after the security issues were dealt with.

ChatSecure is another end-to-end encrypted messaging application that has over 500,000 downloads on the Google Play Store [54]. It is an open-source application that utilizes the Off-The-Record messaging system to achieve end-to-end encryption with the properties of authentication, deniability, and perfect forward secrecy. In addition to that, ChatSecure utilizes the SQLCipher library in order to encrypt database files; it also takes advantage of the IOCipher library to benefit from the virtual encrypted disks for secure storage. In the paper [67], Anglano et al.

perform a digital forensic analysis regarding the Android version of the application. Their investigative scenario consisted of a seized Android smartphone, presumably not password locked, and their objective was to analyze the multiple artifacts (e.g. metadata, files, databases) that were generated by ChatSecure in the interest of evaluating the security offered by the application.

Upon analyzing the open-source code along with the artifacts associated with the application, Anglano et al. discovered that ChatSecure stores data (regarding accounts, exchanged messages, contacts, etc.) in multiple encrypted databases. They also discovered that the key utilized to encrypt/decrypt these databases is stored in the internal memory, encrypted using a tool known as CacheWord [70]. The key that was utilized to encrypt/decrypt the key associated with the databases consisted of a user selected passphrase, which the user had to enter every time he/she chooses to use the application. Now upon analyzing the volatile memory of the application using a tool named LiME [71], Anglano et al. were capable of retrieving this user selected passphrase, and thus, they were capable of retrieving the key associated with the databases, and subsequently employing it to decrypt their contents.

Although the authors were capable of retrieving the passphrase by reading the volatile memory, the attack that they employed does not seem very practical in real world scenarios, as it requires having the application running, with the passphrase already inputted into the application. In other words, for their attack to work, a user must open the ChatSecure application, input the passphrase which unlocks the key associated with the encryption/decryption of the databases, shortly after which an attacker gains access to the device, and runs the required tools to scan through the volatile memory available at the time, in the interest of detecting the utilized

passphrase. Nevertheless, the attack that they performed could be possible to execute, and proper measures should absolutely be taken to secure the volatile memory as much as possible.

The security involving the volatile memory has been the main research topic for several research papers (see, for example, [68, 72, 73]). For instance, the research paper [68] involved an experiment in which Ntantogian et al. analyzed the volatile memory in an attempt to detect authentication credentials of certain Android applications. They determined that most of the Android applications that they examined were indeed vulnerable to revealing their users' credentials within the volatile memory. These credentials sometimes include user selected passwords that are utilized for more than one service. Thus, a breach of these credentials can without a doubt be extremely harmful. To avoid revealing sensitive information through the volatile memory, developers must take the extra steps in ensuring that all potentially private information is properly deleted from the memory just as they stop being used.

3.3.3.2 Traffic Analysis and Leakage of Metadata

When it comes to secure messaging, ensuring that we do not leak metadata can sometimes be as important as ensuring that the messages that we want to send are properly encrypted. Basically, metadata is information that describes other data [74]. In the context of secure messaging applications, metadata can refer for example to information regarding the sender of a message, the receiver, the date a message was sent, the type of request that was submitted to a central server, the length of a message, etc. There are several research papers that focused on analyzing the existing traffic that results from applications deployed on users' devices (see, for example, [75, 76,

77]). In [77], Coull et al. focused on analyzing the traffic resulting from encrypted messaging services; the authors mainly aimed their attention at Apple's iMessage software, however they explained that their results apply to other mobile applications as well, including WhatsApp, Viber, Telegram, and others. In the paper [77], Coull et al. explained that they started off by monitoring the Apple Push Notification Service (APNS) and TLS connections linked to the iMessage application, whereby they determined that the application involved five main observable user actions: (1) when a user starts typing a message, (2) when a user stops typing a message, (3) when a user sends a message, (4) when a user sends an attachment, and (5) when a user sends a read receipt of a certain message. Coull et al. captured packet examples with regard to each user action. Upon analyzing the packets and monitoring their corresponding payload lengths, they were capable of determining information regarding the user action associated with a certain packet, as well as some information about the operating system being used and about the language of the plaintext content available within the encrypted packets.

The available countermeasures that can be applied to mitigate against such leakage of information from analyzing the traffic on mobile devices are fairly limited. In one study [78], Dyer et al. examined several existing traffic analysis countermeasure techniques. Some of these techniques involved obfuscating the length of the plaintext within packets by adding to it random amounts of extra padding. Other techniques involved changing the packet length to the nearest multiple of 128 (linear padding) or to the nearest power of two (exponential padding). The authors' results demonstrated that none of the existing countermeasure techniques that they examined were effective. They explained that hiding the length of packets is not sufficient to mitigate traffic analysis attacks.

3.4 Common Issues with E2EE Messaging Applications

Based on the EFF's Secure Messaging Scorecard and our literature review that we discussed in sections 3.3, we developed a list of common issues that are applicable to E2EE messaging applications. The issues are listed below.

3.4.1 Trust-Related Issues

There are numerous mobile applications that support E2EE messaging that are available on the Google Play store. The extent to which these applications are trusted by users highly depends on their associated level of transparency. Essentially, the transparency associated with these applications boils down to three of the seven criteria listed in EFF's Secure Messaging Scorecard [64]:

1. "*Is the code open to independent review?*"
2. "*Is the crypto design well-documented?*"
3. "*Has there been an independent security audit?*"

The first criterion which requires that the code be open for independent review is crucial when it comes to users' trust with regard to an application. The fact is that an application can claim to offer limitless security features as part of its implementation; however, if the code associated with the application is not open for independent review, then there is no way for the users to be sure that

this application indeed secures their data as promised. In other words, the users have to trust that the application is truthful and is indeed properly securing their data. The second criterion which requires that the crypto design be well-documented is also important as it can facilitate potential review processes by professional cryptographers [64]. The third criterion, which has to do with available independent security audits, is also crucial when it comes to the trustworthiness of these applications. In the case of the EFF's Secure Messaging Scorecard, the independent security audits did not necessarily need to be published to satisfy this criterion, unpublished security audits could also be satisfactory; the condition however for unpublished security audits is that it is required that there exists a named party that is willing to confirm that the audit indeed occurred [64]. It is important to note that having the code available for independent review is never a sufficient condition for applications to be considered secure; the review of an application's structural design and the continuous audit of an application's code are essential in order to have some confidence with regard to the security of these applications [79].

Unfortunately, the results displayed in the EFF's Secure Messaging Scorecard showed that only 41% of the implicated applications had their code open for independent review. 62% of the applications had their cryptography design well-documented, and 57% of the applications had an independent security audit [64]. In other words, roughly half of the secure messaging applications implicated in the scorecard did not provide a way for their users to verify the security that they offered, and the users essentially have to trust their services when using them.

To increase the confidence that users have in their application, application developers can resort to publishing their code for open review, and ask professional cryptographers to audit their

code and publish the results they obtain. However, publishing the source code online is not always a feasible option; in fact, there are many reasons that can prevent this. Sometimes the code cannot be published as it embodies intellectual property, other times the code may contain private API keys used to control the access to a service's APIs, etc.

3.4.2 Protocol-Related Issues

As we have seen in our literature review, sometimes there are issues that are related to the algorithms and security protocols applied within certain applications. Security protocols tend to be very complex in E2EE messaging applications. In general, it is recommended that applications utilize published algorithms that have been the subject to extensive cryptanalysis [80]. In the article published in [81], Schneier points out: "*Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis. And the only way to prove that is to subject the algorithm to years of analysis by the best cryptographers around.*" Unfortunately, some applications implement their own cryptographic algorithms to provide end-to-end encrypted messaging to their users. These algorithms may or may not have major vulnerabilities; nevertheless, users are taking a risk by utilizing such applications.

3.4.3 Implementation-Related Issues

Upon reviewing the literature regarding some of the existing secure messaging applications, we realized that often times, the cryptographic principles applied are well designed and

theoretically very secure. However, there are often problems resulting from the implementation details concerning these protocols. Exploits relating to secure messaging applications most often have to do with the implementation specifics regarding cryptographic functionalities. Essentially, any small bug in the cryptography-related implementations can potentially lead to a breach of the entire security of an application; without proper analysis and penetration testing, there exists plenty of room for error and for major vulnerabilities. In E2EE messaging applications, when exploits are found, they often have to do with either the authentication aspect of the application, e.g. at the registration stage, or with the key management aspect, e.g. by interfering with the central server setting up and providing public key information to the entities involved in a conversation. Sometimes, there are exploits that take advantage of bad key generation, e.g. by not seeding key generators properly, and at other times, exploits have to do with retrieving key related information from the memory, e.g. keys or passwords that were not properly stored or deleted from the external, internal, or volatile memory. The fact is that creating a secure E2EE messaging application while considering all the possible angles through which an exploit may occur is a fairly difficult task, especially when there are various security functionalities that our application needs to implement.

3.4.4 Metadata-Related Issues

Leakage of metadata is an issue that is often over-looked when dealing with secure applications. As we have discussed in section 3.3.3.2, in the context of E2EE messaging applications, metadata can include information regarding the sender of a message, the receiver, the date a message was sent, the language of a message, etc. Even if the principal messages between two parties are end-to-end encrypted, mismanagement of metadata can allow an eavesdropper to

infer sensitive information (e.g. location, relationship) about these involved parties [82]. It has been shown that it is possible to extract metadata from several existing secure messaging applications [77]. Additionally, some of the applications encrypt and keep records of their users' metadata [10, 83]. There are several techniques that help application developers reduce the loss of privacy resulting from the leakage of metadata. However, many of the existing countermeasures have not proven to be very effective [78].

3.5 Conclusion

In this chapter, we listed and described security properties that are often desired in E2EE messaging applications. We examined existing work that has to do with analyzing the security of such applications, and we presented a literature review. Based on the results of our research, we derived four categories of common issues that are typically associated with E2EE messaging applications. The four categories consisted of (1) trust-related issues, (2) protocol-related issues, (3) implementation-related issues, and (4) metadata-related issues. In Chapter 5, we propose a new design for E2EE messaging applications which employs security features available in the Android operating system, in the interest of diminishing the common issues that we listed in this chapter.

Chapter 4: Unauthorized Access to Protected Data Using IPC Mechanisms

4.1 Introduction

In Chapter 2, we provided a brief overview regarding the Android Software Framework, wherein we listed some of the principal security features available in the Android operating system. These security features included the user-based permissions model, the inter-process communication mechanisms, and many more. The user-based permissions model, which is often referred to as the Android permission model, is one of the most important security features that are available in the Android operating system, as it controls the access applications have to protected data resources (e.g. SMS data, calendar events, current location, etc.) that are available on a user's device. As an "access surface" of protected data, it is currently impossible to bypass the security imposed by the model. Upon studying the architecture of the Android operating system, and reviewing some of the existing literature [42, 43, 44], we learned that although the Android permission model is initially the unique access surface to certain protected resources, once an application becomes privileged (i.e., granted permissions to access certain resources on a user's device), that application may become an "access surface" itself to the protected resources it has been given access to. This is permitted by the inter-process communication mechanisms provided by the Android operating system. Essentially, there is nothing stopping a malicious privileged application from sending protected data to another unprivileged application on a user's device via the IPC mechanisms. In this chapter, we explain and demonstrate how IPC mechanisms can be

used to achieve this behavior. Moreover, we propose potential countermeasures that can help prevent this unauthorized access of protected resources on a user's device.

4.2 Access Surfaces of Protected Data Resources

4.2.1 Accessing Protected Data Through the Permission Model

One of the main roles of the Android permission model is to control the access applications have to content providers and to data resources available on a user's device. Whether users grant permissions to applications at installation-time or at run-time, applications by default cannot access protected data resources on a user's device without having the appropriate privileges. The main idea behind this access control mechanism is depicted in figure 6 below.

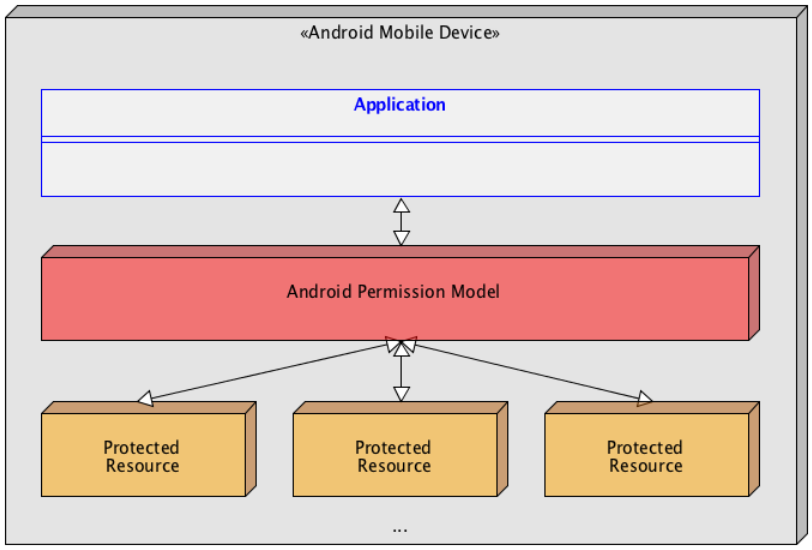

Figure 6 - Accessing Protected Resources Through the Android Permission Model

Basically, when an application wants to access a protected resource (e.g. current location), the permission model verifies whether that application has been granted the appropriate privileges or not, and provides access to the data accordingly. Upon gaining access to protected resources, an application may choose to perform any of the following actions:

- Store the data locally.
- Store the data online.
- Perform an action/operation using the data.
- Dismiss the data.

In order for an application to access protected resources secured by the Android permission model, the application must declare the permissions it requires from the user in its AndroidManifest.xml file. Thereafter, the user would need to grant that permission to the application during installation-time or run-time. Once this permission is granted, the application may access the protected resources.

4.2.2 Accessing Protected Data Through Privileged Applications

Although the Android permission model does a great job at protecting users' data, the inter-process communication mechanisms accessible to all applications available on a user's device make it so that not all applications are forced to go through the permission model to access protected resources. The fact is that once an application gains access to certain protected resources,

there is nothing preventing that application from serving as an access surface to this data for other applications. This concept is illustrated in figure 7 below.

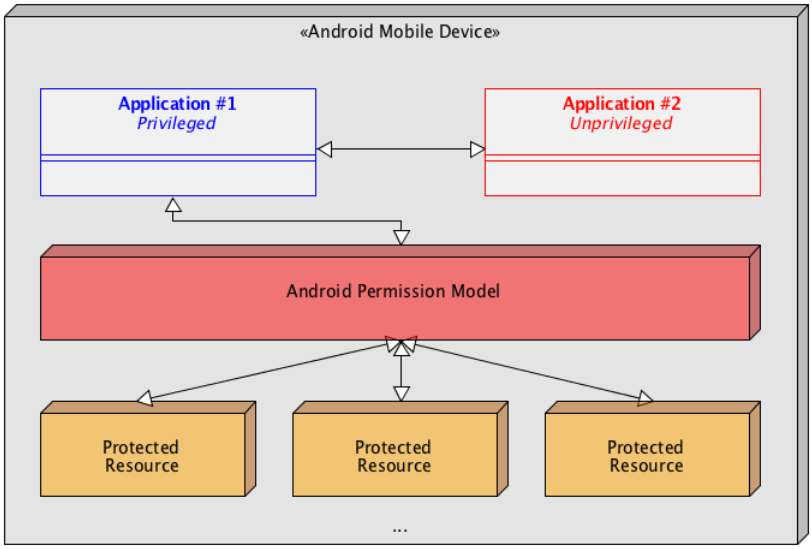

Figure 7 - Accessing Protected Resources Through Privileged Applications

In Chapter 2, we saw that there are multiple IPC mechanisms that can be utilized to allow applications to communicate with each other. Figure 8 below demonstrates how the broadcast receiver and broadcast sender intent components can be combined together to achieve this behavior.

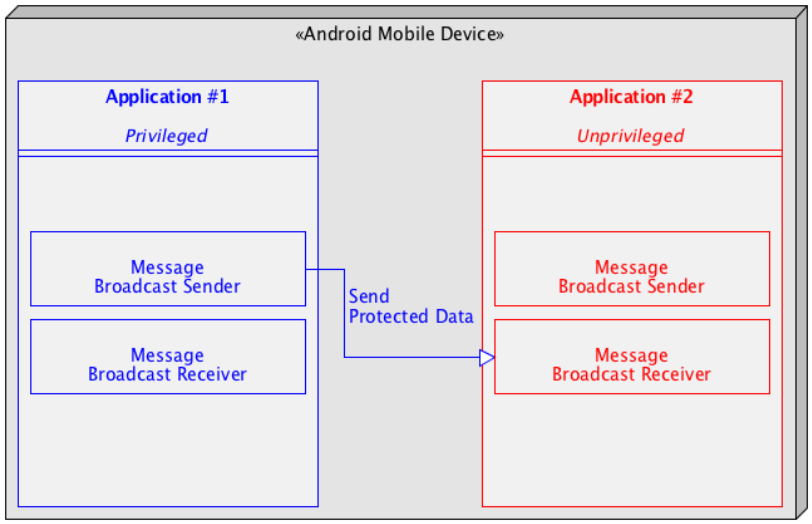

Figure 8 - Using broadcast receivers and broadcast sender intents to send protected data from a privileged application to an unprivileged application

The IPC mechanism described in figure 8 involves three notable components:

1. **Protected Data**: The protected data refers to any kind of data protected by the Android permission model. This could include SMS/MMS messages, calendar events, call log information, location details, etc. In this context, application #1 is authorized to access this data, while application #2 is not authorized. In other words, the user granted application #1 the appropriate privileges to access this data, but he has not granted the same privileges to application #2.
2. **Message Broadcast Sender**: This component uses broadcast sender intents to send a message from one application to the other. In order for application #1 to send a message directed at application #2, the intent must target an action specified by application #2 (e.g. getProtectedData). The intent must also include the message content (i.e. the protected data) that application #1 wants to send to application #2. Application #1 can then utilize

the built-in function sendBroadcast(Intent) [84] to instantly send the message to application #2.

3. **Message Broadcast Receiver**: The broadcast receiver listens to broadcasted messages from other applications. The broadcast receiver must specify an action to listen to (e.g. getProtectedData). Whenever an application sends a broadcast with the action specified by the broadcast receiver, the broadcast receiver is capable of picking up the broadcasted message, extracting its content and processing it.

4.3 Proof of Concept #1: Unauthorized Access of Protected Data

4.3.1 Introduction

In order to demonstrate how IPC mechanisms can be applied to allow a privileged application to send protected data to an unprivileged application on a user's device, we created two Android applications: application #1 and application #2. Application #1 is a privileged application that is granted the permission to access the user's current location. Application #2 is an unprivileged application that is not allowed to access the user's location. In this proof of concept, we demonstrate how application #2 can easily retrieve the user's current location from application #1 by using broadcast sender intents and broadcast receivers. In this section, we provide brief descriptions of application #1 and application #2. More details regarding these applications can be found in Appendix A.

4.3.2 Application #1 (Privileged Application)

In order to access the current location of a user's device, application #1 must request the permission entitled android.permission.ACCESS_FINE_LOCATION, which the user must grant to the application. Screenshots of application #1 are shown in figure 9 below.

