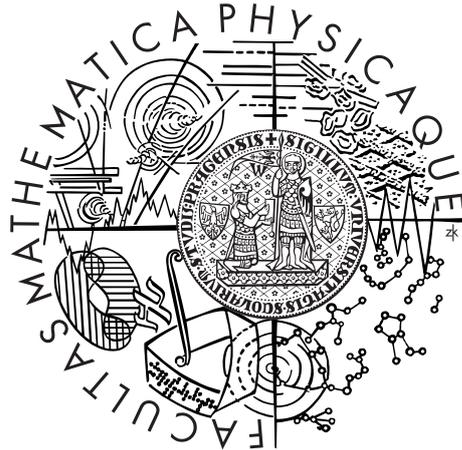


Charles University in Prague  
Faculty of Mathematics and Physics

## MASTER THESIS



Matúš Gažo

## Secure Communicator

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Leo Galamboš, Ph.D.

Study programme: Computer Science

Specialization: Software Systems

Prague 2012

Here I would like to thank my supervisor Leo Galamboš for his guidance and valuable insights into the topic as well as ideas for the thesis. I am also grateful for the steady support I received from my family and my girlfriend Katka. Thank you.

Also a thank you belongs to Google for the well done documentation on Android and for the search engine that brings people facing the same problems together.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date .....

signature of the author

Název práce: Bezpečný komunikátor  
Autor: Matúš Gažo  
Katedra: Katedra distribuovaných a spoľahlivých systémů  
Vedoucí diplomové práce: RNDr. Leo Galamboš, Ph.D.

Abstrakt: Zabezpečená dĺlkov komunikace byla vzdy dležitm tmatem pro lidi manipulujcimi s citlivmi informacemi. Nyn s pchodem tzv. inteligentnch mobilnch telefonů je odposlech a shromadřování informac snadn jako nikdy. Naštst chytr telefony pedstavuj nejenom problmy v oblasti bezpenosti, ale tak prležitost k ochran soukrom. Tato prce se pokou vytvořit generickou softwarovou architekturu komuniktora, kter by ml bt schopen penet hlas, video a dal rzn formy binrnch dat bezpenm způsobem. Budou v n analyzované a pouit rzn komunikan kanly k dosaen maximln rovn autenticity, integrity a dvrnosti dat v prostřed, kde je poteba se vyhnout centrnmu bezpenostnmu prvku. Vsledn architektura bude testovn na Voice-over-IP (VoIP) prototypu pro mobiln platformu Android Google, čím se uke, zda je tento prstup prakticky pouiteln na aktuln dostupnch telefonech.

Klıov slova: bezpenost komunikace, kryptografie, VoIP, SIP, Android

Title: Secure Communicator  
Author: Matúš Gažo  
Department: Department of Distributed and Dependable Systems  
Supervisor: RNDr. Leo Galamboš, Ph.D.

Abstract: Secured long-distance communication has always been an important topic for people handling sensitive information. Now with the arrival of “intelligent” mobile phones eavesdropping and information gathering is as easy as never. Luckily smartphones present not only problems in terms of security but also an opportunity to protect ones privacy. This thesis attempts to construct a generic software architecture of a communicator which could be capable of transferring voice, video and other various forms of binary data in a secure way. It will analyse and use different communication channels to reach a maximum level of data authenticity, integrity and confidentiality in an environment where a central security element needs to be avoided. The resulting architecture will be tested on a Voice-over-IP (VoIP) application prototype for the mobile Google Android platform to show whether the approach is practically usable on currently available phones.

Keywords: secure communication, cryptography, VoIP, SIP, Android

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Goals of the thesis . . . . .	4
1.2	Text organization . . . . .	5
1.2.1	Conventions . . . . .	6
<b>2</b>	<b>Analysis</b>	<b>7</b>
2.1	Main issues . . . . .	7
2.2	Proposed architecture overview . . . . .	8
2.2.1	Events and controllers . . . . .	9
2.2.2	HAL . . . . .	12
2.2.3	Configuration . . . . .	13
2.2.4	Database and utility classes . . . . .	14
2.3	Secure data storage and authentication . . . . .	18
2.4	Secure data transfer . . . . .	23
2.4.1	Available channels . . . . .	23
2.4.2	Public keys exchange . . . . .	25
2.4.3	Signalling plane . . . . .	28
2.4.4	Media plane . . . . .	36
2.4.5	Call history and recording . . . . .	39
2.5	Security analysis . . . . .	41
2.6	Compatibility . . . . .	46
2.7	Android platform limitations . . . . .	47
2.8	Related work . . . . .	47
<b>3</b>	<b>Prototype implementation</b>	<b>49</b>
3.1	Project and package organization . . . . .	49
3.1.1	Used libraries . . . . .	50
3.2	Implementation specifics of components . . . . .	52
3.2.1	The Events and services system . . . . .	52
3.2.2	Hardware abstraction . . . . .	53
3.2.3	Cryptography and CryptoBoxes . . . . .	53
3.2.4	The SIP stack and OpenSIPS . . . . .	54
3.2.5	Calls . . . . .	55
3.2.6	Database . . . . .	57
3.2.7	Graphical user interface (GUI) . . . . .	58
3.3	Implementation specific security issues . . . . .	60
3.4	Building and testing the application . . . . .	60
3.5	Prototype evaluation . . . . .	62
<b>4</b>	<b>Conclusion</b>	<b>65</b>
4.1	Summary and future work . . . . .	65
	<b>Bibliography</b>	<b>67</b>