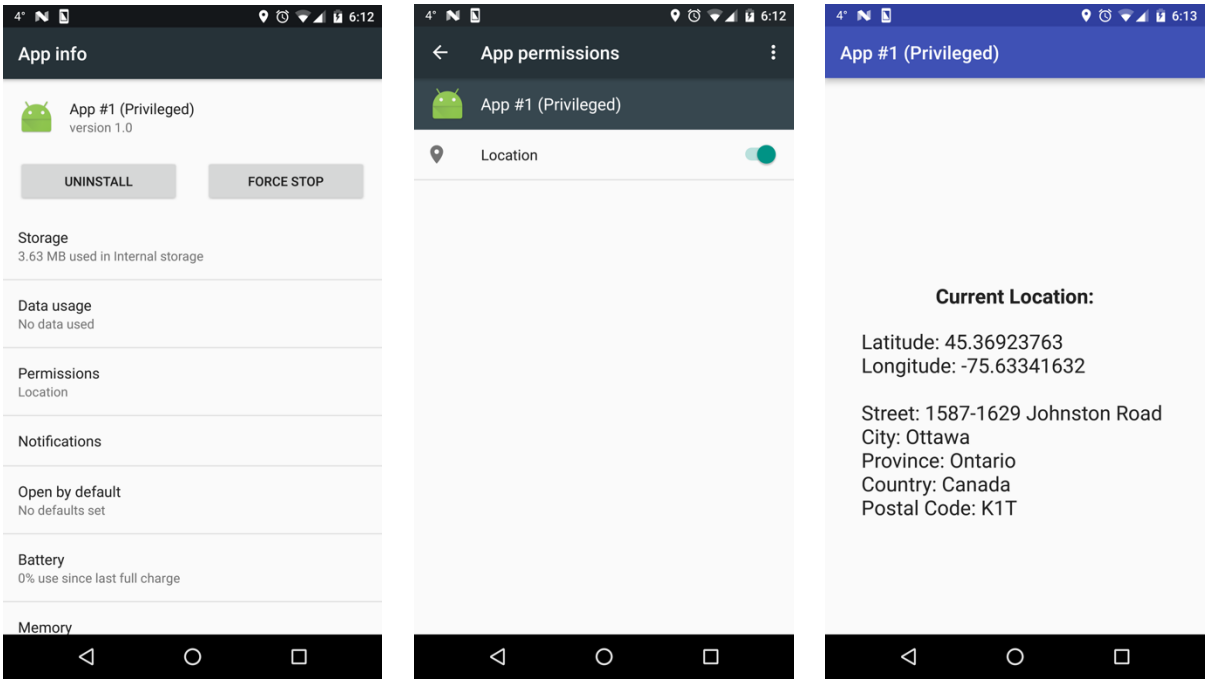

Figure 9 - Screenshots of Application #1

4.3.3 Application #2 (Unprivileged Application)

Application #2 does not have the permission to access the user's current location. We built application #2 such that there is a button labelled "Get User's Location". When the user clicks on that button, the following actions occur:

1. Application #2 sends a message to application #1 via a broadcast sender intent.

2. Application #1's broadcast receiver picks up the message sent by application #2.
3. Application #1 launches a service which accesses the user's current location.
4. Application #1 sends a message with the current location of the user's device back to application #2 via a broadcast sender intent.
5. Application #2's broadcast receiver picks up the message sent by application #1, and displays the current location on the screen.

Screenshots of application #2 are shown in figure 10 below.

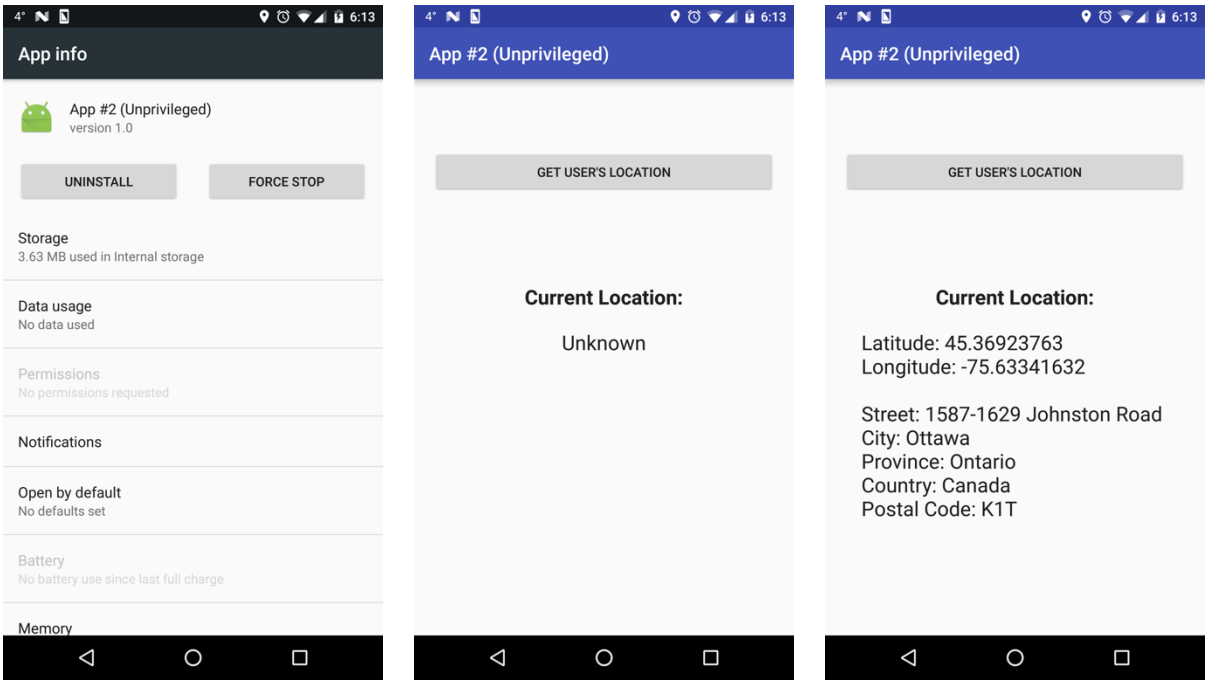

Figure 10 - Screenshots of Application #2

As shown in figure 10, application #2 is not granted any permissions by the user, however it is still capable of accessing the user's current location, by retrieving the data from the privileged application #1.

4.4 Potential Countermeasures and Defenses

The security imposed by the Android permission model works great when the permission model is the one unique "access surface" of protected resources. However, as we have discussed in section 4.2.2, once an application is granted permissions by a user, that application may itself serve as an "access surface" of protected resources. Furthermore, with the current architecture of the Android operating system, there are no built-in mechanisms that prevent such privileged applications from sending protected data to unprivileged applications on a user's device. In order to prevent this behavior from happening, changes need to be made to the operating system itself. In this section, we list and discuss three potential countermeasures that can help prevent or reduce the impact this behavior has against users' privacy.

4.4.1 Same Privilege Requirement for IPC

In section 2.5.4, we mentioned that some research papers explored solutions to the confused deputy attack, whereby a deputy is a non-malicious privileged application that has been granted permissions by the user. In this chapter, we consider that a malicious application that is deployed on a user's device may choose to send protected data to other unauthorized applications using IPC mechanisms. In [42], Felt et al. presented IPC Inspection as a security mechanism to prevent confused deputy attacks. The idea behind IPC Inspection was to reduce the permissions that correspond to applications that are communicating with each other using IPC mechanisms, in such a way that if an application that is less privileged communicates with an application that is more privileged, the permissions of the application that is more privileged get reduced during run-

time, thus preventing the less privileged application from gaining unauthorized access to protected resources. As shown in [42], IPC Inspection proved to be successful in preventing confused deputy attacks. However, when we consider the privilege escalation problem occurring due to a malicious privileged application (as opposed to non-malicious), the defence mechanism becomes trickier. For instance, a malicious application may periodically access certain protected resources and cache the data. Thereafter, when an unprivileged application communicates with this application, although the permissions get reduced, the privileged application can still access the data through the cache, and forward it to the unprivileged application. One way to prevent this behavior from happening would be to prevent applications from communicating with each other, unless they have been granted the same permissions by the user. In other words, if an application A is granted the permission to read the calendar events on a user's device, that application should not be allowed to send a message to another application, application B, that does not have the same permission. Due to the fact that we cannot always monitor (e.g. in the case of explicit intents) the content of messages that are being exchanged between two applications, the only way to guarantee that a privileged application is not sending protected data to an unauthorized application would be to require that the two applications be granted the exact same permissions by the user.

This problem is similar to other problems in the field of multilevel security. In multilevel security systems, we sometimes have a hierarchy of classifications. For example, in very simple terms, we could have classifications such as (1) *Top Secret*, (2) *Secret*, (3) *Confidential*, and (4) *Unclassified*, and we could want to control the data access in such a way that data can only flow from the lower levels to higher levels, and not the other way around [85]. For instance, a process with the *Confidential* classification should be able to read and write *Confidential* or *Unclassified*

data, however a process with the *Unclassified* classification should not be able to access any *Confidential* data, and may only read or write *Unclassified* data. With regards to the privilege escalation problem that we discussed in this chapter, one can think of application #2 as the *Unclassified* process, and of application #1 as the *Classified* process. The issue that we are having would be that the *Classified* process is capable of sending data to the *Unclassified* process without any restrictions. In the field of multilevel security, certain solutions were derived for such problems, an example of which would be the Bell-LaPadula model [85]. For example, the Bell-LaPadula model could be used to achieve the same privilege requirement that we discuss in this section by utilizing the *Strong Star Property* which requires that the READ and WRITE for a user be only at the same level (i.e. cannot read or write from higher or lower levels). The effectiveness and feasibility of this and other possible methods to achieve the *Strong Star Property* are outside the scope of this thesis, and should be explored and studied further in future work.

4.4.2 More Transparent IPC Mechanisms

Another technique to help reduce the impact that the discussed behavior has on user's privacy would be to enforce transparency of the IPC mechanisms. In the proof of concept that we discussed in section 4.3, we utilized statically declared (i.e. in the AndroidManifest.xml file) broadcast receivers and dynamic (i.e. in code) broadcast sender intents to permit the inter-process communication between application #1 and application #2. In other words, if a user were to review the statically declared components in our applications, he would determine that application #1 and application #2 both have broadcast receivers, but he would not be able to determine whether these two applications are indeed communicating with each other or not, due to the fact that the broadcast

sender intents are not statically declared. In fact, there is currently no possible way to declare broadcast sender intents statically. Providing means to statically declare components used for IPC, and enforcing the utilization of such declarations would render these components more transparent. Although it may not always be the case, transparency of such components could increase users' trust when it comes to applications deployed on their phones. This technique is in no way sufficient to guarantee that a privileged application is not sharing protected data with other unprivileged applications. However, it can raise user awareness with regard to applications that are unexpectedly communicating with each other on their device.

It is important to note that it is indeed possible for users to get information regarding the components that were statically declared in applications deployed on their devices. In Android, some components are forced to be statically declared in the AndroidManifest.xml file, while others can be declared statically or dynamically [86]. Out of the principal components that we saw in Chapter 2 section 2.3, all activities, services, and content providers have to be declared in the AndroidManifest.xml file [87, 88, 89]. On the other hand, broadcast receivers may be declared in the AndroidManifest.xml file, or context-registered dynamically during run-time [90].

There are lots of applications available on the Google Play Store that collect and display to users information regarding the permissions requested/granted to applications available on their mobile devices. However, none of these applications listed the other statically declared components. For this thesis, we also built an application that lists, for each application installed on a user's device, all the statically declared permissions, activities, receivers, services, and providers. We retrieved the data through the `PackageManager` [91] and `PackageInfo` [92] classes

available in the operating system. Although this information does not solve the issue of unauthorized access of protected data that we brought to attention, this application can help users gain some insights into the components available within the applications deployed on their devices. Figure 11 shows screenshots of the application that we built. More details regarding this application can be found in Appendix A.

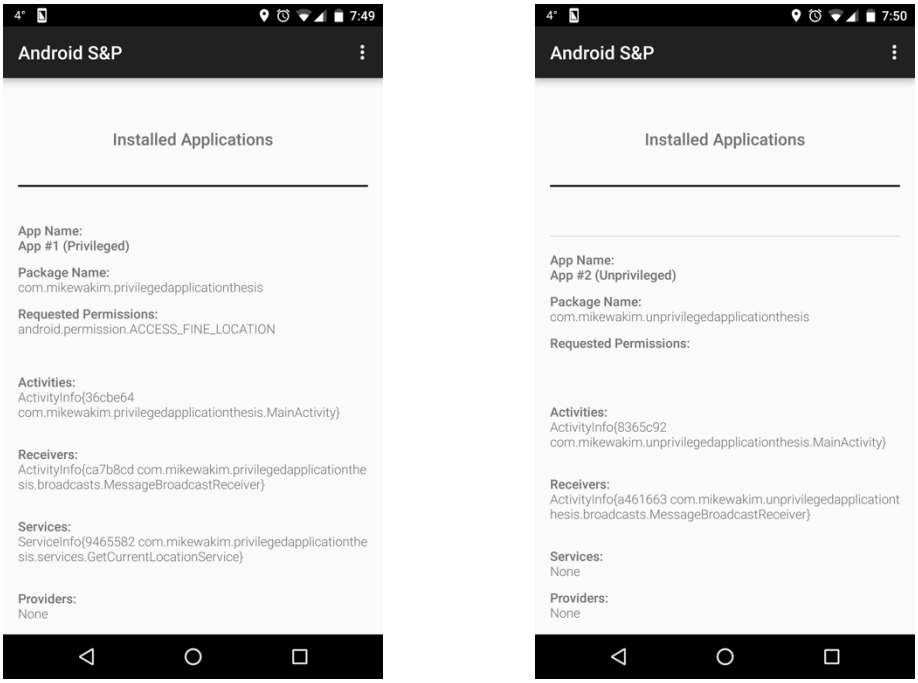

Figure 11 - Screenshots of the Android Manifest Explorer Application

4.4.3 Intermediate Layer of Security for Ongoing IPC

Instead of enforcing that applications have the exact same permissions in order to be capable of communicating with each other, it would be possible to create an intermediate layer of security for ongoing IPC, wherein when two applications are communicating with each other, the

user gets presented with a dialog asking him to authorize this communication. This can be somewhat similar to the run-time permissions available in later versions of the Android operating system. This technique may be problematic if there are too many IPC instances ongoing at the same time, however it is worth exploring nonetheless.

4.5 Conclusion

In this chapter, our objective was to provide an overview of the issue wherein a privileged application on a user's device may choose to send, through the utilization of IPC mechanisms, protected data to other unauthorized applications. We demonstrated through a proof of concept how an application without the appropriate permissions can access a user's current location by retrieving it from a malicious privileged application by employing broadcast receiver and broadcast sender components. Furthermore, we listed and discussed three potential countermeasures that can help prevent or reduce the impact caused by the issue that we discussed. The countermeasures all required changes to be made to the operating system. The effectiveness and efficiency of these techniques should be studied further in future work.

Chapter 5: Employing IPC for E2EE Messaging Applications

5.1 Introduction

As we have seen in Chapter 3, there are issues often associated with existing E2EE messaging applications. We classified these issues as:

1. Trust-Related Issues
2. Protocol-Related Issues
3. Implementation-Related Issues
4. Metadata-Related Issues

In this chapter, we take a look at the conventional design that is typically implemented by Android E2EE messaging applications. We consider the effect that this design has on these applications with regard to the common issues listed above. In the aim of diminishing these common issues, we propose a new design that takes advantage of the existing Android security features we discussed in Chapter 2 and which allows applications to support E2EE messaging. Our design focuses on the trust-related issues applicable to E2EE messaging applications. With regard to the protocol, implementation, and metadata details, the design is generic; developers can choose to implement whichever protocols and algorithms they need. Moreover, we demonstrate, through a proof of concept, how our design can be put to use in a real-world application. In this chapter, we

provide a brief overview of the system that we built in our proof of concept. We provide more details regarding this system in Appendix B.

5.2 Conventional Design

E2EE messaging applications have proven to be extremely useful when it comes to securing users' communications on their mobile devices. They are without a doubt better alternatives in comparison to native applications that do not provide encryption at all. Although there are various applications available on the Google Play store that offer their users different security properties, these applications share a lot in common. From a high-level architectural point of view, these applications are nearly identical. Figure 12 below illustrates the high-level architecture typically deployed by E2EE messaging applications.

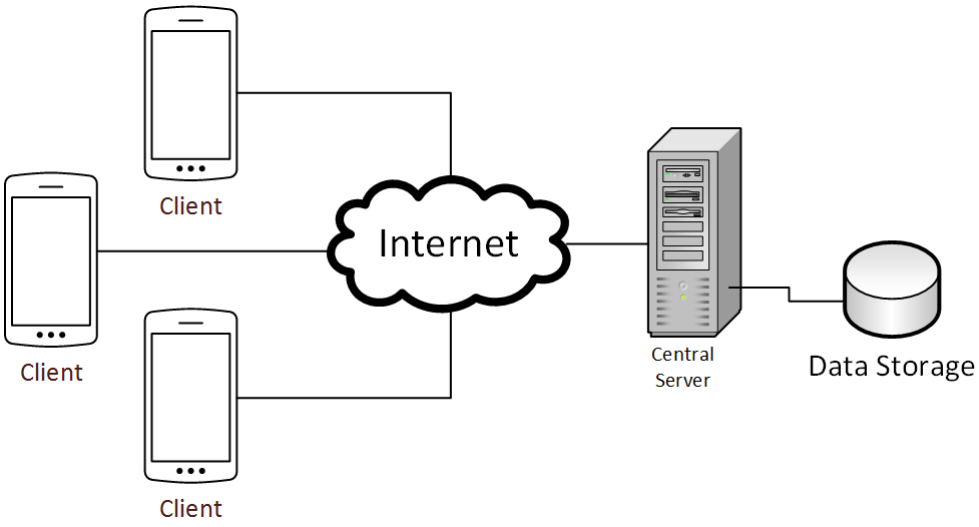

Figure 12 - Typical High-Level Architecture of E2EE Messaging Applications

Essentially, these applications typically implement a client-server architecture, with the applications on the users' mobile devices serving as the clients, and the central server serving as the server.

In the context of E2EE messaging applications, the client side consists of the application itself deployed on a user's device. Once again, although different E2EE messaging applications may offer different security properties, the structure of these applications is in most cases very similar. Essentially, there are several groups of functionalities that each of these applications must implement. These groups of functionalities are illustrated in figure 13 below.

Figure 13 - Typical Groups of Functionalities in E2EE Messaging Applications

In figure 13, we list some of the groups of functionalities that typically constitute a secure messaging application. We describe these groups as follows:

1. **Security Functionalities**: Without a doubt, E2EE messaging applications need to implement their own set of security-related functionalities to deliver to their users the security properties that they offer. In general, applications need to implement the functionalities required for the following:
    1.1. **Authentication**: Authentication can be required by an application for two principal intents: (1) for controlling the access to the application itself, and (2) for authenticating any ongoing communications. To control the access to an application, application developers can resort to authentication techniques such as text-based or graphic passwords, biometrics (e.g. fingerprint), PINs, etc. For additional security, it is recommended that users lock their devices using the mechanisms offered by the Android operating system (i.e. password, fingerprint, etc.). Authenticating ongoing communications is a more complex task that typically depends on the specific protocols the developers choose to implement.
    1.2. **Key Management**: Key management is defined by [58] as "*the activities involving the handling of cryptographic keys and other related security parameters (e.g. initialization vectors) during the entire lifecycle of the keys, including their generation, storage, establishment, entry and output, use and destruction*". Regardless of whether the applications utilize symmetric or asymmetric encryption techniques, the functionality required for key management is essential. Different algorithms may require different techniques for key generation and key exchange.
    1.3. **Session Management**: Session management consists of the activities involving the creation, modification, and deletion of session-related data. In the context of E2EE secure messaging applications, a session between two (or more) different parties would be

associated with data such as conversation IDs, sender information, receiver information, temporary tokens, etc.