<b>List of Illustrations</b>	<b>70</b>
Figures . . . . .	70
Tables . . . . .	70
Listings . . . . .	70
<b>List of Abbreviations</b>	<b>72</b>
<b>A Event Types</b>	<b>77</b>
<b>B User Preference Types</b>	<b>78</b>
<b>C User Manual</b>	<b>79</b>
<b>D CD-ROM Contents</b>	<b>83</b>

# 1. Introduction

From the dawn of humankind's history the need to protect gathered information or knowledge existed as it would offer an advantage over a sometimes fierce competition. This "informational advantage" as we would call it today can and was a deciding factor in the survival of individuals, tribes or even whole nations or armies. Take for example a caveman, 10.000 BC, that had the knowledge of where the sparse fresh water and food sources are located, the Roman empires that had technology beyond anything that the competing northern tribes could possibly invent at that time or a company developing a breakthrough piece of software or hardware. All of them had the edge in battles over their rivals for a long time, or to be exact, up the point where their rivals could obtain and use the same information.

As sharing information is in many situations at least as important as storing it the need to protect it would also expand to the exchange channels through which the information was exchanged. Julius Caesar was also aware of that and invented one of the first substitution ciphers that allowed him to command his generals and soldiers on the front lines in secrecy even if his couriers carrying the messages got kidnapped. Of course he was not the ultimately first one in history to pay attention to this matter, various others such as Spartans or Indians have used ciphers before him.

This is where cryptography comes into play. Cryptography – from the Greek word "kryptos" which means hidden, secret - is the study of techniques for secure communication in the presence of malicious third parties. It allows us to transmute data – the "plaintext" – in way to a so called "ciphertext" that only people who possess the corresponding key can decrypt it back and read it. Therefore it enables individuals to keep their informational advantage by minimizing the risk of their information being revealed to eavesdroppers while communicating.

Nowadays, in the digital age so to speak, there is a consensus that sensitive information in form of digital data and its exchange should be cryptographically protected from theft, forgery and abuse. Sensitive data in this context can mean anything from high-grade military technology blueprints, a company's payroll list up to a crooked broker's Swiss account/credit card number or even baby photos if someone considers them private and worth protecting. The stress in previous sentence lies on the word "should" as there are not always the means or even the willpower present to accomplish just that. This applies even more to smart- or mobile phones as the most private and corporate users do have one, but their security lies mostly in the hands of the network operators. Users on the other hand simply rely on the chance of "going under" in the vast mass of other users and therefore presenting a small target to e. g. thieves or corporate spies. They are also hiding behind the logic that information about them is too unimportant or not worth to others to facilitate a greater effort to obtain it. Whether this logic is false or not is more of a philosophical and psychological question and is outside the scope of this text. However fact is that despite a lot of cryptographically highly sophisticated security software, ciphers and protocols designed to keep the data exchange safe do exist, it is not always used and not always is it

easy to use. Also with the quick success and acceptance of smartphones as well as online social networks many people are willing and getting used to share a lot of information about them in a blink of the eye. But privacy concerns do arise from time to time, usually only after it is already too late and someone's credit card has been abused or a trading secret was compromised and private users, companies or even politicians are becoming more and more aware of the continuously easier methods of wire-tapping a GSM call or reading their SMS messages<sup>1</sup>. Smartphones unwillingly contribute in a way to these threats as they present a potentially dangerous combination of having or transmitting the user's private data, their application distribution system ("App stores" and the like) and their mobile internet connectivity which makes them more traceable and vulnerable to attacks. Things may turn even worse as the current trend is to shift payment services to the smartphones as well which would present them as an even more valuable target<sup>2</sup>.

However there are also some positives. Smartphones with their easily programmable platforms do provide an opportunity for 3<sup>rd</sup>-party applications to enhance not only the user experience, but also their security. This opportunity and the trending topic of mobile security provided the main motivation for this thesis. It tries to design and implement a secure but extensible communication software architecture on top of a current smartphone platform. A reasonably easy to use prototype application of this architecture is part of this thesis with focus set on the widespread Google Android platform and on secure voice calls as they are still by far the mostly used communication channel today. Details on the goals of the thesis will be elaborated in the following chapter.

## 1.1 Goals of the thesis

Based on the original specification of this thesis there were three main goals predefined. Each of them has a number of sub-goals which have been summarized for clarity in the following checklist:

1) Preparing an analysis and design for a secure communicator a software architecture. This includes:

- Examining and choosing available I/O channels and mechanisms for data and cryptographic keys exchange. The communicator should be capable to use these channels individually or as a group.
- A detailed description of the architecture itself which has to be extendible, capable of safely storing and transmitting data and potentially portable to other platforms than Google Android (the application's prototype platform). It should not rely on a central authority that would support identity verification or key exchange.

---

<sup>1</sup>See e.g. [1]

<sup>2</sup>The main keyword here is NFC, or Near-field communication technology, that is built into newer smartphones and aims to replace credit cards.

- Pointing out known limitations of the prototype’s underlying hardware and software platform that constrict its implementation.
  - Security analysis of the proposed design with elaboration of possible attack vectors and threats.
- 2) Developing a prototype implementation within given constraints:
- The application is to be developed for the Google Android platform following the proposed architecture.
  - It has to have proper code documentation and a ready to deploy installation package.
  - Secured VoIP<sup>3</sup> calls and methods for exchanging cryptographic keys must be implemented.
- 3) Evaluating the programmed application prototype:
- Verify that the original requirements set in the analytical part were met.
  - In case a requirement was not met, explain why and whether it could be accomplished under different circumstances.
  - Give suggestions for possible future extensions and optimizations of the application or architecture respectively.
  - Comparison of the prototype against related applications.

## 1.2 Text organization

The thesis text is divided into four main chapters, the first being this introductory text. The remaining chapters mimic the outline set in the thesis goals.

Chapter 2 covers the proposed software architecture solution and it’s analysis in terms of the high-level design choices made as well as the possible caveats and security threats for VoIP calls on mobile platforms and how the final application should cope with them.

The third chapter focuses on selected implementation details of the prototype application and some of the low-level choices made at development time.

The fourth and final text chapter evaluates the final implementation against the design requirements specified in the analysis. A conclusion is made about where the limitations of the suggested design lie within the current hardware and software platforms. This is backed up by the summarized knowledge gained in the development process and performance benchmarks. Also possible future extensions are elaborated here along with a comparison to existing software solutions.

Appendixes such as the used literature, listings of illustrations and abbreviations and a short user guide for the application along with other attachments are to be found at the end after the fourth chapter (in this order).

---

<sup>3</sup>Voice over IP: A commonly used acronym for voice calls over the Internet’s IP network.

### 1.2.1 Conventions

Some of the text passages may be highlighted depending on their importance or special meaning, e. g. chapter and subchapter titles are bold and numbered.

Code classes, packages, function calls and variable names referenced directly in the text are formatted with a fixed width font, e.g. `CallActivity` or `getData()`.

File or folder names mentioned in text have a cursive typeface, e.g. *CallActivity.java*. Fragments of programming code or other file contents are pre-formatted and placed in a separate listing such as Listing 1.1.

Smaller figures are placed directly in the text flow where it is convenient, larger pictures or diagrams are placed at the end of the corresponding chapter or subchapter. All tables, code fragments and figures are indexed in the List of Illustrations on page 70.

```
1 // Main method comment
2 public static void main(String[] args)
3 {
4     System.out.println("Hello_world");
5 }
```

**Listing 1.1:** Sample code fragment

## 2. Analysis

This chapter discusses the analytical part of the thesis goals (section 1.1, 1<sup>st</sup> list item) which starts in section 2.1 with identifying the main architectural and implementation problems for designing such secure communicator architecture. The proposed architecture is sketched and analysed in section 2.2 whereas the following sections 2.3 to 2.7 give a deeper insight into each identified major issue with the selected and alternative solutions respectively.

Related work is examined at the end in section 2.8.

### 2.1 Main issues

Building a mobile VoIP application, which in the end the secure communicator is going to be in greater part, is for various reasons a quite difficult task in itself. Requiring certain properties based on the thesis goals such as security and extensibility adds further complexity. In this subchapter the main issues and obstructions that had to be overcome are reflected. Each of them is marked in bold typeface in the following text.

Beginning with the **portability** requirement the final architecture has to take into account it may be run or ported to various platforms with varying hardware and software environment parameters. This means not only different underlying platform APIs but also different means of presentation. For example, mobile platforms come with a relatively small screen size and have a set of own standards for the user interface compared to classical desktop software. Besides this the architecture needs to be **extensible** in the sense that it has to be able to easily incorporate (or even remove) components or features without impacting the existing ones if possible. **Performance**, if possible, should be regarded as a criterion due to limited computational capacities of current mobile hardware. Section 2.2 covers these issues in detail.

**Compatibility** stands not only for that various instances of the same architecture should be able to communicate with each other independent on the endpoints' platforms but also between other, already existing solutions. Section 2.6 elaborates this topic.

Security on the other hand has to be established on all interfaces of the application including safe data storage and network transfer.

**Transferring data securely** involves multiple steps starting with the initial distribution of cryptographic keys and identity authentication continuing with the negotiation of session encryption keys and other security related parameters. In addition typical problems for all VoIP related applications have to be solved like **controlling media sessions**, **NAT<sup>1</sup> traversal** and the **media transmission** itself. See section 2.4.

---

<sup>1</sup>Network Address Translation – A technique used in network routers to modify IP address information in packet headers.

As an application needs also to **store data a secure way** of doing this had to be established in chapter section 2.3.

Safely storing and transmitting data is only one part of the overall security scheme. Without verifying the authenticity and integrity of the received and saved data encryption would be next of useless. A complex overview on the security of the architecture is handled by a thorough **security analysis** in section 2.5.

Last but not least the final prototype implementation must be developed with respect to non-negotiable limitations of the selected platform. These constraints will be examined in section 2.7 in order to expose possible discrepancies between the intended design and the resulting solution.

Having the main issues identified and categorized the following sections analyse each of them in detail.