1.4. **Encryption/Decryption**: There are various encryption algorithms that can be utilized by E2EE messaging applications. The encryption algorithms can be symmetric, whereby the parties involved in a session share a common secret key, or asymmetric, whereby two keys are used, one of which is public and utilized for encrypting data, while the other one is private and utilized for decrypting data [93]. Applications often resort to combinations of symmetric and asymmetric encryption techniques to provide good security while maintaining a reasonable system performance.

1.5. **Integrity Verification**: E2EE messaging applications often implement integrity verification techniques as a means to ensure that the data being exchanged between different entities has not been modified in any way by another party.

1.6. **Digital Signing**: Although they may not always be necessary within E2EE messaging applications, digital signatures may be required by certain applications, as they can serve as means for authentication, integrity verification, along with support for non-repudiation for the original signer [58]. There are various digital signature algorithms that developers of E2EE messaging applications can choose to implement.

2. **Network Functionalities**: The network functionalities are the functionalities needed for achieving two-way communication with the central server associated with the corresponding application. This typically includes the functions needed to send/receive web requests, and to make API calls.
3. **User Interface Functionalities**: The User Interface (UI) functionalities are the functionalities needed to supply a working user interface for the users. In Android applications, this generally

contains all the activities corresponding to a certain application, in which the expected behavior of certain components gets implemented.

4. **Other Functionalities**: If required, other groups of functionalities may also exist in E2EE messaging applications. For example, some applications may need to implement database functionalities, cache functionalities, etc.

Applications combine these groups of functionalities to allow their users to securely communicate online using their service. Assuming that the key management aspect has already been taken care of, the sequence of events that occur between a user, an E2EE messaging application, and the application's central server is illustrated in figure 14 below. In figure 14, Alice (sender) sends an encrypted message to Bob (receiver).

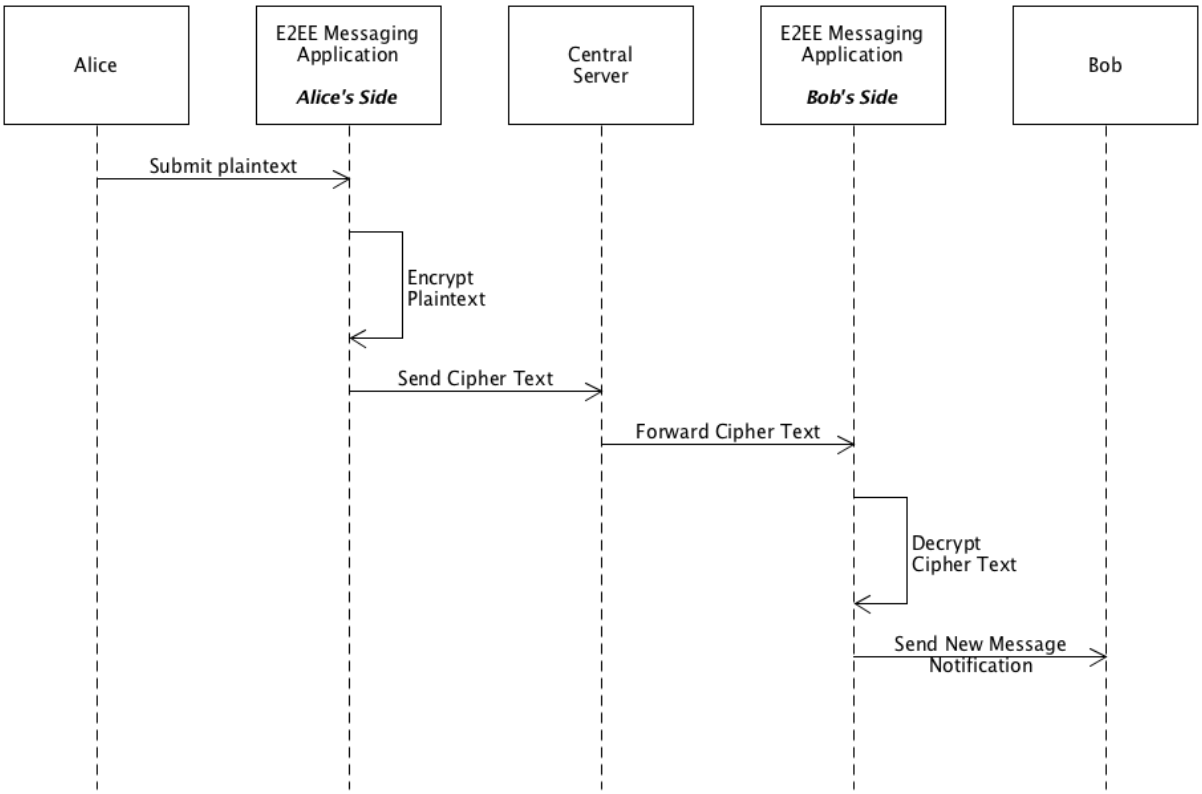

Figure 14 - Sequence Diagram of Basic E2EE Messaging Sequence

5.2.1 Effect of the Design on Issues Related to E2EE Messaging Applications

The conventional model illustrated in figures 13 and 14 above is straight-forward. Basically, applications must implement all the groups of functionalities that they need, to provide the services that they want to offer. There are existing libraries online that offer certain functionalities, e.g. for specific cryptographic primitives, for performing web requests, etc. However, developers would still need to assemble functions together to realize their application-specific requirements, and if certain functions are not used properly, they could still lead to major exploits; for example, an application that uses a secure AES implementation from an existing library, may still fail to

securely generate and manage keys. The conventional design simply works. However, it is when we look at the model from a security and privacy preservation perspective that some disadvantages become apparent.

5.2.1.1 Effect on the Trust-Related Issues

As we have discussed in section 3.4.1, the trust-related issues that are implicated with some E2EE messaging applications mostly have to deal with three of the seven criteria listed in EFF's Secure Messaging Scorecard [64]:

1. "*Is the code open to independent review?*"
2. "*Is the crypto design well-documented?*"
3. "*Has there been an independent security audit?*"

When it comes to the criterion that has to do with the crypto design being well-documented, and the criterion that requires that there be an independent security audit, the conventional design has no effect. However, the conventional design can have a great influence on whether the code is open to independent review or not.

Secure systems that deal with users' sensitive information are often very complex systems that require the implementation of various components, such as security-related components, network-related components, user interface components, database components, etc. Due to the fact that the conventional design bundles all the groups of functionalities associated with these

components together, rendering the code open to independent review would require exposing the code for all these implicated components. However, not all of these components necessarily deal with users' sensitive information. For instance, the network-related components can sometimes strictly deal with cipher text along with inevitable (preferably minimized), accompanying metadata. The fact is that network-related components often implicate certain elements that the application developers may not want to make public; these elements may include proprietary communication protocols, proprietary optimization techniques, secret API keys that control the access to certain APIs and servers, etc. From that perspective, the conventional design may prevent some applications from exposing their entire code for independent review.

5.2.1.2 Effect on the Protocol, Implementation, and Metadata Related Issues

The conventional design does not have any direct effects on the protocol, implementation, and metadata related issues. This is due to the fact that there are no restrictions with regard to which protocols the developers can implement and how they choose to implement them. When a user installs an application that supports E2EE messaging, if that application was subjected to extensive analyses, penetration testing, and professional independent security audits, the conventional design can be extremely cooperative and create no setbacks whatsoever. Nevertheless, it is important to keep in mind that due to the fact that the conventional design requires that applications implement all the functionalities that they require, it is inevitable to have some duplication/reproduction of security-related implementations when users download multiple E2EE messaging applications. In other words, if a user downloads two or more applications that provide E2EE messaging, each of these applications must have its own implementation of its required

security-related functionalities. As a consequence, unless every single one of these applications has been subjected to extensive analyses and penetration testing, there will be more potential room for error. As we have discussed in section 3.4.3, any small bug in the security-related implementations can potentially lead to a breach of the entire security of an application.

5.3 Proposed Design

5.3.1 Main Concept

Upon examining the architecture of the Android operating system, along with its platform and its application building blocks, we came up with a new model that separates the group of security-related functionalities from the group of network-related functionalities. The separation is illustrated in figure 15 below.

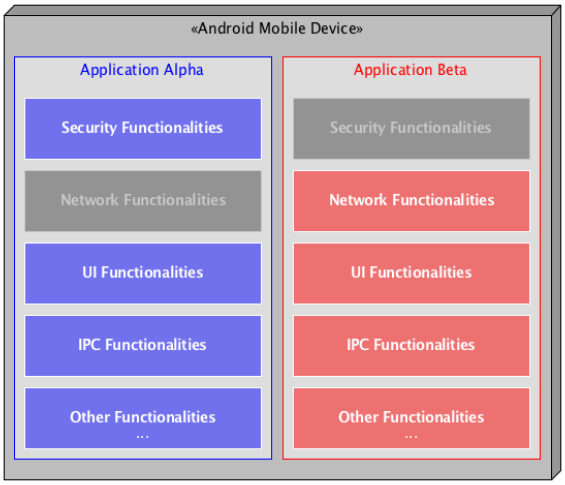

Figure 15 - Separated Groups of Functionalities

Essentially, unlike the conventional design whereby we have one application that implements all the groups of functionalities, we instead divide the groups of functionalities and implement them within two separate applications: application Alpha and application Beta. Application Alpha is responsible for implementing the group of security-related functionalities. Inevitably, it must also implement the corresponding user interface functionalities, and possibly other groups of functionalities as well (e.g. database, cache, etc.). On the other hand, application Beta is responsible for implementing the group of network-related functionalities. It must also implement its group of user interface functionalities and possibly other groups of functionalities if required. The main idea here is that we want to separate all the security-related functionalities from the network-related functionalities, in the aim of making it possible for application developers to release their security-related code (i.e. application Alpha) for independent review, without being forced to expose their network-related code (i.e. application Beta) along with it.

Once again, assuming that the key management aspect has already been taken care of, the sequence of events that occur between a user, application Alpha, application Beta, and the applications' central server is illustrated in figure 16 below. In figure 16, Alice (sender) sends an encrypted message to Bob (receiver).

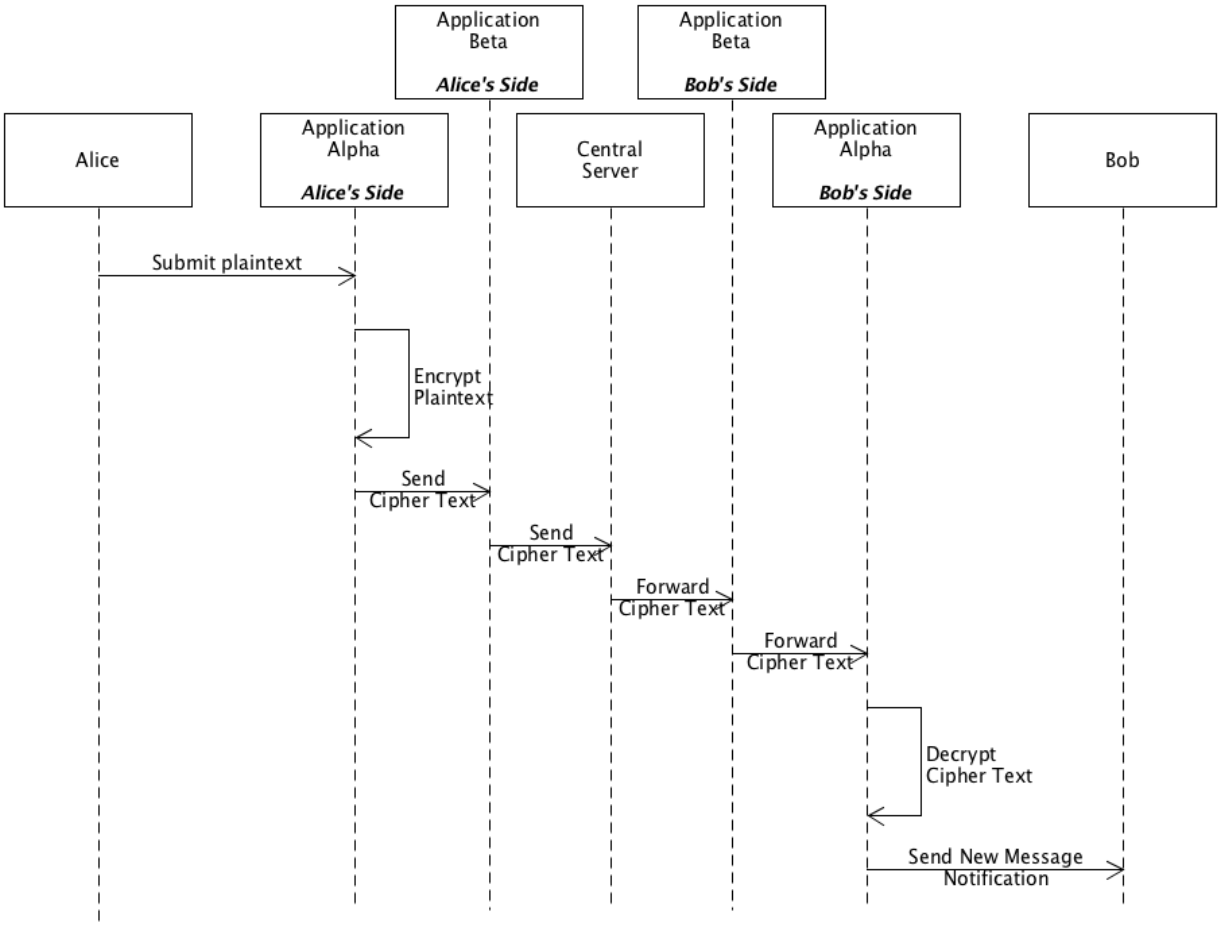

Figure 16 - Sequence Diagram of the E2EE Messaging Sequence in the Proposed Design

5.3.2 IPC Between Application Alpha and Application Beta

Application Alpha is the application that deals with all the security-related functionalities. From the sender's point of view, it is the entry point of plaintext, and the exit point of cipher text. On the other hand, from the receiver's point of view, it is the entry point of cipher text and the exit point of plain text. As depicted in figure 16 above, in order for a user to send the cipher text to another party, application Alpha must send the cipher text to application Beta, which in turn forwards the cipher text to the central server. Following this, the central server forwards the cipher

text to the intended recipient's application Beta, which then forwards the cipher text to the recipient's application Alpha. This communication channel that takes place between application Alpha and application Beta is possible due to the inter-process communication security feature inherited from the Linux kernel. To achieve this inter-process communication, we employ the broadcast receiver component that we discussed in sections 2.3.5 and 2.4.3, along with broadcast sender intents. Figures 17 and 18 below demonstrate how our design puts to use the broadcast receivers and broadcast sender intents to permit the exchange of data between application Alpha and application Beta.

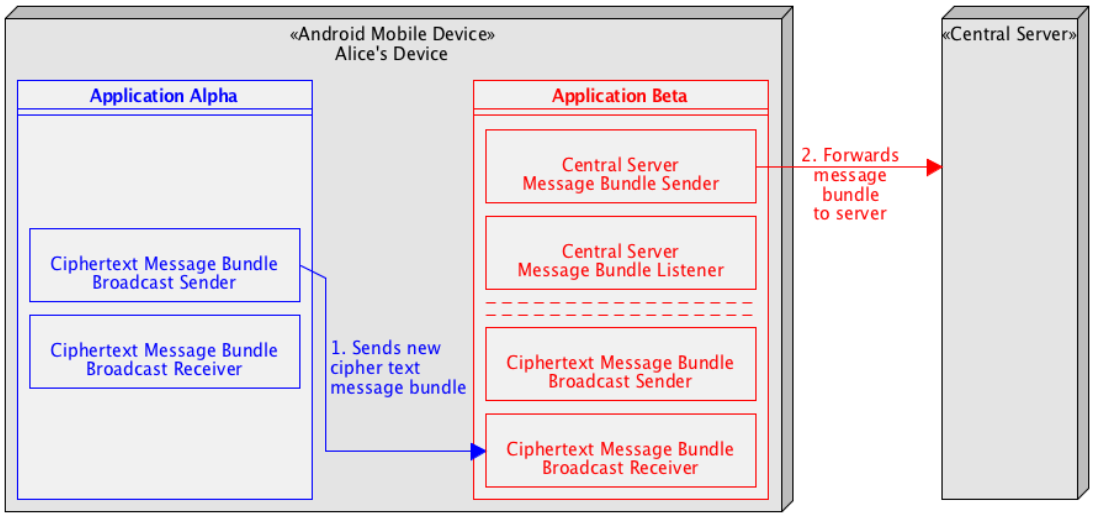

Figure 17 - IPC from Application Alpha to Application Beta

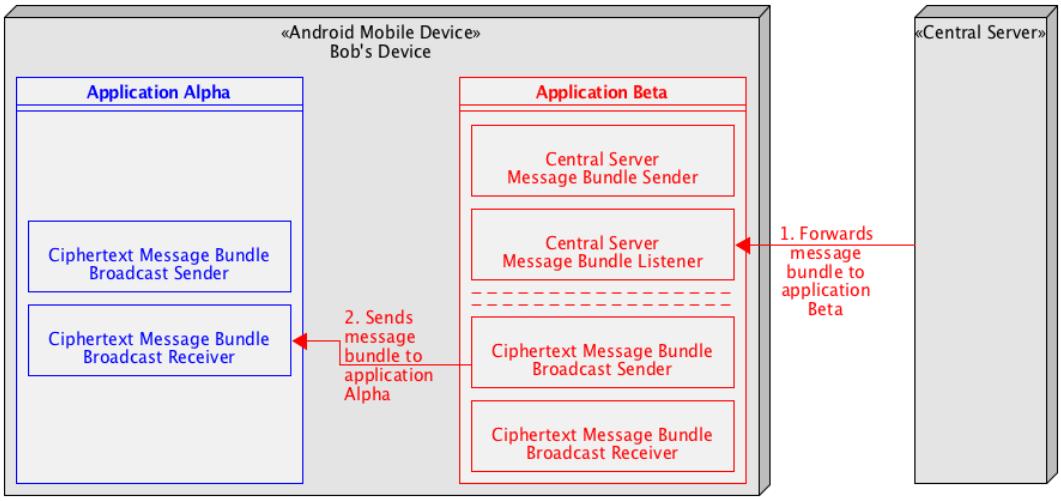

Figure 18 - IPC from Application Beta to Application Alpha

Figures 17 and 18 above involve five notable components:

1. **Cipher Text Message Bundle (CMB)**: The cipher text message bundle is a bundle that encapsulates the encrypted message itself, an HMAC value for integrity verification purposes, along with metadata that is necessary to transmit this message to the correct destination (e.g. conversation ID, recipient ID, etc.). The content of the metadata will depend on the specifications set by application Beta.
2. **CMB Broadcast Sender**: The CMB Broadcast Sender is used to send a message from one application to the other. For application Alpha to send a message to application Beta (or vice-versa), application Alpha must create an "intent" targeted at an action specified by application Beta (e.g. getEncryptedMessage), and attach to the intent the message content that it wants to send. Thereafter, application Alpha must utilize the built-in function sendBroadcast(Intent) [84] to instantly send that message to application Beta.
3. **CMB Broadcast Receiver**: The CMB Broadcast Receiver is used to listen to broadcasts sent by other applications. The broadcast receiver must specify an action to listen to (e.g.

getEncryptedMessage). Whenever an application sends a broadcast with this action specified by the broadcast receiver, the broadcast receiver is capable of picking up the broadcasted message, extracting its content and processing it.

4. **Central Server CMB Sender**: The Central Server CMB Sender is used by application Beta to forward CMBs to the central server.
5. **Central Server CMB Listener**: The Central Server CMB Listener is used by application Beta to listen to incoming CMBs from the central server.