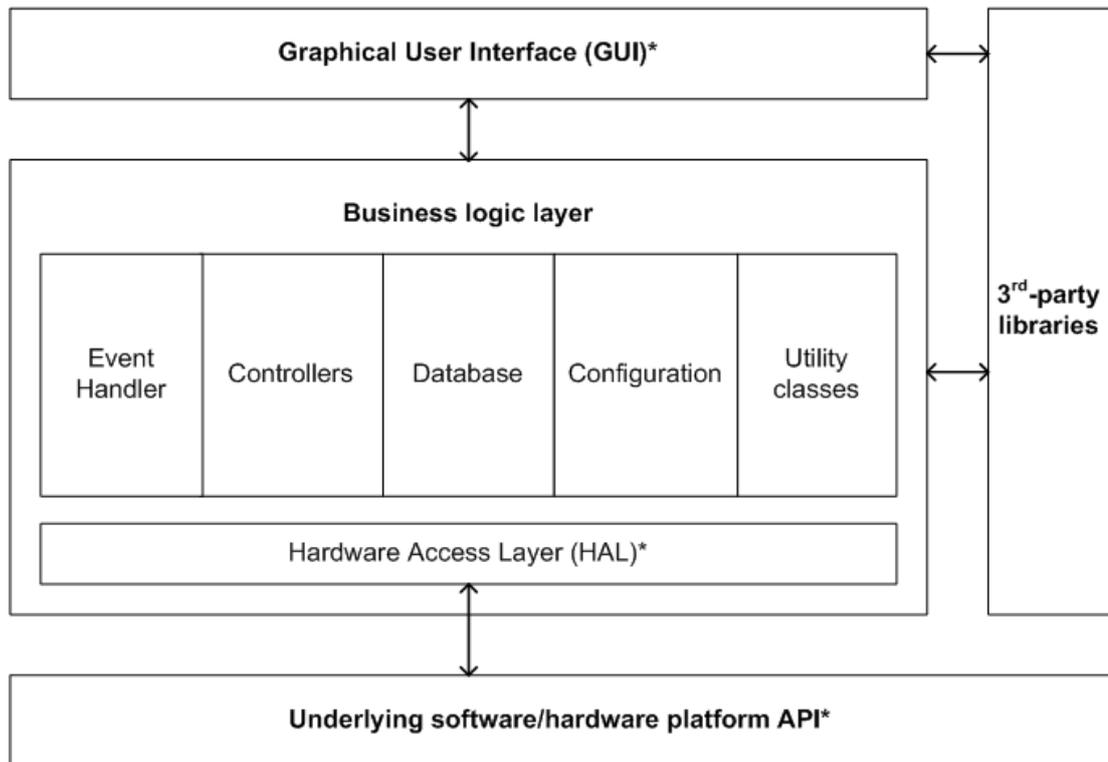
## 2.2 Proposed architecture overview

In this chapter the secure communicator architecture from a high-level perspective is discussed so that it covers the requirements on portability and extensibility. Security and lower-level problems are discussed in subsequent sections.

As already pointed out the issues for designing a portable architecture for this type of application are twofold, once related to displaying data on various display configurations and on the other hand related to accessing the underlying platform's API. This gave reason to use a 3-tier architecture with loose ties between the graphical user interface (GUI) and the core logic unit that is situated on top of a platform abstraction layer. The core logic is based on a universal events system which is easily extendable with additional components. The question of why this is advantageous and what cons this set-up may have will be answered in the process of explaining it in detail.

The proposed architecture thus consists of four top-level components ordered in three tiers (depicted in Figure 2.1):

1. A **presentational layer** responsible for drawing the GUI and handling interactions with the user and the application logic.
2. A **business logic layer** containing the core logic of the application. It has five sub-components. A platform abstraction layer serves as the connector between the components and the underlying platform.
3. **3<sup>rd</sup>-party libraries** shared by both of the previous layers as they may address cross-cutting concerns and it would make no sense to include them twice.
4. The **platform** itself on which the application will eventually be deployed.



\* Components dependent on the selected platform

Figure 2.1: High level architecture overview

## 2.2.1 Events and controllers

The architecture specifically lays no constraints on the final form nor the inner workings of the GUI as it may differ from platform to platform. However the connection between the GUI and the logic layer is of great importance as it is their single common point. Hence the connection needs to be well specified. This is achieved through an event based system. Events in this context mean invoking a specified action without calling a particular class method or function directly. The class that handles the event eventually may send an event in response to signal a result of the invoked action or to trigger another action in order. All event types and their corresponding actions and responses are defined precisely in Table A.1 (p.77) in the attachments section.

The `EventHandler` component aside from receiving the mentioned events is responsible for keeping a registry of listeners to them. A listener is basically every class that:

- A. Implements the `ICryptDroidEvents` interface which contains a single method `handleIncomingEvent(Event e)`.
- B. Registers its bindings – a set of event types they want to listen to – in the handler by calling one of the subscription methods.

Listeners are typed as static or runtime. Static means they are known at compile time and so are their bindings which must be registered at application start-

up automatically by calling `subscribeStaticListener(EventType e, ICryptDroidEvents callback)`. Runtime listeners may be for example dynamically created GUI windows or dialogs with a relatively short life span where it is undesirable that they should respond to events if they are hidden or not existent. Therefore they register themselves in the event handler at creation time and should deregister themselves at destruction time (e.g. closing a dialog). For this the methods `subscribeRuntimeListener(...)` and `unsubscribeRuntimeListener(...)` should be used respectively (parameters omitted as they are equal to the static subscription method).

A further categorization of static listeners is that they may be specified to be singletons or prototypes. Singletons must follow the singleton pattern ([2]) and possess a static method `getInstance()`, prototypes on the other hand a valid parameterless constructor. The difference is that singleton listeners are a single instance of the class that must be returned by the `getInstance()` method. The method is called by the `EventHandler` every time an event is to be processed by that listener. Similarly every time an event bound to a prototype listener is fired a new instance of that listener is to be created.

Runtime listeners do not share this distinction as an instance always already exists – the one that registered itself.

Listeners in the logic layer than contain the core logic were labelled as controllers.

Firing an event means to call one of the `EventHandler`'s `sendEvent()` methods with the desired event type or the event object directly. Event objects should be created only in the `EventFactory` class by calling `EventFactory.createEvent(EventType)` and they are allowed to carry a generic object as the payload (parameter). The receiver of the event has to downcast the parameter by himself. UML diagrams with complete interfaces for the `EventHandler` and `EventFactory` are shown in Figure 2.2.

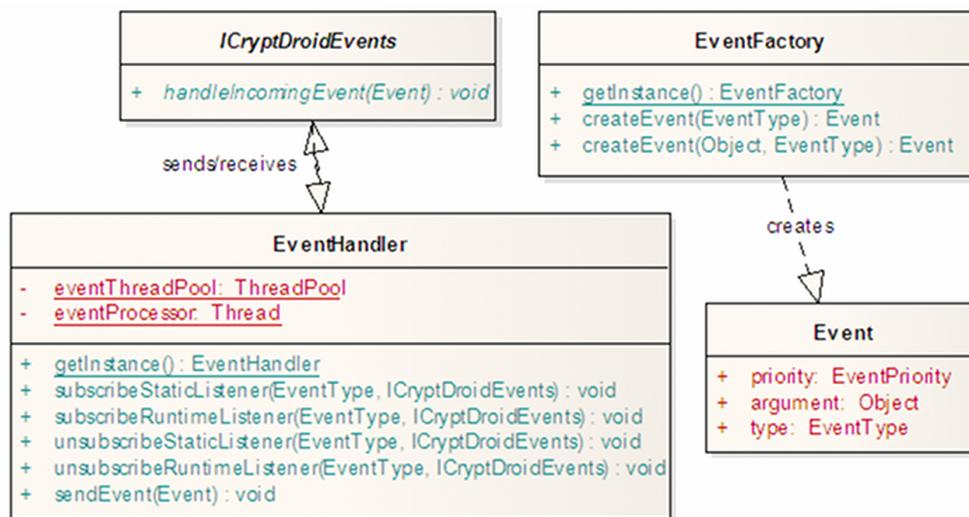
After the event is received by the event handler it is processed and queued. The event handler should process the events asynchronously meaning that a continuously running thread should take events from a queue and sent them to all registered listeners by calling their `handleIncomingEvent(Event e)` method. In respect to their type the listeners may have to be created first. It is advisable to use a thread pool to invoke the callback method in as a separate task as it a) limits the number of concurrent running callbacks and b) is better for exception handling should the callback method fail in some way. Listing 2.1 demonstrates this process in pseudo-code. This also means the event originator implementation should make no assumptions about the time, duration or the order of the processing of the fired event.

```

1 for each registeredListener to given event
2 begin
3   if registeredListener.type = "runtime"
4     then
5       listener = registeredListener.get()
6     else if listener.scope = "prototype"
7       listener = new registeredListener.class
8     else
9       listener = registeredListener.getInstance()
10    end if
11    ThreadPool.submitTask(listener, event)
12 end

```

**Listing 2.1:** Event processing pseudo-code



**Figure 2.2:** UML Diagram of EventHandler and EventFactory

As long as all the specified event actions and responses to them (see Table A.1 on page 77) are stable and predictable the implementation may add custom controllers/listeners or even event types without impacting the core logic. By all means this is definitely an advantage over tightly coupled implementations as it allows easy extensions and modifications of the final implementation. However by adding an extension it must not happen that an event is not responded to when it should be or it is responded twice to (except when this would be the expected behaviour for some reason).

Such a setup has also other advantages, one being that using events does not tie the GUI implementation to a specific class implementation or interface. In a sense events therefore replace the classical programming interface. Also firing a single event may trigger multiple actions, but the GUI does not need to keep track of this which in turn reduces the logic complexity in the presentational layer. Another benefit is that all events are handled by as single `EventHandler` component which can perform operations on the events such as to queue and prioritize them before their further distribution to event callbacks.

The biggest disadvantage is synchronization between two related events. For example imagine the user starting a call and after a while cancelling it while the phone is still ringing on his side while in the meantime the remote party has already accepted the call. Correctly the cancellation should be ignored. Senders and receivers of events therefore have to be able to take care of such situations even if the event handler makes no guarantees on the order of processing events. A partial solution to this is that event types can be prioritized. However prioritization does not guarantee that the higher priority event processing gets finished first and can send a proper response before the second event arrives in another controller. In case of similar situations the listeners may have to use a synchronization lock in the events callback method to prevent the object state being in disarray.

Synchronization issues do also arise in complicated object state flows, where multiple events must or can be chained one after the other. This requires multiple event responses and for the controllers to wait for them before firing another event. This may lead to an undesirable high amount of event types (one for each status change). A solution to that is to use the same event type with different arguments (such as a sequence number, a tag or a flag).

One of the alternatives to the event system would be to move the (connecting) parts of the application's core logic to the GUI layer and keep the controllers only as simple black-box approach objects with a single strictly defined functionality. Apart from the obvious shift of responsibility towards the presentational layer this approach would require a set of a set of callback interfaces in order to respond to naturally occurring events in the logic of the application (such as an incoming call). The GUI then would have to either register these callbacks in the corresponding controllers and services or at a central repository that keeps track of all callbacks. Since tracking down all needed callbacks and responsible objects would be error-prone and tedious a central repository would seem as the better choice here. But this is basically how the event handler works now, with the added benefit that the GUI does not need to care from which controller the event originates or where to subscribe to listen for a specific event as all subscriptions are made in the event handler. In addition the events system removes the burden of maintaining complex logic from the GUI and keeps the presentational layer simple. Ergo it was decided to use the events system as it offers more flexibility, features and structure for less effort.