5.3.3 Generic Aspect of the Proposed Design

In the same manner as the conventional design, our proposed design poses no restrictions with regard to which protocols the developers can implement and how they choose to implement them. In other words, developers can implement application Alpha, application Beta, and the central server, in whichever way they like. The proposed design however revolves around the concept of separating the security-related functionalities from the network-related functionalities, for the purpose of making it possible for the developers to release their security-related code (i.e. application Alpha) for independent review, without having to release their network-related code (i.e. application Beta). For this reason, it is crucial that all sensitive information (e.g. message content, cryptographic keys, sender and receiver information, etc.) remain strictly within application Alpha. Application Beta and the central server must strictly deal with encrypted messages along with the necessary accompanying metadata required to send the messages to the

correct destinations. Having said that, there are two principal forms of integrating application Alpha, application Beta, and the central server.

5.3.3.1 Integration #1: One-to-One Model

In the one-to-one model, a developer that wants to implement our proposed design must explicitly implement his/her unique application Alpha, application Beta, and required central server. The relationship between the implemented application Alpha and application Beta is such that, in terms of employing IPC mechanisms, application Alpha may only be linked to application Beta, and application Beta may only be linked to application Alpha. If a user downloads multiple applications that implement our proposed design, the user would have multiple combinations of application Alpha and application Beta instances. This is illustrated in figure 19 below.

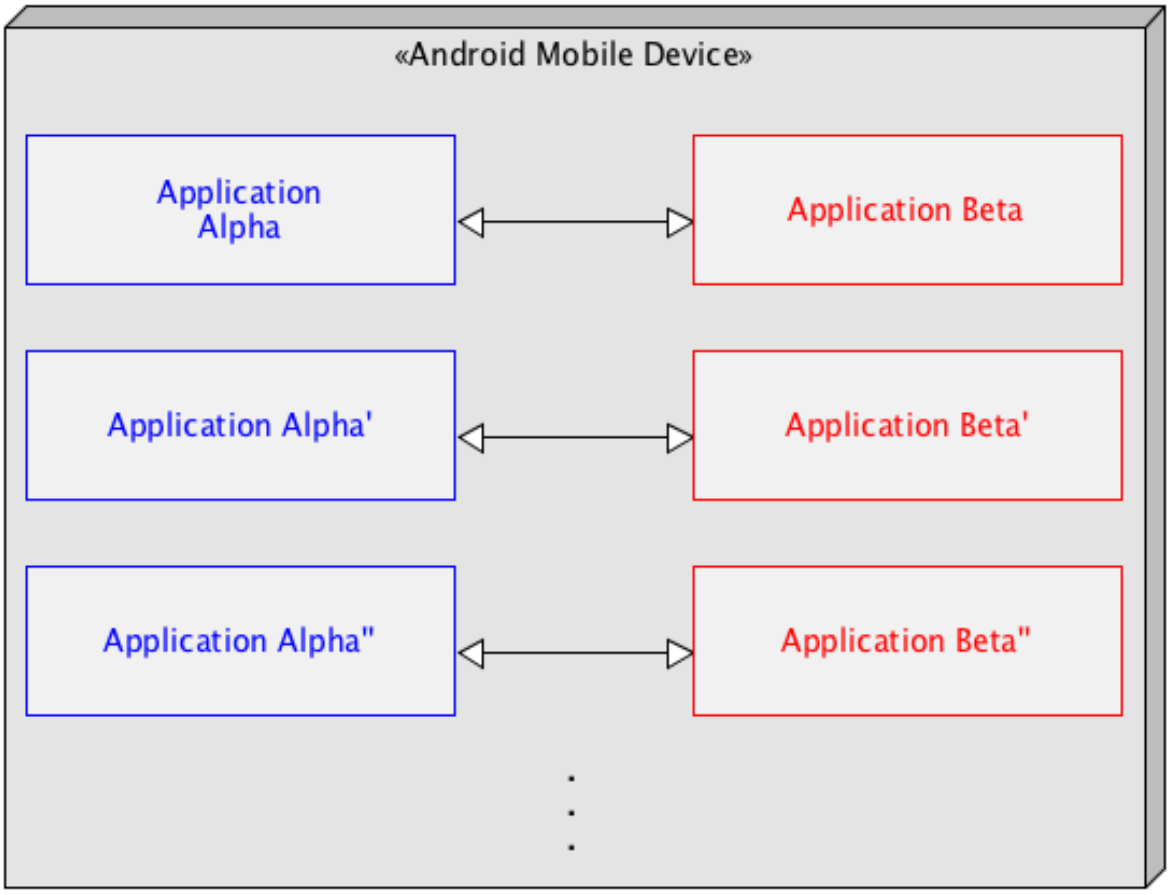

Figure 19 - One-to-One Relationship between Application Alpha and Application Beta Instances

5.3.3.2 Integration #2: One-to-Many Model

In the one-to-many model, a developer that wants to implement our proposed design can utilize the functionalities of an existing application Alpha. The developer would still need to implement his/her application Beta, and required central server. The relationship between the existing application Alpha and the newly implemented application Beta is such that application Alpha may be linked to multiple implementations of application Beta, and implementations of application Beta may only be linked to application Alpha. If a user downloads multiple applications

that implement our proposed design, the user would have one application Alpha and multiple implementations of application Beta. This is illustrated in figure 20 below.

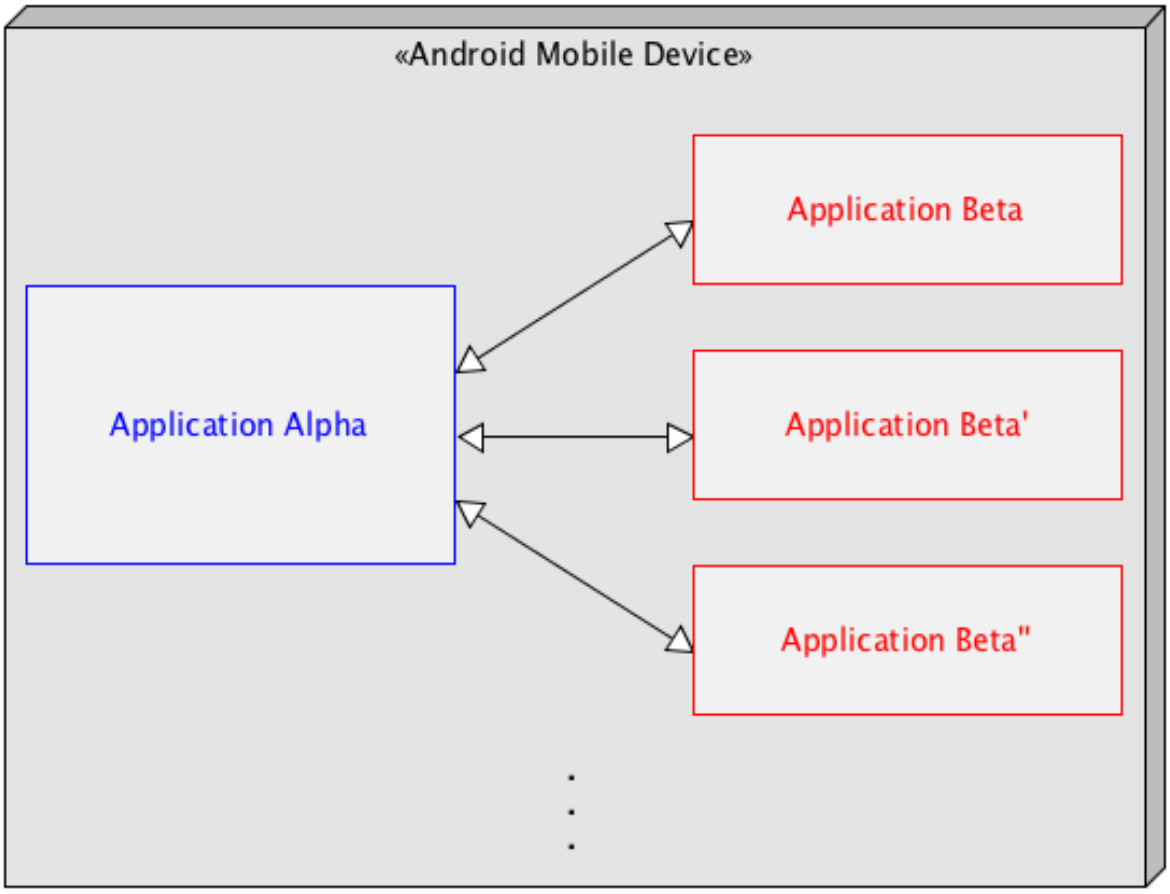

Figure 20 - One-to-Many Relationship between Application Alpha and Application Beta Instances

It is important to note that this model would require more work to be done for both application Alpha and application Beta. In addition to all the functionalities that it provides, application Alpha needs to (1) allow users to manage IPC mechanisms between application Alpha and different instances of application Beta, and (2) implement a flexible user interface that can tolerate sending different data elements to different application Beta instances. Essentially, the user

should be given the option in application Alpha to enable or disable IPC mechanisms between application Alpha and application Beta instances at any time. Moreover, when an instance of application Beta needs to communicate with application Alpha for the first time, the instance must request the permission to do so from the user, through application Alpha. This permission request should also include the different data elements (e.g. sender_id, receiver_id, current_time, etc.) that the IPC will incorporate. Figures 21, 22 and 23 show an example of a one-to-many system setup.

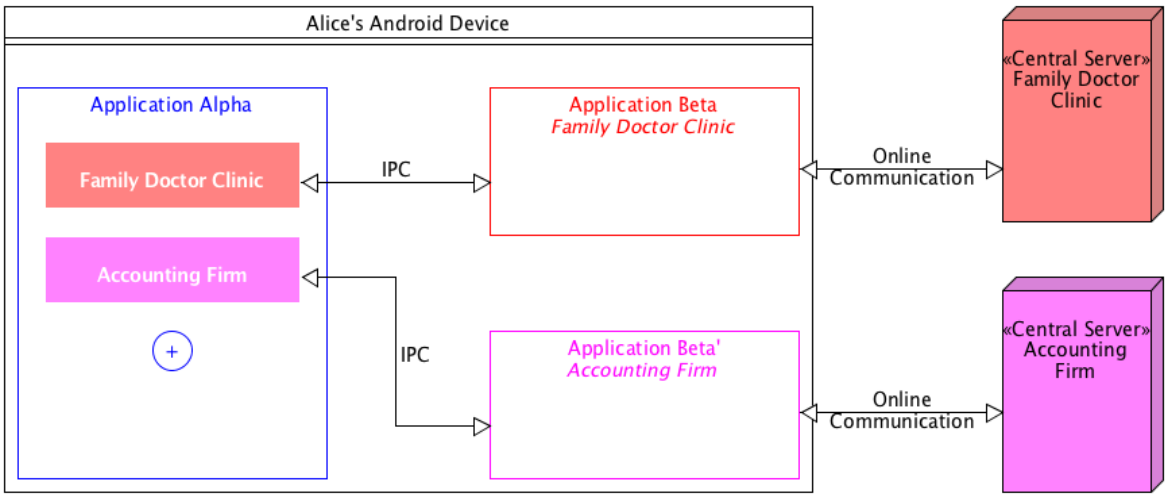

Figure 21 – One-to-Many System Setup: Overview

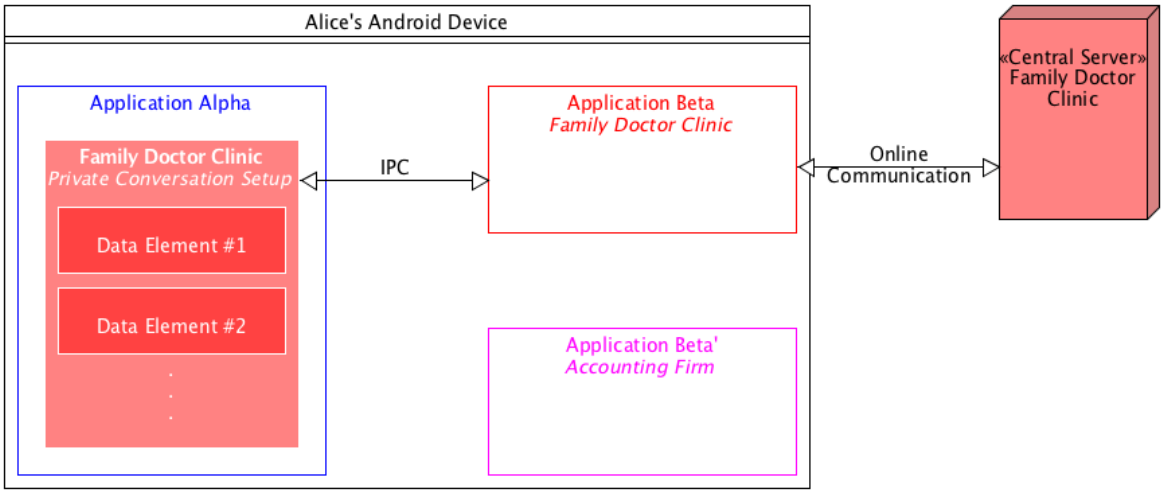

Figure 22 – One-to-Many System Setup: UI for Beta

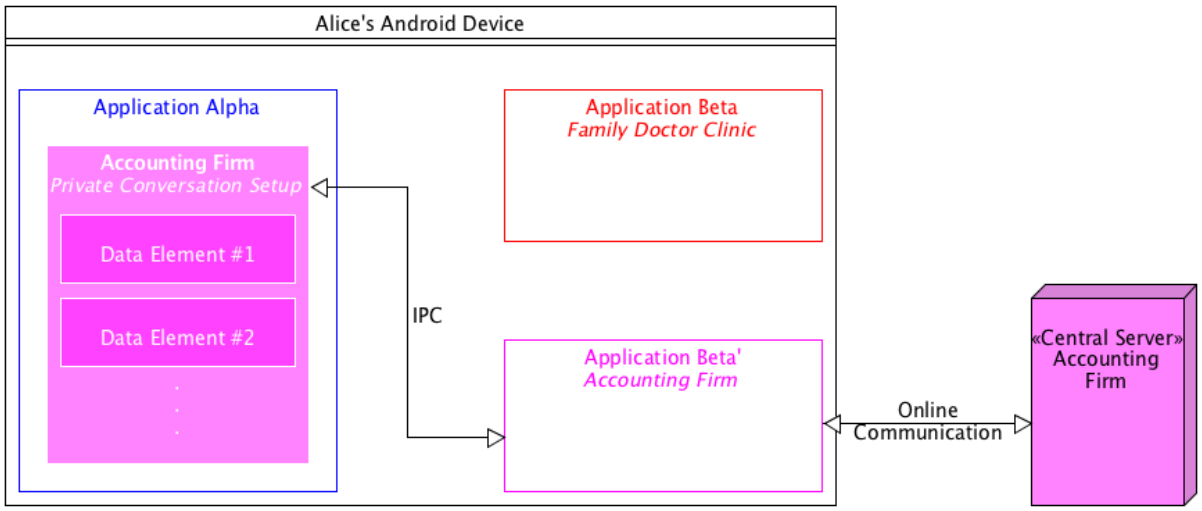

Figure 23 – One-to-Many System Setup: UI for Beta'

The IPC mechanisms would ideally consist of broadcast receivers and explicit broadcast sender intents. Basically, when a user wants to send encrypted data through a specific application Beta instance, the user must first specify in application Alpha the instance that he would like to send data to. Furthermore, the cipher text message bundle must specify an ID (e.g. package name) that represents the utilized application Beta instance. Decryption of the cipher text message bundle remains the same as before, with the addition of the information regarding the ID of the application Beta instance that sent the bundle. The user interface provided in application Alpha to communicate with different application Beta instances will have to be identical for all instances. The only difference between the user interface of different instances would be the data elements that are being displayed (e.g. application Beta might have 4 data elements, while application Beta' might have 6 data elements). This behavior can be achieved by utilizing a flexible data representation language (e.g. XML, JSON, etc.), and creating a user interface that can dynamically adjust to the data being represented by the chosen language. This model would not be ideal for applications that require a rich unique user interface; it is more ideal for scenarios that involve

simply sending encrypted messages from various locations (i.e. different central servers) to a single location (i.e. application Alpha).

5.3.4 Effect of the Design on Issues Related to E2EE Messaging Applications

5.3.4.1 Effect on the Trust-Related Issues

In section 5.2.1.1, we explained that due to the fact that the conventional design bundles all the groups of functionalities associated with different components together, rendering the code open for independent review would require the developers to expose their entire code base, including the code associated with their network-related functionalities which does not necessarily deal with sensitive information. To avoid this, E2EE messaging application developers would not publish their code at all for independent review. In our proposed design, we separate the security-related functionalities from the network-related functionalities required for E2EE messaging applications into two separate applications. We carry out this separation of functionalities in the interest of making it possible for application developers to render their security-related code open for independent review, without being forced to expose their network-related code. This goes hand-in-hand with criterion #5 from EFF's Secure Messaging Scorecard, the criterion that requires that the code for secure messaging applications be open for independent review. The results from the scorecard showed that out of the 37 listed applications, only 15 applications complied with that criterion. Our proposed design is meant to help applications bypass the roadblocks preventing them from rendering their security-related code open for independent review, especially if their roadblocks are not directly linked to the sensitive information the applications deal with.

It is extremely important to note that although having the code open for independent review is essential when it comes to E2EE messaging applications, this condition is in no way sufficient to consider such applications secure. The fact is that having the code open for independent review is simply a first step for evaluating the security of such systems, while maintaining the users' trust. It is imperative that these applications get subjected to continuous analyses, security testing, and independent security audits by security professionals.

5.3.4.2 Effect on the Protocol, Implementation, and Metadata Related Issues

Just like the conventional model, our proposed model also does not pose any restrictions with regard to which protocols the developers can implement and how they choose to implement them. However, by splitting an E2EE messaging application into two applications, we acquire a potential implementation-related security benefit due to the fact that we get to manage the privileges that are associated with each application separately. Application Alpha and application Beta can have two completely different sets of permissions, while still being capable of communicating with each other via IPC channels. For instance, application Alpha may require the permission to read and write data from/to the external storage, while there would be no reason for application Beta to be granted this permission. However, it is important to note that this security benefit would no longer apply if the Android operating system implements the same privilege requirement that we discussed in section 4.4.1, as both applications would be required to have the same sets of permissions in order to be allowed to communicate with each other.

When we consider the one-to-one model that we discussed in section 5.3.3.1, the conventional model and our proposed model are very similar with regard to how they affect the protocol, implementation and metadata-related issues. However, when we consider the one-to-many model that we discussed in section 5.3.3.2, we acquire a grand benefit: reusable security. Whether it be for key exchange (e.g. Diffie-Hellman), for encryption (e.g. AES, RSA), or for any other security-related functionality (e.g. SHA256, HMAC, etc.), there are certain cryptographic algorithms that are quite often implemented in the majority of E2EE messaging applications and that are accessed via trusted libraries (e.g. javax.crypto.cipher [94]). In the one-to-many model, application Alpha has the capacity to implement these commonly used algorithms, and the implementations can be utilized by different instances of application Beta. In other words, these security-related implementations become centralized and reusable within application Alpha. One of the main advantages offered by this centralized reusable security is that it reduces the repeated implementations of the same cryptographic primitives, and thus, as a result, there is less room for potential errors relating to the security-functionalities. On the other hand, it becomes even more critical for the functionalities implemented within application Alpha to be very well analyzed, tested, and subjected to security audits, since exploits of the reusable security may lead to more severe consequences on a larger scale.