## 2.2.2 HAL

Exceptions to the single point of contact rule between the GUI and the business logic layer are classes with public static methods that may be called from everywhere. This applies to utility classes with helper functions but mainly to the database<sup>2</sup> and the platform abstraction layer, or "HAL". The technically correct abbreviation would be "PAL", but the more common and known term HAL is used nevertheless.

---

<sup>2</sup>See subsection 2.2.4

HAL is responsible for abstracting the underlying platform layer away from the core logic and GUI. It thus defines interfaces to system services, hardware and any other functionality that would require a direct platform API call or is somehow tied to the underlying OS. The GUI can probably access the API straightforwardly because it has direct access to the platform, but this should be avoided whenever possible as it makes code less reusable and portable. Directing all possible calls through HAL is also preferable as it allows it to manage or pool resources more effectively.

Platform specific implementations of HAL in the current version have to support at least abstractions for system notifications, audio playback and recording, accessing the file system, system information (such as the device's serial number) and preferences. Additional abstractions are allowed as deemed necessary to implement the core logic of the application.

An alternative solution to the HAL pattern would be similar to the alternative approach of the events system where all parts of the application's core logic requiring a system API call would be moved to the presentational layer. Because of the previously mentioned portability and efficiency arguments and in this case also the added complexity of the GUI implementation the loose-coupling HAL standard pattern was favoured.

### 2.2.3 Configuration

Each application should be configurable to some extent without rewriting portions of code. Our proposed architecture needs to calculate with that and makes therefore a – admittedly classic – distinction between static (compile-time) configuration and user's (runtime) preferences. Static configuration is to be understood as globally used non-changing variable properties such as the application name or the hashing algorithm for the database password. It is up to the implementation to manage its compile time preferences.

Preferences are dynamic settings of various parts of the application that the user has to be able to change through the GUI at runtime. The architectural specifications for user preferences are that they have to be stored as key-value pairs of strings in the local application database (subsection 2.2.4). Strings have been selected as they are descriptive and most commonly used<sup>3</sup> versatile information holders across all platforms. Naturally not all preferences are of string type, thus preferences have to have beside a default string value also a type. The type of the preference defines the value's class in the given programming language so that the value can be converted (typed) to e.g. a platform native boolean or number type from the string it is stored in. Default values may be empty but the preference should be then set in the initial set-up phase of the application.

An implementation has to support all preferences listed in Table B.1 (p.78). It may add custom preferences if needed to. Other implementations can ignore the

---

<sup>3</sup>For example Windows INI files or Android's SharedPreferences class use strings as keys.

unknown preference keys in situations like importing a database that was created by a different implementation. To avoid such complications an implementation may decide to support also local, platform-specific preferences (e.g. INI files) to outsource the storage of additional preferences outside the database. However no sensitive data like credentials or security settings should be stored there unprotected.

## 2.2.4 Database and utility classes

As stated previously, the GUI layer may access the application's logic either through events or utility classes. Utility classes should be in most cases classes containing only static helper methods and variables within a generic context. They are often unavoidable as part of the effort to not repeat code of oneself and as such shouldn't be hidden from the upper layers or hindered by events.

The database component in the logic layer is responsible for persisting data and providing it to the application. Data in this context means anything the application needs to save between individual runs, for example contact information or user preferences. To carry out this task effectively the conceptual proposition is that the database component should be a hybrid between a utility class and a controller. Merely using events to retrieve or store datasets would yield no advantage, only performance loss and added complexity. Theoretically it would be required to use always at least two events – one being the request specifying the dataset to retrieve or save and another one to return the dataset or confirm its (un-)successful write to the database in case of inserting. Also the responsibility for data organization lies exclusively on a single component which in this case makes events partly obsolete. Therefore it is more practicable to view the database as a singleton utility class in which case other components may easily call its static methods to perform CRUD (Create-Read-Update-Delete) operations. Obviously all operations have to be synchronized using either programmatic locks or some other kind of transaction mechanism to prevent data inconsistency.

The database component should be a hybrid solution in the sense that it has traits of a controller as it should register itself as a static singleton listener to receive events. More specific it has to register for the `SYSTEM_EXIT` event type to recognize when the application is about to exit and data should be written to disk. Another event type is `SYSTEM_PERSIST_DATA` which has to be invoked either periodically or in certain situations that may require data persistence. It is up to the implementation to decide in which situations this applies (e.g. on minimizing the GUI or adding a contact). Triggering the event should have the effect of saving the current database state without closing the database connection and destroying the listener object.

There are more event types to be handled by the component as the database component serves also for authentication purposes. They will be explained in section 2.3 along with the data security design.

Of course, the attractiveness of the events concept is that the implementation is free to handle the approach of being a data provider in another way as long as all required data is kept consistent and all required events are serviced properly.