5.3.5 Drawbacks with the Proposed Design

5.3.5.1 Implementation Overhead

In our proposed design, some of the groups of functionalities have to be implemented on both of the implicated applications (i.e. application Alpha and application Beta). Essentially, although application Alpha's main focus is to provide the security-related functionalities, application Alpha still has to implement its own group of UI-functionalities, and groups of other functionalities if required (e.g. database, cache, etc.). The same applies to application Beta, wherein the main focus is to provide the network-related functionalities; application Beta still has to implement its own group of UI functionalities and other functionalities if required. Furthermore, application Alpha and application Beta are both required to implement the IPC functionalities in order to be capable of exchanging messages with each other. This implementation overhead does not exist in the conventional design, wherein each group of functionalities only needs to be implemented once, and no IPC functionality may be required.

5.3.5.2 Negative Effect on Usability

In comparison to the conventional model, our proposed model poses a negative effect on the usability of an application, mainly due to the fact that our model implicates two separate applications, as opposed to the conventional model's singular application. To effectively determine how the usability of the applications that implement our proposed design compares to the usability of the applications that implement the conventional design, it would be important to perform usability studies (e.g. formal usability experiments, cognitive walkthroughs, etc.). These usability studies are not within the scope of this thesis.

5.3.6 Related Work

In this chapter, we propose the idea of splitting an E2EE messaging Android application into two separate applications in such a way that separates the security-related functionalities from the network-related functionalities. It is important to note that there are certain systems that share a similar architecture in comparison to our proposed design. The Secure Shell (SSH) protocol [95] and Proxos [96] are two examples of such systems.

The main objective of the SSH protocol is to "*secure remote login and other secure network services over an insecure network"* [95]. As explained in [95], the protocol does so by employing three principal components: (1) the *Transport Layer Protocol*, (2) the *User Authentication Protocol*, and (3) the *Connection Protocol*. The three protocols are combined together in a way such that the *User Authentication Protocol* runs on top of the *Transport Layer Protocol*, and the *Connection Protocol* runs on top of both the *User Authentication Protocol* and the *Transport Layer Protocol*. The *Transport Layer Protocol* takes care of server authentication, and provides users with important security properties such as confidentiality, integrity verification and forward secrecy [95]. The *User Authentication Protocol* takes care of authenticating a user on the client-side to the server. And finally, the Connection Protocol manages "*login sessions, remote execution of commands, forward TCP/IP connections, and forwarded X11 connections*" [97]. The SSH protocol was also designed to be extensible, allowing developers to select their own techniques for encryption, authentication, and key exchange [95]. If we compare the SSH protocol to our proposed design from an architectural perspective, we can see that the two are very similar. Both designs involve two or more components that must communicate with each other and split

functionalities in a way that the data is secured properly. However, the objective of the SSH protocol, which is to essentially provide a secure way to perform remote login and other network functionalities over an insecure network [95], is different than the main objective of our proposed design. In our design, the IPC is occurring locally on a user's device, within trusted IPC channels (i.e. on a trusted device), and the separation of security-functionalities from the network-related functionalities is strictly being done in the interest of making it possible for developers to render their security-related code open for independent review, without being forced to do the same for their network-related code. Moreover, in SSH, the communication that occurs between a client and server is encrypted, however, the server must still implement certain security functionalities in order to be capable of decrypting cipher text in order to execute the appropriate commands or to perform certain other functionalities that are requested by the client. In the case of our proposed design, if we consider that application Alpha is a client and application Beta is a server, then unlike SSH, the server (i.e. application Beta) is never capable of decrypting cipher text that is incoming from the client (i.e. application Alpha); the sole purpose of the server is to relay messages from Application Alpha to the online central server, and the other way around.

Another system that shares a similar architecture to our proposed design is Proxos [96]. As explained in [96], Proxos is a system that "*allows applications to configure their trust in the OS by partitioning the system call interface into trusted and untrusted components"*. Ta-Min et al. explain that some commodity operating systems provide an overly permissive interface to applications, and as a result, privileged applications have the potential to abuse their granted permissions and make the kernel access data or modify the states of other applications. The authors then explain that due to this problem, applications that require the execution of sensitive operations

should not completely trust the kernel. The Proxos system, as explained in [96], was designed in such a way that application developers are required to partition their code into components that fall in two main categories: (1) components that trust the commodity operating system, and (2) components that do not trust the commodity OS. Thereafter, by running the commodity operating system along with other private operating systems on a virtual machine manager (VMM), the Proxos system serves as a "*thin operating system proxy*" [96], and directs non-sensitive system calls to the commodity OS, while sensitive system calls get directed to the appropriate private OS. The architecture of the Proxos system is similar to our proposed design, as both designs require the separation of functionalities. In Proxos, sensitive operations are separated from non-sensitive operations and are executed in a private OS in a safe manner. This separation of functionalities is occurring due to the fact that the corresponding commodity OS cannot necessarily be trusted. In our proposed design, sensitive operations are all done in application Alpha, while application Beta strictly deals with message relay. By having the operations that deal with sensitive data take place on application Alpha, we no longer require that the code for application Beta be open for independent review. Having the code of application Alpha open for independent review would be sufficient for opening room for security experts to perform security audits and effectively evaluate the security of the overall application.

5.4 Proof of Concept #2: End-to-End Encrypted Messaging Application

5.4.1 Introduction

To demonstrate how our proposed design can be put to use in a real-world E2EE messaging application, we created and implemented a one-to-one configuration of our system, which included an application Alpha, application Beta, and a central server. For our central server, we utilized the Firebase Cloud Messaging service available through Google's Firebase mobile platform [98, 99], which provided us with the functionality required to relay messages from one entity to another. It is important to note that the applications that we will be discussing in this section were strictly built to demonstrate how our design can be put to practice. The applications apply basic cryptographic techniques to achieve E2EE messaging. The applications however did not get subjected to any form of security testing or forensic analyses and should not be considered secure. In this section, we will briefly describe the applications that we built and how they interact with each other to achieve end-to-end encrypted messaging. More details concerning the design and implementation of these applications can be found in Appendix B.

5.4.2 System Overview

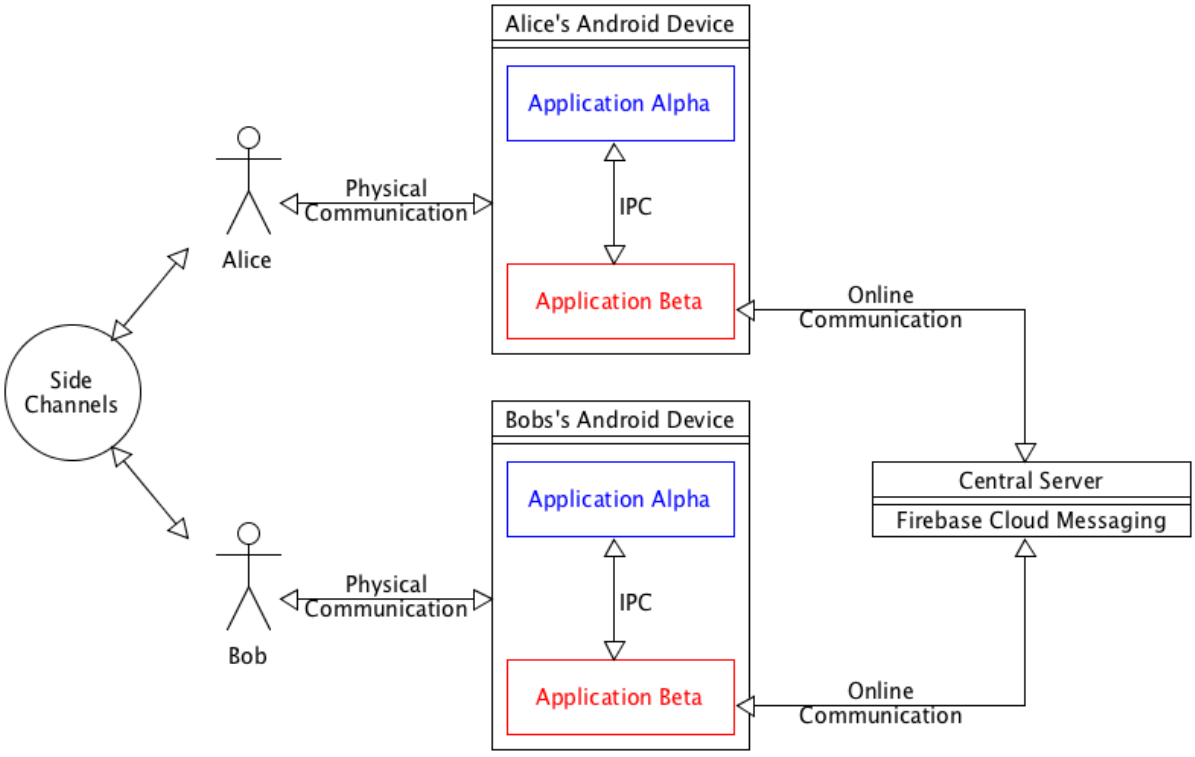

Figure 24 - System Overview

Figure 24 above displays an overview of the system that we built. Essentially, in order for Alice and Bob to securely communicate with each other using our system, they must both deploy our instances of application Alpha and application Beta on their device.

5.4.2.1 Application Alpha

Our configuration of application Alpha allows users to perform the following actions:

1. Create a new conversation

2. Import an existing conversation
3. Share conversation setups with other users
4. Send/Receive encrypted messages

In order for two users to communicate with each other, both users must have the same conversation setup on their phones. For instance, in order for Alice to communicate with Bob, Alice must create a new conversation, and then share this new conversation setup with Bob (through the use of side channels). When Bob receives the conversation setup that is sent by Alice, Bob imports that conversation in his instance of application Alpha. Once Alice and Bob have the same conversation setup on their device, they can start communicating with each other. In our proof of concept, we utilized AES-256 for encryption, and HMAC-SHA256 for integrity verification. To achieve IPC, application Alpha had to implement broadcast sender intents and a broadcast receiver. Essentially, whenever a user inputs a text into application Alpha and clicks on the "ENCRYPT AND SEND" button, the plain text gets encrypted in the background, a cipher text message bundle gets prepared, an HMAC value gets computed for integrity verification purposes, and this entire bundle gets sent to application Beta through an explicit broadcast sender intent. On the other hand, whenever application Alpha receives a new incoming cipher text message bundle, it initially re-computes the HMAC value to verify the integrity of the content that was received. If the HMAC value is valid, application Alpha decrypts the cipher text message bundle and displays the plaintext to the user. Figure 25 shows certain screenshots of application Alpha. More screenshots of the application and more details regarding the protocol that we implemented can be found in Appendix B.

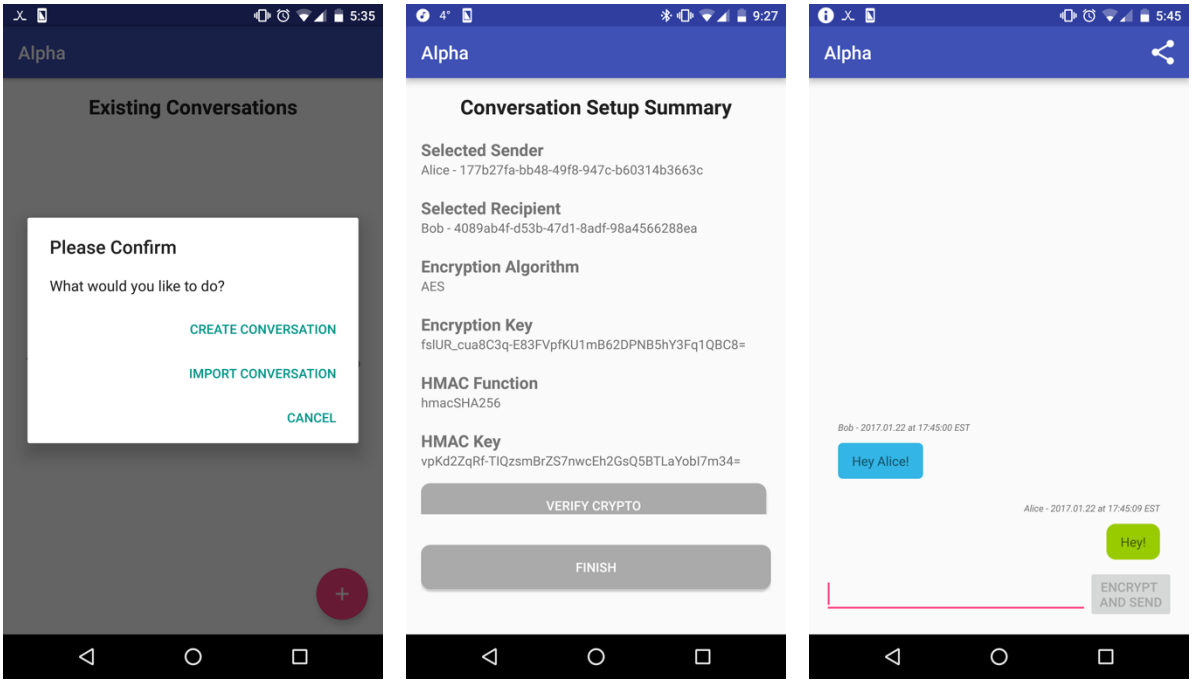

Figure 25 - Screenshots of Application Alpha

5.4.2.2 Application Beta

In our proof of concept, application Beta strictly deals with sending and receiving data to/from the central server and to/from application Alpha. Essentially, when application Beta's broadcast receiver receives a cipher text message bundle from application Alpha, application Beta sends that bundle to our server. On the other hand, when application Beta receives a cipher text message bundle from the central server, it sends it to application Alpha via an explicit broadcast sender intent. The functionalities required for communication with the central server and for IPC are always executed in the background, the user only needs to open application Beta upon the initial installation in order to get the connection to the Firebase server automatically set up. If a user chooses to open application Beta, he will simply see a stream of the cipher text message bundles sent to the application. A screenshot of application Beta is shown in figure 26 below.

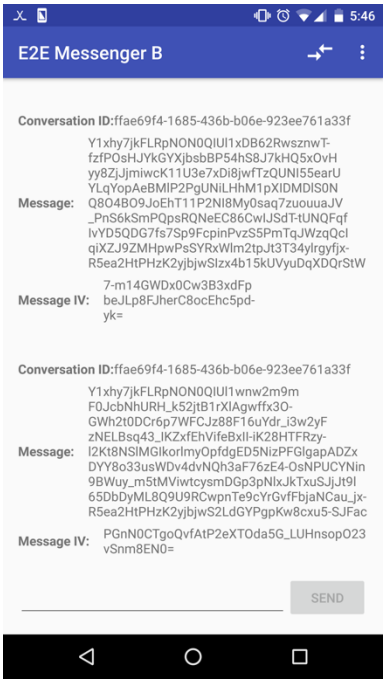

Figure 26 - Screenshot of Application Beta

5.4.2.3 Central Server

As we already mentioned, we used the Firebase mobile platform for our central server. Our Firebase server takes of care of relaying messages from one entity to another. In other words, the server listens to new messages incoming from application Beta instances, and then forwards these messages to other application Beta instances.

5.5 Conclusion

In this chapter, we described the conventional design that existing E2EE messaging applications typically implement on the Android operating system. We discussed the effect that

this conventional design has on the common issues related to E2EE messaging applications. We then proposed a new design that separates the security-related functionalities from the network-related functionalities in E2EE messaging applications, in the interest of making it possible for developers to render their security-related code open for independent review, without being forced to do the same for their network-related code. Our proposed design is possible thanks to the IPC mechanisms provided by the Android operating system. Finally, we demonstrated how our proposed design can be put to use through a proof of concept wherein we built a system that consisted of two Android applications and a central server.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

Whether it be from the Google Play Store or from other sources, there are numerous Android applications that users can download, and often times, these applications have access to some of the users' sensitive data. This sensitive data may include SMS/MMS messages, phone call logs, GPS locations, photos and videos, etc. The Android operating system has several built-in mechanisms that help secure users' data. Additionally, there are various applications available on the Google Play Store that help users preserve their privacy on their devices. In this thesis, we initially examined the architecture underlying the Android operating system. We looked into the principal components that typically constitute Android applications, along with the different security features that are offered by or within the operating system. Moreover, we examined existing Android E2EE messaging applications, and we derived four categories of issues that are commonly associated with them. The four categories consisted of (1) trust-related issues, (2) protocol-related issues, (3) implementation-related issues and (4) metadata-related issues.

In this thesis, we provided an overview of the known issue of privilege escalation whereby an unauthorized application on a user's device may access protected data from a malicious privileged application by employing IPC techniques. We explained the importance of this issue, and demonstrated through a proof of concept how it can be done in practice. We also listed and discussed possible countermeasures that can help prevent or reduce the impact caused by this issue.

Furthermore, in the interest of minimizing the common issues that exist in today's E2EE messaging applications, we proposed a new design for the latter that employs some of the security features that we studied. We showed that our design mainly helps eliminate the trust-related issues, however depending on the selected configuration, it can help minimize issues in the other categories as well. We explained in detail how our design works, how it can benefit the users, and how it differs from the conventional design that is typically implemented by E2EE messaging applications. We also demonstrated through a proof of concept how our design can be used in practice.

6.2 Future Work

In Chapter 4, we listed three potential countermeasures that can help prevent or reduce the impact caused by the issue of unauthorized data access that we discussed. The countermeasures that we presented all required modifications to the Android operating system. In future work, the effectiveness and feasibility of these countermeasure techniques should be studied further.

To prove that the design that we proposed in Chapter 5 works, we implemented a proof of concept system with a certain configuration. Although the system that we built works as intended, the system uses basic cryptographic techniques to achieve E2EE messaging, and it should not be considered secure. In future work, it would be a good idea to build a fully-secure system that implements our proposed design. As we explained in Chapter 5, that system would require the configuration and implementation of techniques for authentication, key management, session management, encryption/decryption, integrity verification, and digital signing. Moreover, the

system that we built in our proof of concept implements the one-to-one model. It would also be useful to create a configuration of our one-to-many model, and to demonstrate how it would be possible to set up a reusable application Alpha that is linked to multiple instances of application Beta. Additionally, it is important to note that whenever it comes to real-time applications, performance is very important. Although the system that we built in our proof of concept functioned without any noticeable latency, our system was not designed to handle heavy traffic. It would be a good idea to evaluate and optimize the communications that are happening between application Alpha and application Beta, as well as between application Beta and the central server.