Figure 2.3 shows an entity-relationship (E-R) diagram of all data objects that need to be persisted by the application (their meaning will be specified throughout the text). It is not relevant to the final implementation that the figure is an E-R diagram as it does not mean that it has to use a relational database, the E-R form is just for illustration. In fact the approach prescribed by the architecture is quite different and uses the Google Protocol Buffers (GPB) data interchange format. Developed by Google Inc. protocol buffers are an extensible language and platform neutral technology for serializing structured data. Protocol buffers use so called prototype objects (or “messages“) to hold data in named fields. Fields have a name, a data type and can be optional, required or repeated. They are defined in textual form in .proto files which look much alike a CORBA IDL<sup>4</sup> file. Proto files are converted to data access classes in the specified programming language by the *protoc* compiler. These classes can be then used programmatically and also converted to a byte stream and read back from one. In Google protocol buffers language terms this is referred to as “building“ and “parsing“ the message. For a full reference on protocol buffers please consult [3].

The protocol buffers data format was chosen as the best fit for the required data structure, speed and portability among other competing approaches. Table 2.1 compares data formats and technologies that were considered in the analysis for storing data. Main criteria for selection were:

1. The ability to process the format on different platforms, namely Microsoft Windows & Windows Phone, GNU Linux, Apple iOS and Google Android.
2. Does the technology support CRUD operations in forms of queries?
3. The speed of serialization (write) and de-serialization (read).
4. The comparative amount of required space to store the data. That is given mainly by two factors - the size of the metadata (such as indexes or markup tags) and the data itself (e.g. compressed binary vs. text form).
5. The technology’s ability to read files that were stored within an older data scheme (version). This is important due to being able to import or read older files with a newer version of the application without specifically customizing the code for each version.
6. Other unique disadvantages or benefits of the technology.

Please note that the table’s column values reflecting points 3. and 4. regarding the speed and required space of each solution are only relative estimates of the author due to a lack of comparison material of such a wide range of technologies.

Having this table made the decision process fairly straightforward in favour of protocol buffers. The reasoning behind this decision is that a custom solution to serialize data for storage would mean a lot of work to invest in something that

---

<sup>4</sup>CORBA, or Common Object Request Broker Architecture, is a middleware architecture; IDL stands for Interface Definition Language.

<b>Solution</b>	<b>Multi-platform</b>	<b>Query-able</b>	<b>Speed (est.)</b>	<b>Space re-quired (est.)</b>	<b>Back-wards compatible</b>	<b>Other</b>
<b>Custom</b>	Yes	Yes (?)	High	Low	Yes	Error prone
<b>SQLite</b>	Yes	Yes (SQL)	High	Medium	No	Storage only
<b>XML</b>	Yes	Yes (XPath)	Low	High	Yes	
<b>Protocol Buffers</b>	Yes	No	High	Low	Yes	RPC-able
<b>Java serialization</b>	No	No	Medium	Medium	Partially	Java only

**Table 2.1:** Comparison of data formats

already exists and is tested by time and other users. The Java serialization technique is as the name already says for Java only which limits the multi-platform abilities significantly. Also while being backwards compatible the compatibility is quite limited as for example it is not able to read objects in a newer version that lack an attribute present in an older version ([4] section 5.6.1 Incompatible Changes). While XML could work around this problem (e.g. by making all elements optional) it still has drawbacks in comparison to protocol buffers. Quoting from [5] “Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

- are simpler
- are 3 to 10 times smaller
- are 20 to 100 times faster
- are less ambiguous
- generate data access classes that are easier to use programmatically“

These advantages clearly favour GPB over XML. The last considered solution was SQLite, a light and embedded version the SQL relational database management system (RDMS). It fulfils all criteria and in addition to GPB is queryable as being non-queryable is the most striking weak point of protocol buffers. Despite this protocol buffers were chosen as the best solution available as encrypting data in SQLite databases would have either to use a difficult cell-by-cell encryption or encrypt the whole database file at once. While encrypting whole databases would be a valid solution it offers no other benefits such as integrity checks of

the encrypted database or backwards compatible versioning. Also only proprietary commercial solutions for encrypting SQLite databases are available (such as SQLCipher [6]). The tipping point was that SQLite databases can be used only for storing data locally as they are not a data exchange format. Protocol buffers on the other hand are one and will be used in the architecture not only for data storage but repeatedly also as the basic building block for most of the data transfers.

As the security of the solution is tightly tied to the used technology it will be explained in section 2.3 together with how exactly the architecture defines its data structure using the protocol buffers format.