References

[1] Lockheimer, H. (2015, September 29). S'more to love across all your screens. Retrieved August 17, 2016, from https://googleblog.blogspot.com/2015/09/smore-to-love-across-all-your-screens.html

[2] Pichai, S. (2015, May 28). You say you want a mobile revolution... Retrieved August 17, 2016, from https://googleblog.blogspot.ca/2015/05/io-2015-mobile-revolution.html

[3] Smith, C. (2017, January 21). 108 Amazing Android Statistics. Retrieved August 17, 2016, from http://expandedramblings.com/index.php/android-statistics/

[4] Android Auto. (n.d.). Retrieved August 17, 2016, from https://www.android.com/intl/en_ca/auto/

[5] Number of available Android applications. (2016, December 1). Retrieved December 1, 2016, from http://www.appbrain.com/stats/number-of-android-apps

[6] Edwards, L. (2014, August 14). How do companies make money from your data? Retrieved December 4, 2016, from http://www.pocket-lint.com/news/130366-how-do-companies-make-money-from-your-data

[7] All About Computer Cookies - privacy concerns on cookies. (n.d.). Retrieved December 4, 2016, from http://www.allaboutcookies.org/privacy-concerns/

[8] Baker, N. (2014, November 11). New apps meet need for more privacy after over-sharing. Retrieved January 30, 2017, from http://www.reuters.com/article/apps-privacy-idUSL1N0SX37G20141111

[9] Hughes, S., & Johnson, B. (2016, November 16). Encryption apps see growth after election. Retrieved January 30, 2017, from http://www.marketplace.org/2016/11/15/world/encryption-app-signal-sees-400-growth-election

[10] Lee, M. (2016, June 22). Battle of the Secure Messaging Apps: How Signal Beats WhatsApp. Retrieved February 11, 2017, from https://theintercept.com/2016/06/22/battle-of-the-secure-messaging-apps-how-signal-beats-whatsapp/

[11] Platform Architecture. (n.d.). Retrieved December 6, 2016, from https://developer.android.com/guide/platform/index.html

[12] ART and Dalvik. (n.d.). Retrieved November 30, 2016, from https://source.android.com/devices/tech/dalvik/index.html

[13] An Overview of the Android Architecture. (n.d.). Retrieved December 6, 2016, from http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture

[14] Security. (n.d.). Retrieved September 7, 2016, from https://source.android.com/security/

[15] Activities. (n.d.). Retrieved December 13, 2016, from https://developer.android.com/guide/components/activities/index.html

[16] The Activity Lifecycle. (n.d.). Retrieved June 30, 2017, from https://developer.android.com/guide/components/activities/activity-lifecycle.html

[17] Service. (n.d.). Retrieved December 13, 2016, from https://developer.android.com/reference/android/app/Service.html

[18] AsyncTask. (n.d.). Retrieved December 13, 2016, from https://developer.android.com/reference/android/os/AsyncTask.html

[19] BroadcastReceiver. (n.d.). Retrieved December 15, 2016, from https://developer.android.com/reference/android/content/BroadcastReceiver.html

[20] Content Provider Basics. (n.d.). Retrieved March 06, 2017, from https://developer.android.com/guide/topics/providers/content-provider-basics.html

[21] Content Providers. (n.d.). Retrieved March 06, 2017, from https://developer.android.com/guide/topics/providers/content-providers.html

[22] Intent. (n.d.). Retrieved December 12, 2016, from https://developer.android.com/reference/android/content/Intent.html

[23] Intents and Intent Filters. (n.d.). Retrieved March 15, 2017, from https://developer.android.com/guide/components/intents-filters.html

[24] System and kernel security. (n.d.). Retrieved September 8, 2016, from https://source.android.com/security/overview/kernel-security.html

[25] MFILLPOT. (2010, May 18). Understanding Linux File Permissions. Retrieved March 06, 2017, from https://www.linux.com/learn/understanding-linux-file-permissions

[26] Requesting Permissions. (n.d.). Retrieved February 22, 2017, from https://developer.android.com/guide/topics/permissions/requesting.html

[27] Normal Permissions. (n.d.). Retrieved March 06, 2017, from https://developer.android.com/guide/topics/permissions/normal-permissions.html

[28] Requesting Permissions at Run Time. (n.d.). Retrieved December 8, 2016, from https://developer.android.com/training/permissions/requesting.html

[29] Security Tips. (n.d.). Retrieved December 12, 2016, from https://developer.android.com/training/articles/security-tips.html#IPC

[30] Android Keystore System. (n.d.). Retrieved March 09, 2017, from https://developer.android.com/training/articles/keystore.html

[31] Encryption. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/encryption/index.html

[32] Full-Disk Encryption. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/encryption/full-disk.html

[33] File-Based Encryption. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/encryption/file-based.html

[34] Authentication. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/authentication/index.html

[35] Gatekeeper. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/authentication/gatekeeper.html

[36] Fingerprint HAL. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/authentication/fingerprint-hal.html

[37] Application Signing. (n.d.). Retrieved March 09, 2017, from https://source.android.com/security/apksigning/index.html

[38] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., & Wagner, D. (2012, July). Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security* (p. 3). ACM.

[39] Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011, October). Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (pp. 627-638). ACM.

[40] Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012, October). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 217-228). ACM.

[41] Davi, L., Dmitrienko, A., Sadeghi, A. R., & Winandy, M. (2010, October). Privilege escalation attacks on android. In *International Conference on Information Security* (pp. 346-360). Springer Berlin Heidelberg.

[42] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., & Chin, E. (2011, August). Permission Re-Delegation: Attacks and Defenses. In USENIX Security Symposium (Vol. 30).

[43] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., & Wallach, D. S. (2011, August). QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In USENIX Security Symposium (Vol. 31).

[44] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., & Sadeghi, A. R. (2011). Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04.

[45] Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011, June). Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (pp. 239-252). ACM.

[46] Keep your online communication private. (n.d.). Retrieved January 31, 2017, from https://securityinabox.org/en/guide/secure-communication/

[47] Hughes, S., & Johnson, B. (2016, November 16). Encryption apps see growth after election. Retrieved January 30, 2017, from http://www.marketplace.org/2016/11/15/world/encryption-app-signal-sees-400-growth-election

[48] WhatsApp Messenger - Android Apps on Google Play. (2017, March 20). Retrieved March 29, 2017, from https://play.google.com/store/apps/details?id=com.whatsapp&hl=en

[49] Messenger - Android Apps on Google Play. (2017, March 29). Retrieved March 29, 2017, from https://play.google.com/store/apps/details?id=com.facebook.orca&hl=en

[50] Telegram - Android Apps on Google Play. (2017, March 03). Retrieved March 29, 2017, from https://play.google.com/store/apps/details?id=org.telegram.messenger&hl=en

[51] Google Allo - Android Apps on Google Play. (2017, March 20). Retrieved March 30, 2017, from https://play.google.com/store/apps/details?id=com.google.android.apps.fireball&hl=en

[52] Signal Private Messenger - Android Apps on Google Play. (2017, March 24). Retrieved March 30, 2017, from https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms&hl=en

[53] Wickr Me - Secure Messenger - Android Apps on Google Play. (2016, March 31). Retrieved March 30, 2017, from https://play.google.com/store/apps/details?id=com.mywickr.wickr2&hl=en

[54] The Guardian Project - September 27, 2016 - Everyone. (2016, September 27). ChatSecure - Android Apps on Google Play. Retrieved December 21, 2016, from https://play.google.com/store/apps/details?id=info.guardianproject.otr.app.im&hl=en

[55] The CIA principle. (n.d.). Retrieved January 31, 2017, from http://www.doc.ic.ac.uk/~ajs300/security/CIA.htm

[56] Chia, T. (2012, August 20). Confidentiality, Integrity, Availability: The three components of the CIA Triad. Retrieved February 16, 2017, from http://security.blogoverflow.com/2012/08/confidentiality-integrity-availability-the-three-components-of-the-cia-triad/

[57] Gibson, D. (2011, May 27). Understanding The Security Triad (Confidentiality, Integrity, and Availability). Retrieved February 6, 2017, from http://www.pearsonitcertification.com/articles/article.aspx?p=1708668

[58] Barker, E. (2016). Recommendation for Key Management Part 1: General. *NIST Special Publication 800 - 57 Part 1 Revision 4*. Retrieved March 30, 2017, from http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

[59] Authenticity (Information Security). (n.d.). Retrieved February 6, 2017, from https://www.privacycommission.be/en/glossary/authenticity

[60] Helme, S. (2014, May 10). Perfect Forward Secrecy - An Introduction. Retrieved February 6, 2017, from https://scotthelme.co.uk/perfect-forward-secrecy/

[61] Frosch, T., Mainka, C., Bader, C., Bergsma, F., Schwenk, J., & Holz, T. (2016, March). How Secure is TextSecure?. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on* (pp. 457-472). IEEE.

[62] Marlinspike, M. (2013, November 26). Advanced cryptographic ratcheting. Retrieved March 28, 2017, from https://whispersystems.org/blog/advanced-ratcheting/

[63] What is deniable encryption? (n.d.). Retrieved February 6, 2017, from http://searchsecurity.techtarget.com/definition/deniable-encryption

[64] Secure Messaging Scorecard. (2014, November 6). Retrieved February 16, 2017, from https://www.eff.org/node/82654

[65] Check Back Soon: New Secure Messaging Guide On the Way! (n.d.). Retrieved February 7, 2017, from https://www.eff.org/secure-messaging-scorecard

[66] Jakobsen, J., & Orlandi, C. (2016, October). On the CCA (in) security of MTProto. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices* (pp. 113-116). ACM.

[67] Anglano, C., Canonico, M., & Guazzone, M. (2016). Forensic analysis of the ChatSecure instant messaging application on android smartphones. *Digital Investigation*, *19*, 44-59.

[68] Ntantogian, C., Apostolopoulos, D., Marinakis, G., & Xenakis, C. (2014). Evaluating the privacy of Android mobile applications under forensic analysis. *Computers & Security*, *42*, 66-76.

[69] Diffie, W., van Oorschot, P. C., & Wiener, M. J. (1992). Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, *2*(2), 107-125.

[70] CacheWord: Passphrase Caching and Management. (n.d.). Retrieved December 21, 2016, from https://guardianproject.info/code/cacheword/

[71] LiME. (2017, January 25). Retrieved February 16, 2017, from https://github.com/504ensicsLabs/LiME

[72] Sylve, J., Case, A., Marziale, L., & Richard, G. G. (2012). Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, *8*(3), 175-184.

[73] Dezfouli, F. N., Dehghantanha, A., Mahmoud, R., Sani, N. F. B. M., & bin Shamsuddin, S. (2012, June). Volatile memory acquisition using backup for forensic investigation. In *Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on* (pp. 186-189). IEEE.

[74] What is metadata? (n.d.). Retrieved February 8, 2017, from http://whatis.techtarget.com/definition/metadata

[75] Conti, M., Mancini, L. V., Spolaor, R., & Verde, N. V. (2016). Analyzing android encrypted network traffic to identify user actions. *IEEE Transactions on Information Forensics and Security*, *11*(1), 114-125.

[76] Saltaformaggio, B., Choi, H., Johnson, K., Kwon, Y., Zhang, Q., Zhang, X., ... & Qian, J. (2016, August). Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic. In *Proc. USENIX Workshop on Offensive Technologies (WOOT'16, in conjunction with Security'16)*.

[77] Coull, S. E., & Dyer, K. P. (2014). Traffic analysis of encrypted messaging services: Apple imessage and beyond. *ACM SIGCOMM Computer Communication Review*, *44*(5), 5-11.

[78] Dyer, K. P., Coull, S. E., Ristenpart, T., & Shrimpton, T. (2012, May). Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on* (pp. 332-346). IEEE.

[79] Eckersley, P. (2014, November 8). What Makes a Good Security Audit? Retrieved February 9, 2017, from https://www.eff.org/deeplinks/2014/11/what-makes-good-security-audit

[80] Schneier, B. (2015, May 12). Amateurs Produce Amateur Cryptography. Retrieved February 10, 2017, from https://www.schneier.com/blog/archives/2015/05/amateurs_produc.html

[81] Schneier, B. (1998, October 15). Memo to the Amateur Cipher Designer. Retrieved February 10, 2017, from https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign

[82] Mayer, J., Mutchler, P., & Mitchell, J. C. (2016). Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences*, 201508081.

[83] Crawford, D. (2016, April 7). WhatsApp Tracks Metadata (other security issues). Retrieved February 16, 2017, from https://www.bestvpn.com/blog/47342/whatsapp-still-tracks-metadata/

[84] Context. (n.d.). Retrieved February 14, 2017, from https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent)

[85] MLS Introduction | Cryptosmith. (n.d.). Retrieved July 06, 2017, from https://cryptosmith.com/mls/intro/

[86] App Manifest. (n.d.). Retrieved December 13, 2016, from https://developer.android.com/guide/topics/manifest/manifest-intro.html

[87] Introduction to Activities. (n.d.). Retrieved March 20, 2017, from https://developer.android.com/guide/components/activities/intro-activities.html#da

[88] Services. (n.d.). Retrieved March 20, 2017, from https://developer.android.com/guide/components/services.html#Declaring

[89] Provider Element. (n.d.). Retrieved March 20, 2017, from https://developer.android.com/guide/topics/manifest/provider-element.html

[90] Broadcasts. (n.d.). Retrieved March 20, 2017, from https://developer.android.com/guide/components/broadcasts.html#system_broadcasts

[91] PackageManager. (n.d.). Retrieved March 23, 2017, from https://developer.android.com/reference/android/content/pm/PackageManager.html

[92] PackageInfo. (n.d.). Retrieved March 23, 2017, from https://developer.android.com/reference/android/content/pm/PackageInfo.html

[93] Guide to Cryptography. (2015, September 12). Retrieved February 16, 2017, from https://www.owasp.org/index.php/Guide_to_Cryptography#Cryptographic_Functions

[94] Cipher (Java Platform SE 7). (n.d.). Retrieved July 05, 2017, from https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html

[95] Ylonen, T., & C. Lonvick, Ed. (2006). The Secure Shell (SSH) Protocol Architecture. Retrieved July 03, 2017, from https://www.ietf.org/rfc/rfc4251.txt

[96] Ta-Min, R., Litty, L., & Lie, D. (2006, November). Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation* (pp. 279-292). USENIX Association.

[97] Ylonen, T., & C. Lonvick, Ed. (2006). The Secure Shell (SSH) Connection Protocol. Retrieved July 03, 2017, from https://tools.ietf.org/html/rfc4254

[98] App success made simple. (n.d.). Retrieved February 15, 2017, from https://firebase.google.com/

[99] Firebase Cloud Messaging | Firebase. (n.d.). Retrieved February 15, 2017, from https://firebase.google.com/docs/cloud-messaging/

[100] Java™ Cryptography Architecture Standard Algorithm Name Documentation. (n.d.). Retrieved March 21, 2017, from http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#SecureRandom

Appendix A

A.1. Introduction

In Chapter 4, we provided an overview of the issue wherein a privileged application that is deployed on a user's Android device can send protected data to an unauthorized application by employing IPC mechanisms. To demonstrate how this behavior can be achieved on an Android device, we created two Android applications: Application #1 and Application #2. In Chapter 4, we also listed potential countermeasure techniques that can help prevent or raise user awareness regarding occurrences of this issue. One of our countermeasure techniques required making changes to the operating system in such a way that the IPC mechanisms become more transparent. To show how much information users already have access to regarding the IPC mechanisms employed by applications on their device, we created another application, which we refer to as the "Android Manifest Explorer". In this appendix, we provide more details concerning the design and implementation of Application #1, Application #2, and the Android Manifest Explorer. All applications were designed and tested on a stock Android device, with version 7.1.1 (Nougat).

A.2. Application #1 and Application #2

A.2.1. System Overview

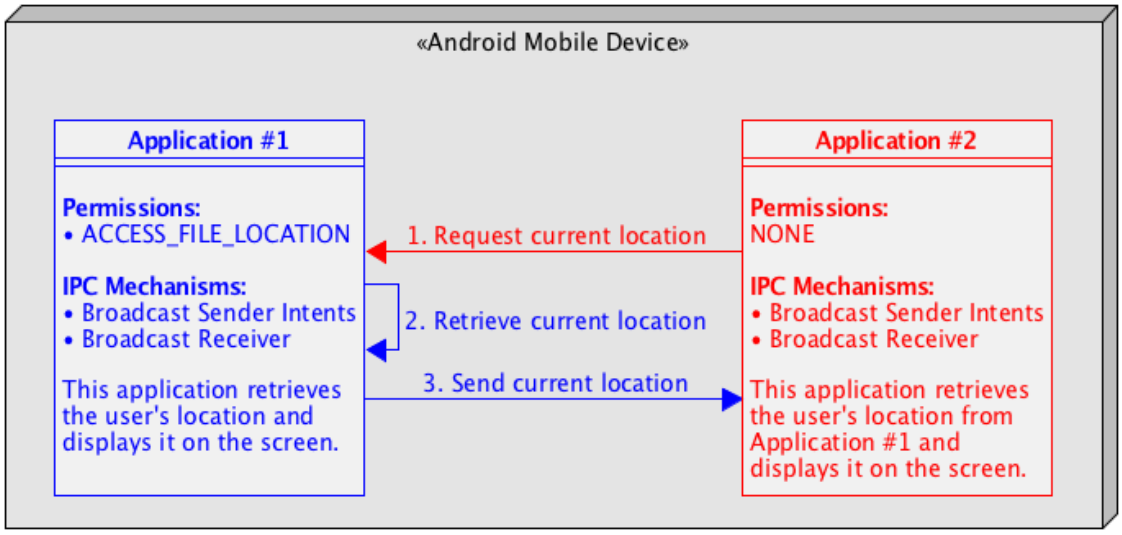

Figure 27 - System Overview: Application #1 and Application #2

Figure 27 above displays an overview of the system that we built. In terms of permissions, application #1 is granted only one permission that is entitled "`ACCESS_FILE_LOCATION`". Application #2 on the other hand is not granted any permissions. In other words, application #1 is a privileged application that has access to the user's current location, while application #2 is unprivileged and does not have access to this data. However, application #2 can retrieve this data from application #1 by employing IPC mechanisms.

In terms of IPC mechanisms, application #1 and application #2 both implement broadcast receivers and broadcast sender intents. The broadcast receivers must specify an action to listen to, and the broadcast sender intents must specify an action to target. In our case, we named this action "getLocationData" on all ends (i.e. both broadcast receivers and both broadcast sender intents).

Basically, in order for application #2 to retrieve the user's current location from application #1, the following sequence of events occurs:

1. Application #2 prepares and sends a broadcast sender intent to application #1, requesting the user's current location.
2. Application #1's broadcast receiver picks up that request. Application #1 launches a service to retrieve the user's current location.
3. Application #1 prepares and sends a broadcast sender intent to application #2. The broadcast sender intent will contain the user's current location.

In terms of activities, application #1 and application #2 each have only one activity. The activity in application #1 simply prints the user's current location. The activity in application #2 presents the user with a button labelled "Get User's Location". Once the user clicks on that button, the IPC mechanisms get triggered, and application #2 gains access to the user's current location and displays it. Screenshots of application #1 and application #2 are shown in figures 28 and 29 below, respectively.

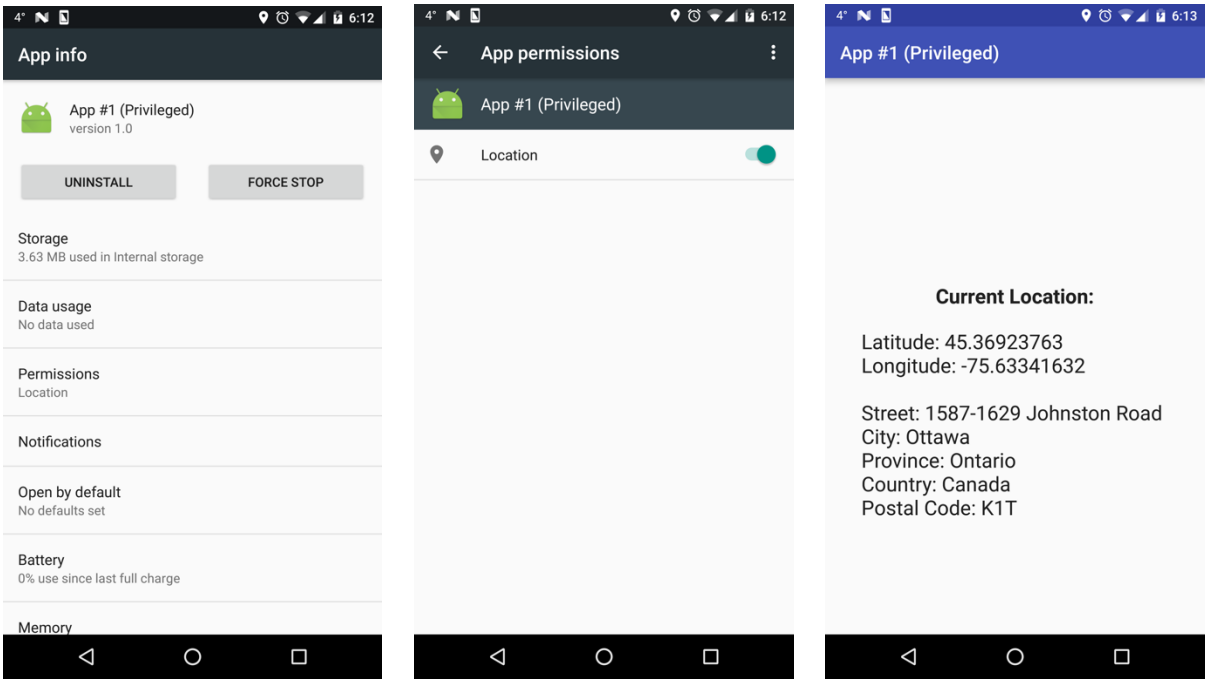

Figure 28 - Screenshots of Application #1

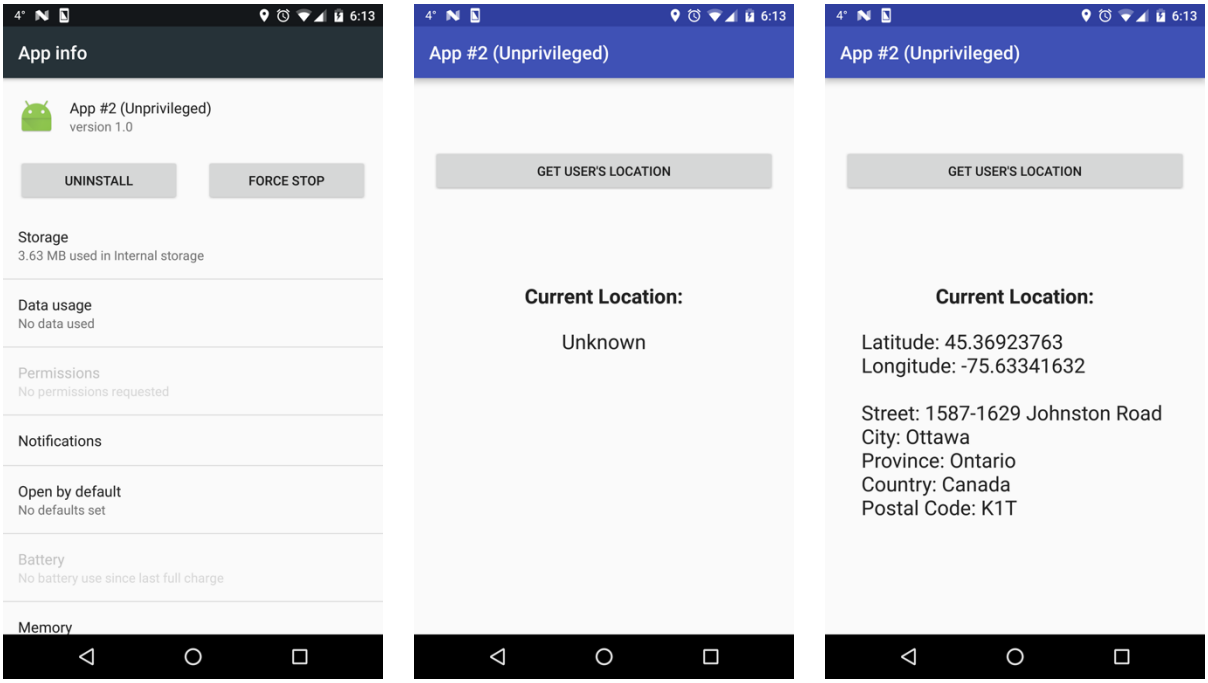

Figure 29 - Screenshots of Application #2

A.3. Android Manifest Explorer

The Android Manifest Explorer that we built is an Android application that extracts and displays to the user information about statically declared components for each of the applications installed on his device. This application does not request or require any permissions. Essentially, all the data is retrieved via the `PackageManager` [91] and `PackageInfo` [92] classes available within the Android operating system. For each application installed on the user's device, the Android Manifest Explorer lists (1) the application's package name (2) the statically declared permissions, (3) the statically declared activities, (4) the statically declared broadcast receivers, (5) the statically declared services, and (6) the statically declared content providers. Our application consists of only one activity that lists all the data in a `ListView` component. Screenshots of the Android Manifest Explorer application are shown in figure 30 below.

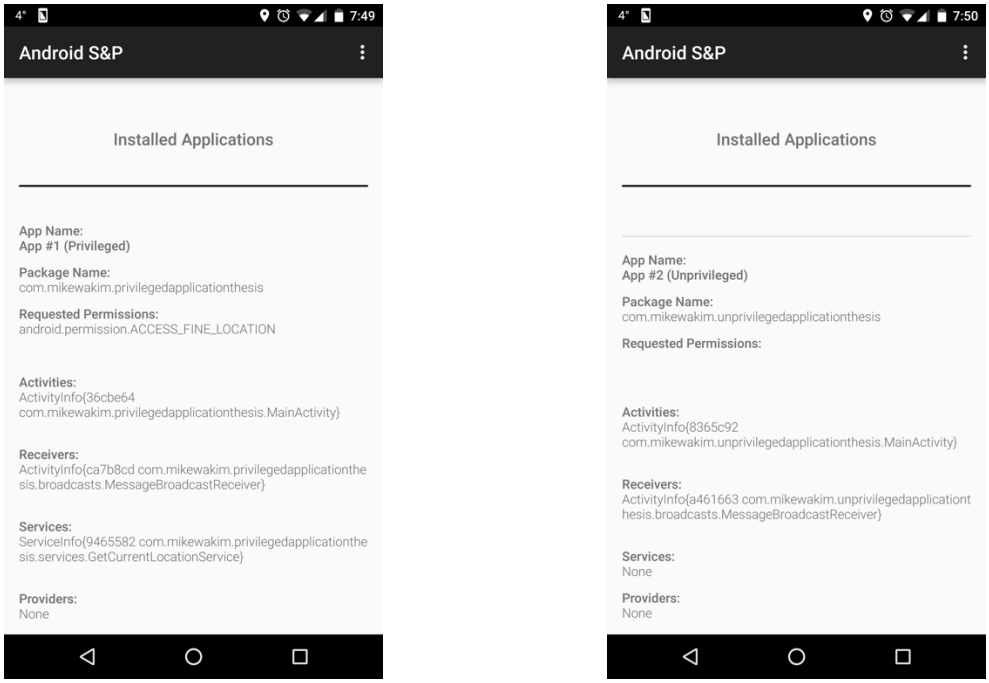

Figure 30 - Screenshots of the Android Manifest Explorer Application

Appendix B

B.1. Introduction

In Chapter 5, we proposed a new design for E2EE messaging applications that implicates two applications: application Alpha and application Beta. Application Alpha is responsible for implementing the security-related functionalities, while application Beta is responsible for implementing the network-related functionalities. In order to demonstrate how our proposed design can be put to use, we decided to build an entire system configuration as a proof of concept. Our system consists of a configuration of application Alpha, application Beta, and a central server. In this appendix, we will provide details concerning the design and implementation of the system that we built.

In Chapter 5, section 5.3.3, we explained that there are two principal forms of integrating application Alpha, application Beta, and the central server: (1) using the one-to-one model and (2) using the one-to-many model. In this proof of concept, the system that we built implements the one-to-one model. In other words, we built only one instance of application Alpha, and only one instance of application Beta. All applications were designed and tested on a stock Android device, with version 7.1.1 (Nougat).

B.2. System Configuration

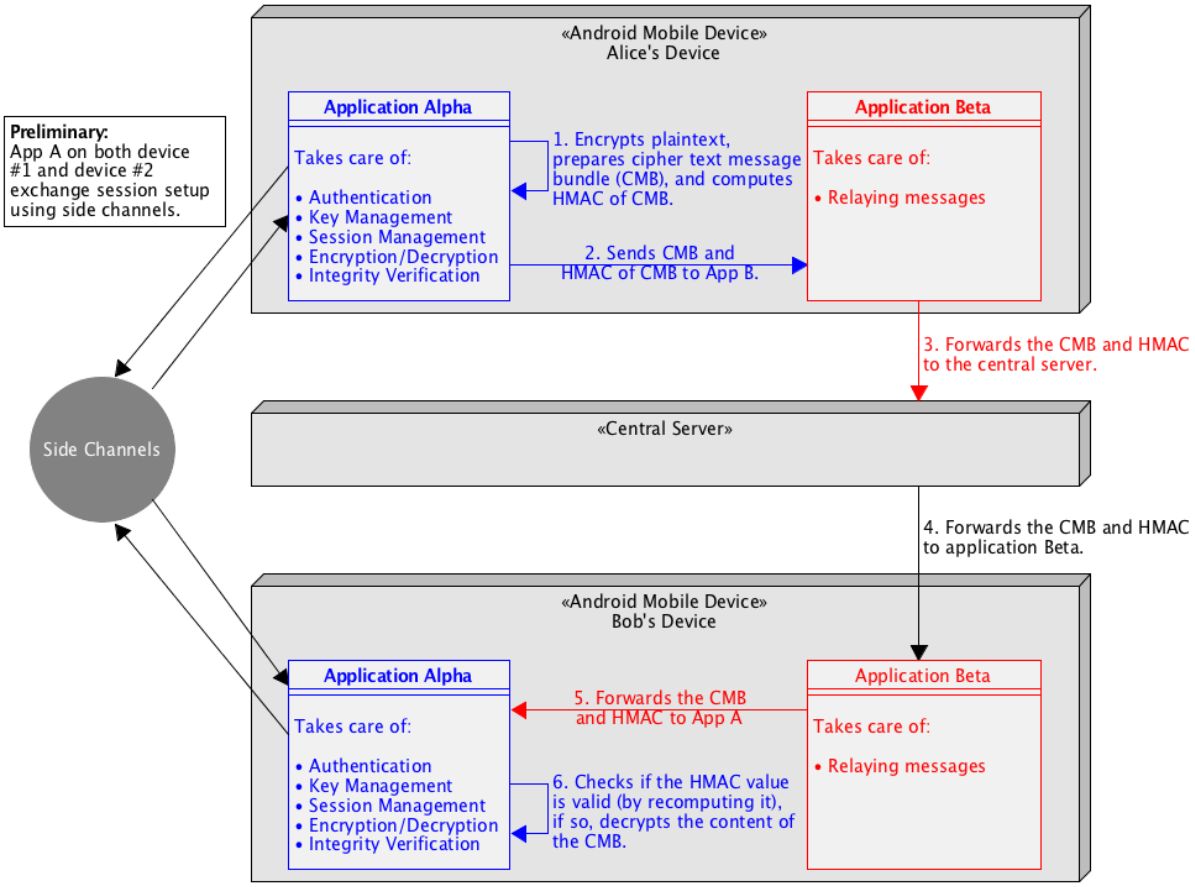

Figure 31 – System Overview: Proposed Design

Figure 31 above illustrates the main concept behind the system that we built. Basically, in order for Alice to send a message to Bob, the following sequence of events must happen:

Preliminary: Alice creates a new session on application Alpha. Alice shares this session setup with Bob through the use of side channels. Thereafter, both Alice and Bob have the same session setup on application Alpha on their devices.

1. Alice inputs in application Alpha the plaintext that she wants to send to Bob. Application Alpha prepares a message bundle that contains details about the sender (in this case Alice), the receiver (in this case Bob), the plaintext, and the date the message is being sent. Application Alpha would then encrypt this message bundle with the key that was shared with Bob and computes and attaches an HMAC value of the encrypted message bundle.
2. Application Alpha sends the cipher text message bundle (CMB) and the HMAC value to application Beta.
3. Upon receiving the CMB and HMAC value from application Alpha, application Beta instantly sends the data to the central server.
4. The central server broadcasts the CMB and HMAC value to all listening applications.
5. Upon receiving the broadcasted CMB and HMAC value from the central server, application Beta on Bob's phone forwards it to application Alpha.
6. Application Alpha verifies the received HMAC value by re-computing it. If the HMAC value is valid, application Alpha proceeds with decrypting the message bundle and accessing its content.

B.2.1. Application Alpha Configuration

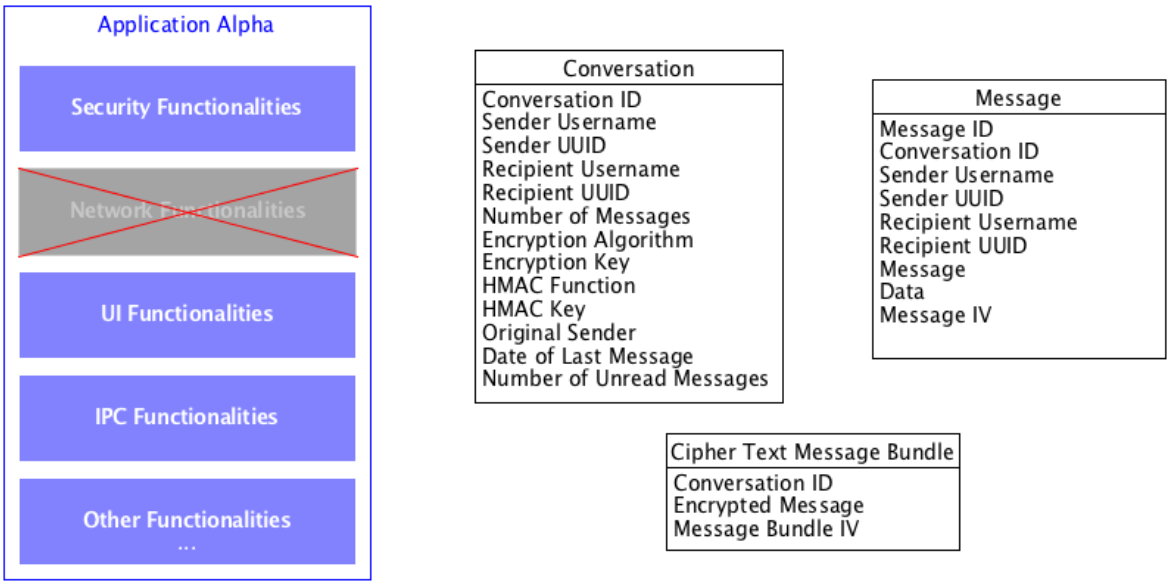

Figure 32 – Functionalities and Models in Application Alpha

Figure 32 above shows the functionalities that application Alpha has to implement along with the models that we utilized. In this section, we will provide some details regarding how we designed / implemented each of the shown functionality groups.

B.2.1.1. Security Functionalities

In terms of security, application Alpha needs to provide functionalities that permit (1) Authentication, (2) Key Management, (3) Session Management, (4) Encryption/Decryption and (5) Integrity Verification. Since the objective of this proof of concept is to simply demonstrate how our proposed design can be used in practice, we did not plan to implement all these functionalities in detail. For each of these functionalities, we will explain what we built, and what we plan to change in future work.

Authentication: In terms of gaining access to application Alpha itself, we are relying on the authentication built in the operating system. Essentially, we are assuming that the user has a locked device (i.e. protected by a password, PIN, pattern, or fingerprint), and that he is the only one capable of accessing the applications installed on it. With regard to existing conversation setups, a user can only participate in a conversation if he has access to all the data constituting a conversation.

Key and Session Management: Before generating cryptographic keys, users must either create a new conversation, or import an existing one. Users may exchange cryptographic keys in any way they want, using whichever side channels they choose. In this proof of concept, we allow users to exchange information about conversation setups (which include required keys) by simply sending these setups by e-mail or by text message. This setup exchange involves plaintext data that is encoded using the Base64 format. Exchanging plaintext data over side channels is not secure at all. Ideally, we should implement an instance of Diffie-Hellman for key exchange, wherein the two parties that want to share a conversation setup agree on a shared key, and encrypt the conversation setup using that key prior to sending it to each other.

Encryption/Decryption: For encryption/decryption, we utilized AES-256. We generated a key using the KeyGenerator class, seeded using a user-inputted passphrase combined with the SecureRandom seed generator provided in the Java security library. For the seed generation, we utilized the "SHA1PRNG" instance. This instance is described in [100] as follows: "*This algorithm uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the*

160-bit SHA-1 output, only 64 bits are used." It is important to note that in this proof of concept, we are storing the keys (for AES and HMAC) directly in a local private database. Although the database is private and protected by the application's sandbox, as a storage option, it is not considered as secure as the Android KeyStore that we mentioned in Chapter 2, section 2.4.4. This is mainly due to the fact that the Android KeyStore was built such that if an attacker somehow gained access to an application's internal storage, or compromised the Android operating system or an application's process, that attacker would still be able to use the keys, but would not be able to extract them from the hardware [30]. The migration of the storage of keys from the local private database to the Android KeyStore should be implemented in future work.

Integrity Verification: We used HMAC-SHA256 for integrity verification purposes. The key used in the HMAC function was also generated using the KeyGenerator class provided in the Java security library.

B.2.1.2. UI Functionalities

Since application Alpha is a standalone Android application, it must implement its own set of activities to supply a user interface for its set of offered functionalities. For end-to-end encrypted messaging, application Alpha must at the minimum support (1) creating new sessions, (2) importing existing session setups, (3) sharing existing session setups with other parties, (4) navigating already existing session setups, and (5) sending and receiving messages within specific session setups. Application Alpha may be augmented to provide user interfaces for other functionalities as well if needed.

In our proof of concept, application Alpha involved the utilization of 6 different activities:

1. **ConversationsActivity**: *ConversationsActivity* is the activity that appears when a user first launches application Alpha. This activity displays all existing conversations, along with a "+" sign to allow users to create/import new conversations. Figure 33 below shows different screenshots of the same activity.
2. **ConversationSetupActivity**: *ConversationSetupActivity* is the activity that takes care of creating a new conversation/session. Figure 34 below shows a screenshot of this activity.
3. **ConversationImportActivity**: *ConversationImportActivity* is responsible for importing existing conversation setups into the application. Figure 35 shows a screenshot of this activity.
4. **ConversationSetupConfirmationActivity**: *ConversationSetupConfirmationActivity* is the activity that appears after a successful creation or import of a conversation. This activity displays the generated AES-256 and HMAC-SHA256 keys. We display these keys only for quality assurance purposes. Figure 36 shows a screenshot of this activity.
5. **CryptoConfirmationActivity**: *CryptoConfirmationActivity* is an activity that is strictly used for quality assurance purposes, we used this activity during development to verify that our implementations of AES-256 and HMAC-SHA256 were working properly. Screenshots of this activity are shown in figure 37 below.
6. **MessagesAcitivity**: MessagesActivity is responsible for displaying the messages in a conversation. Outgoing messages are displayed on the right side of the screen, while incoming messages are displayed in the left side. This activity also provides to the user the

option to share the conversation setup with other users. Screenshots of this activity are shown in figure 38 below.

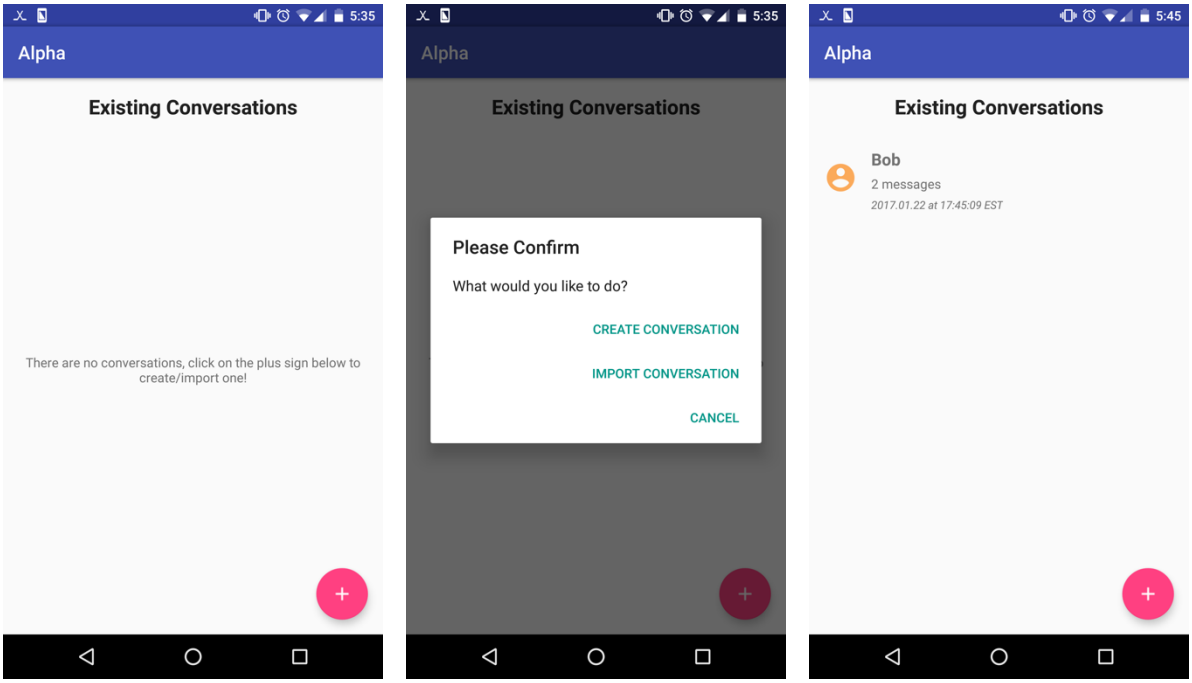

Figure 33 - Screenshots of ConversationsActivity in Application Alpha

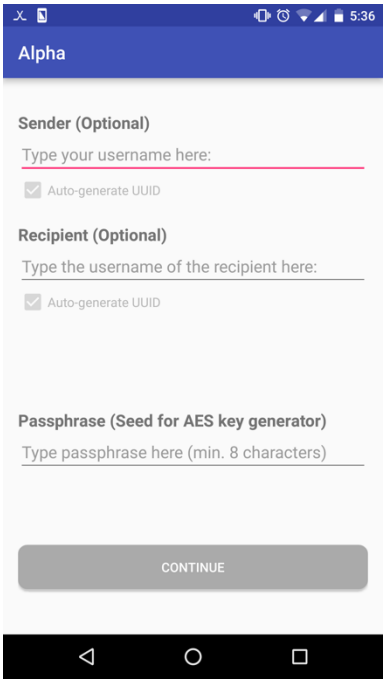

Figure 34 - Screenshot of ConversationSetupActivity in Application Alpha

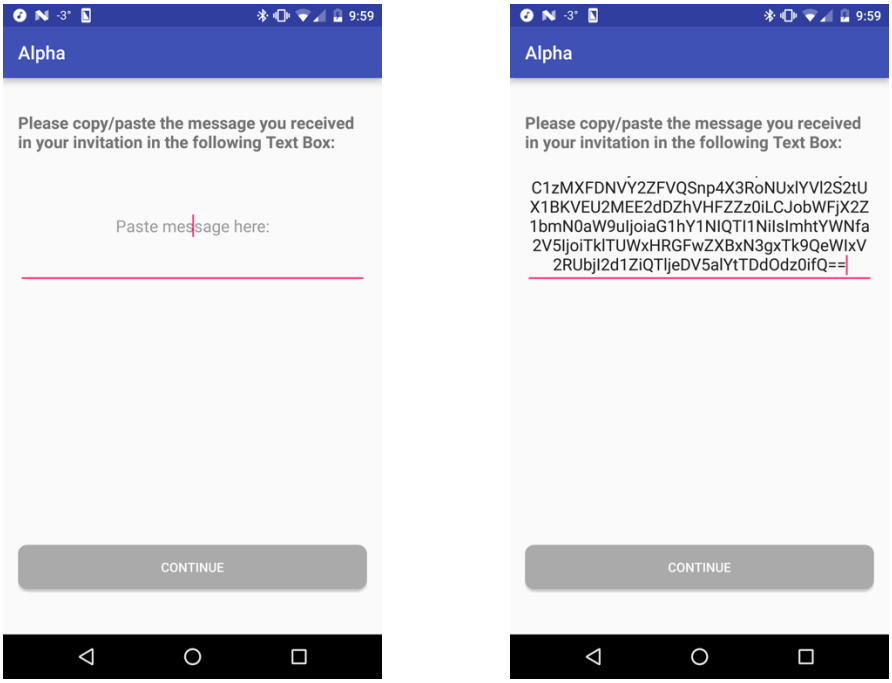

Figure 35 - Screenshots of ConversationImportActivity in Application Alpha

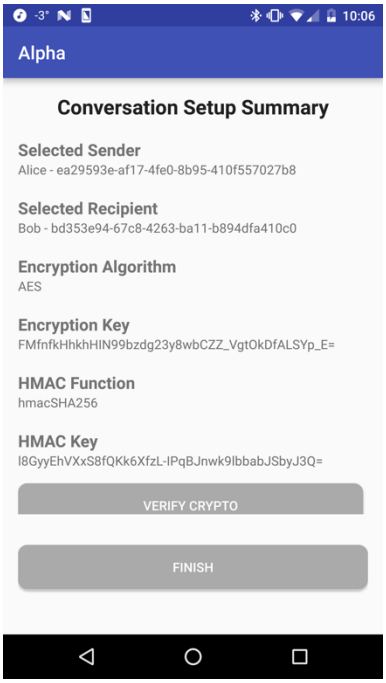

Figure 36 - Screenshot of ConversationConfirmationActivity in Application Alpha

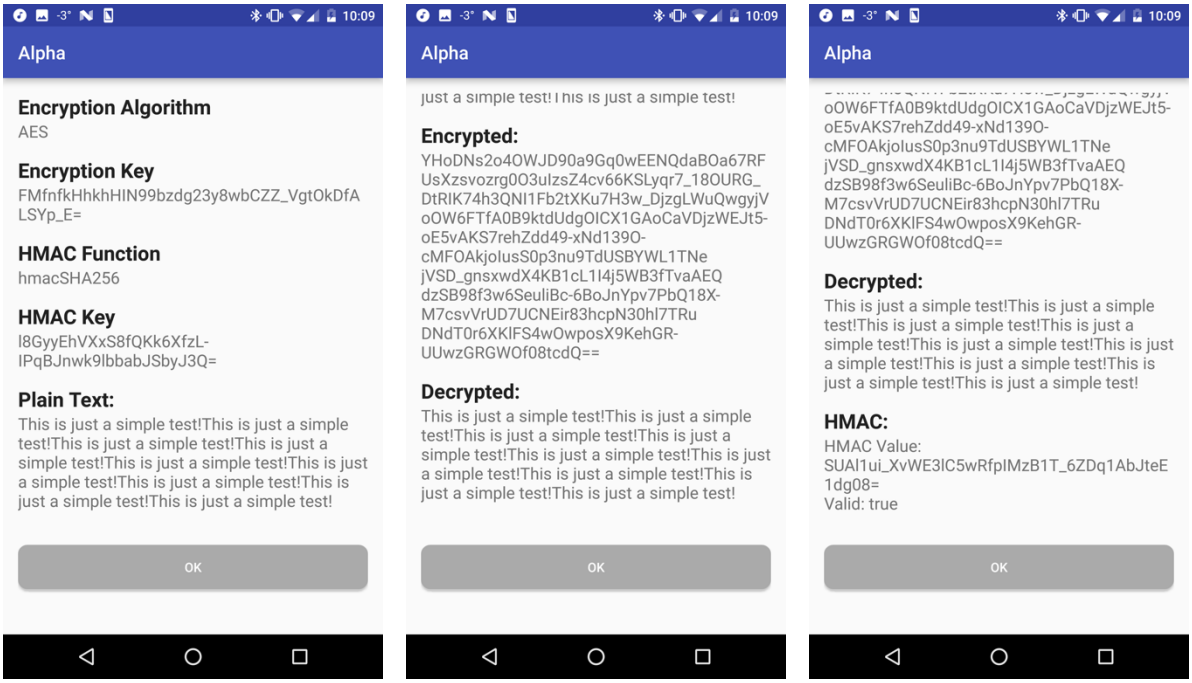

Figure 37 - Screenshots of CryptoSetupActivity in Application Alpha

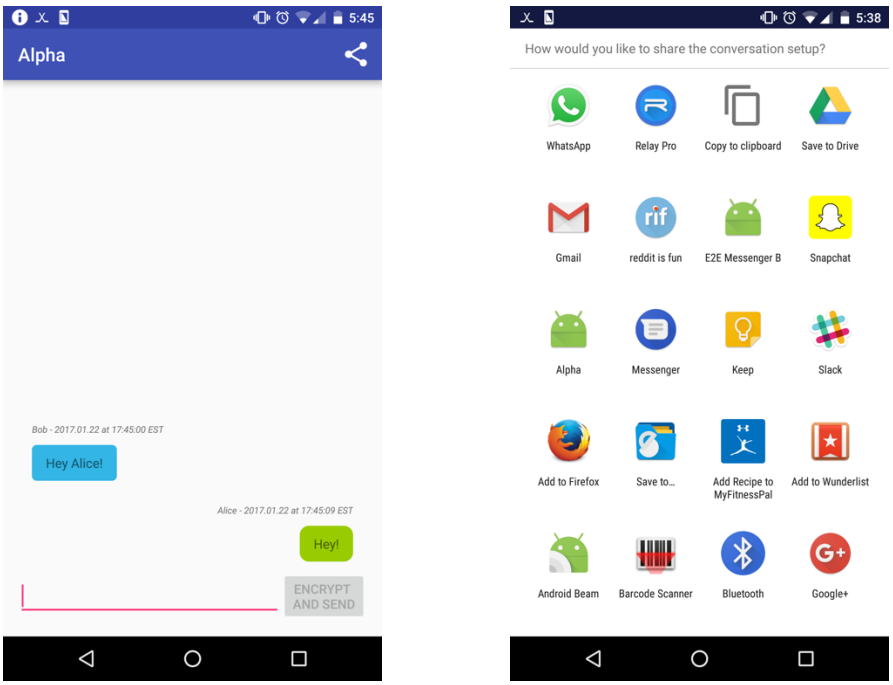

Figure 38 - Screenshots of MessagesActivity in Application Alpha

B.2.1.3. IPC Functionalities

As we have mentioned in Chapter 5, section 5.3.2, application Alpha must implement a cipher text message bundle (CMB) broadcast sender intent and a broadcast receiver. The broadcast sender intent is created in MessagesActivity every time the user wants to send a new message in a conversation. The intent is explicit, it specifies the package of application Beta. The intent targets an action in application Beta that we named "sendEncryptedMessage". On the other hand, the broadcast receiver that we implemented in application Alpha is one that is declared explicitly in the application's AndroidManifest.xml file. Our current broadcast receiver does not require any permissions, and listens to broadcasts that target the action "sendEncryptedMessage". In future work, it would be important to implement custom permissions, and only allow IPC between

application Alpha and an application Beta instance strictly if the user grants that instance the required permissions.

B.2.1.4. Other Functionalities

For application Alpha, we also needed to set up a database in order to store the instances of the models that we discussed earlier in this section. For our database, we utilized SQLite, and we had two tables. We named the first table “ENCRYPTED_MESSAGES”. This table contained all the cipher text message bundles that the user received, and whose HMAC value was successfully verified by the user. We named our second table “CONVERSATIONS”. We stored in this table all the conversation setups that the user created or imported. The AES and HMAC keys are stored in plaintext in our second table. Again, it is important to note that although these tables are protected by application Alpha’s sandbox, in future work, the keys should be stored in the Android KeyStore for better security.

B.2.2. Application Beta Configuration

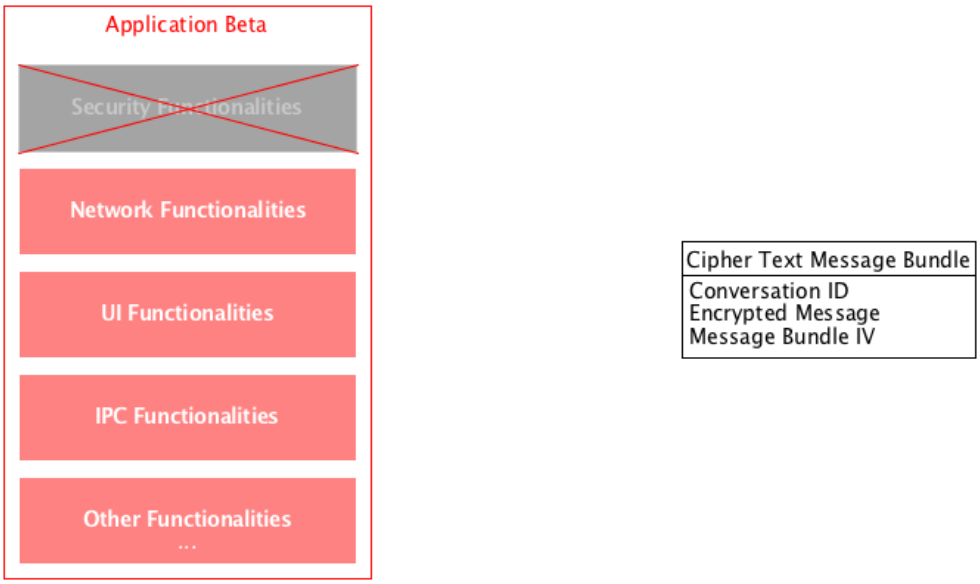

Figure 39 - Functionalities and Models in Application Beta

Figure 39 above shows the functionalities that application Beta has to implement along with the model that we utilized. This application implicated only one model, which was for the cipher text message bundle (CMB). In this section, we will provide some details regarding how we designed / implemented each of the shown functionality groups.

B.2.2.1. Network Functionalities

In terms of network functionalities, application Beta needs to provide the functionalities to (1) send cipher text message bundles from the user's device to the central server and (2) receive incoming cipher text message bundles from the central server. For this proof of concept, we utilized the Firebase Cloud Messaging service available through Google's Firebase mobile platform [98, 99]. We will elaborate more on the setup of our Firebase project in the upcoming section B.2.3.

Essentially, we had to initially set up the firebase project on our Android application, and then we had to implement functions for sending and receiving messages using the cloud messaging service.

For sending cipher text message bundles to the server, we implemented a function that pushes the data to our Firebase database using the Firebase cloud messaging tool. For receiving cipher text message bundles from the server, we set up a service that we named "MyFirebaseDatabaseUpdateService" that continuously listens to push notifications from the server.

B.2.2.2. UI Functionalities

Because application Beta was a standalone application, we also needed to implement a user interface for it. The application consists of one activity which we named "MainActivity". A screenshot of the activity is shown in figure 40 below. The activity simply prints the contents of all incoming cipher text message bundles on the screen.

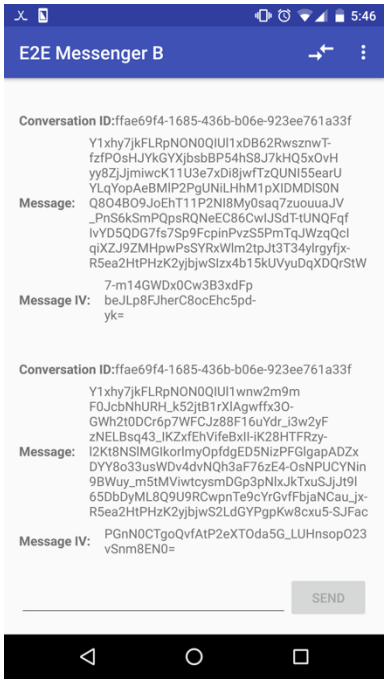

Figure 40 - Screenshot of MainActivity in Application Beta

B.2.2.3. IPC Functionalities

Just like application Alpha, application Beta also needs to implement a cipher text message bundle broadcast sender intent and a broadcast receiver. In this case, the broadcast sender intent is actually implemented in our "MyFirebaseDatabaseUpdateService". Essentially, when application Beta picks up a new cipher text message bundle from the server, it creates an explicit broadcast sender intent, which specifies application Alpha's package, and which targets the action in application Alpha that we named "sendEncryptedMessage". Moreover, in a similar fashion to application Alpha, application Beta also implements a broadcast receiver that is declared statically in the application's AndroidManifest.xml file. Once again, this broadcast receiver does not require any permissions, and listens to broadcasts that target the action "sendEncryptedMessage". As we

already recommended, the IPC between application Alpha and an application Beta should be protected using custom permissions in future work.

B.2.2.4. Other Functionalities

For application Beta, we set up a database in order to store the incoming cipher text message bundles which are sent by the central server. Once again, for our database, we utilized SQLite, however this time we only had one table. We named our table "ENCRYPTED_MESSAGES".

B.2.3. Central Server Configuration

For our central server, we set up a Firebase Cloud Messaging project using the Firebase mobile platform. Upon the creation of the project, we obtained an API key which had to be included in our application Beta. Application Beta cannot communicate with our Firebase server unless it has access to this API key. Furthermore, we set up an online database that is capable of storing our cipher text message bundles. The format of how our data is stored online on the Firebase platform is shown in figure 41 below. Essentially, when a new cipher text message bundle arrives to our Firebase server, the Firebase server stores that message in the database, and then sends a notification that includes the cipher text message bundle to the corresponding instances of application Beta for all users. We implemented this behavior in our proof of concept for its simplicity. In a real-world application, sending notifications to all users at all times is very inefficient and non-scalable. This behavior should also be changed in future work.

APPENDIX B

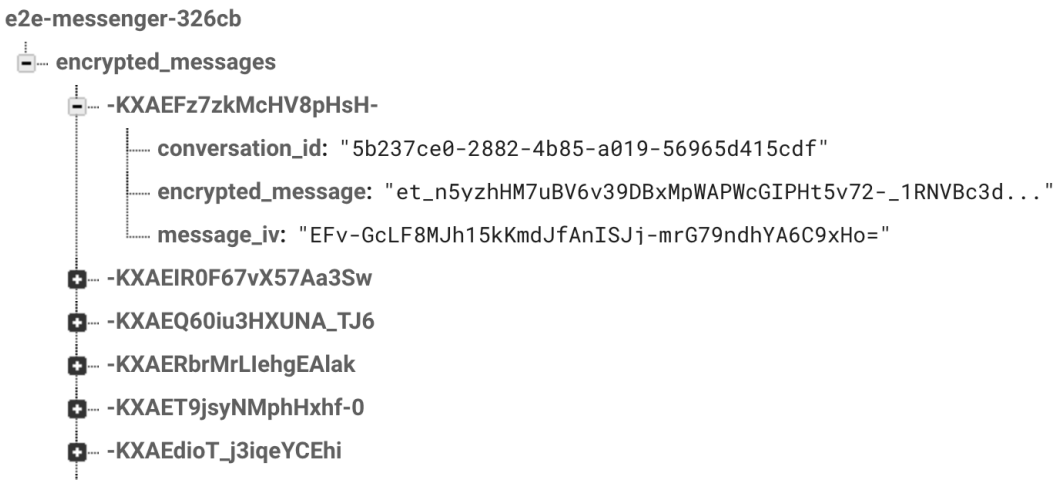

Figure 41 - Format of Data Stored in Online Database