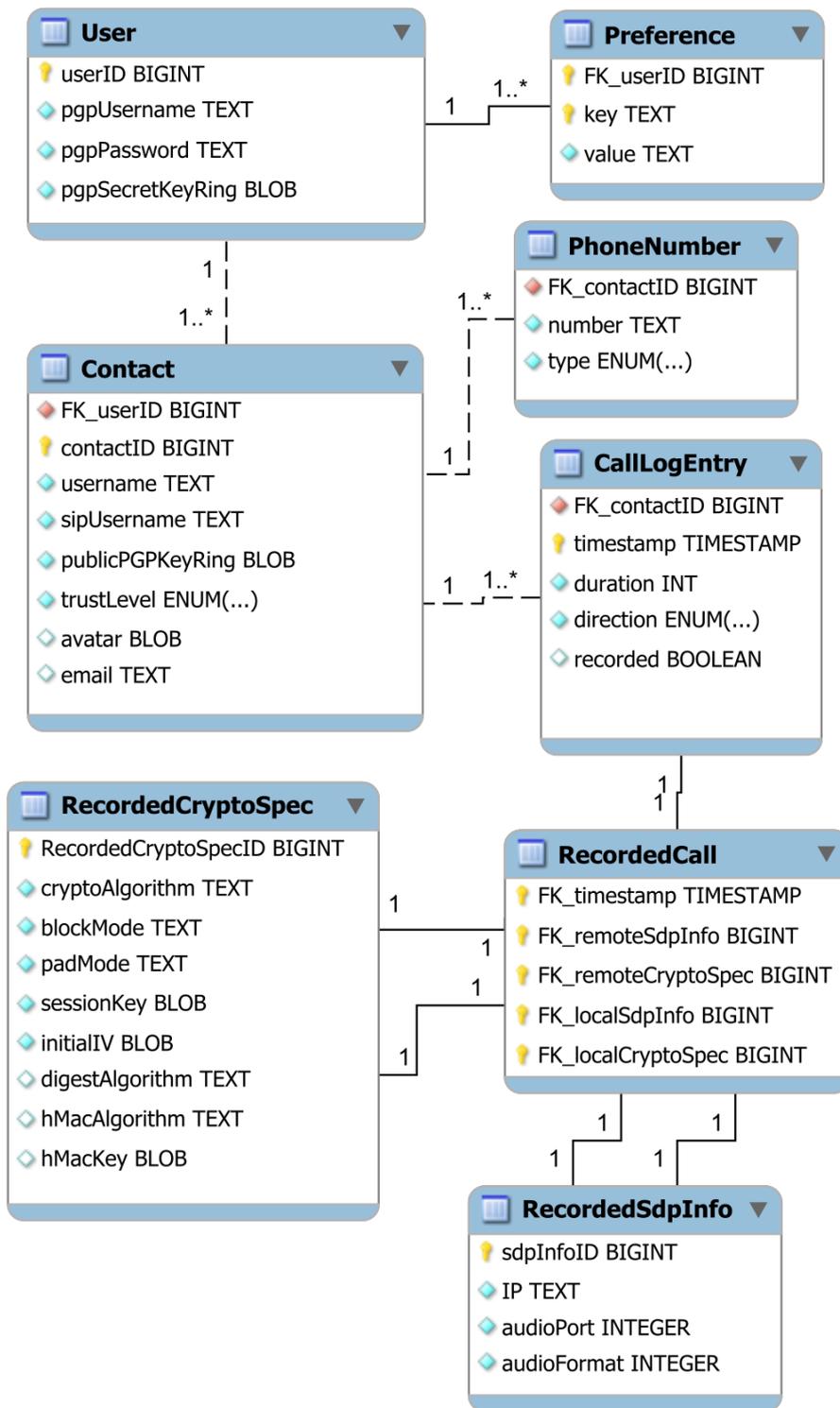


Figure 2.3: Database E-R diagram

## 2.3 Secure data storage and authentication

The previous chapter analysed reasons behind using Google protocol buffers as the choice for data storage. This chapter focuses on the specifications for the database in this format as well as what security measures have to be taken to ensure data safety and integrity. In addition it will look into how user authenti-

cation is related to the database component.

In order to achieve security in terms of data storage the proposed architecture has to solve

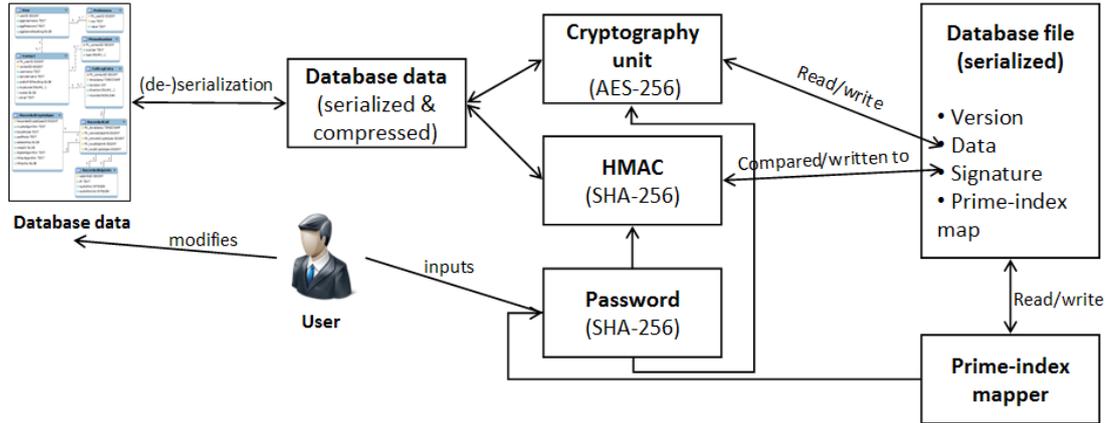
- a) Data confidentiality by encryption to ensure persisted data cannot be meaningfully read and interpreted by 3<sup>rd</sup> parties.
- b) Data integrity to ensure persisted data was not tampered by a 3<sup>rd</sup> party between application runs.
- c) Authentication which is responsible to ensure that only authorized entities may open and decrypt the database file successfully.

Figure 2.4 portrays the proposed security scheme to accomplish all three goals. Basically the scheme uses a password based authentication and involves multiple steps between saving the database data and writing it to a physical storage device (and back). The first step is to take the in-memory data entities and serialize them into a single byte stream. This is achieved by building a `DatabaseData` protocol buffer message (defined in Listing 2.2) and filling the appropriate fields - which in substance is only the root `User` entity as defined in Figure 2.4. The `User` message itself has fields corresponding to other database entities. In turn each database entity has a corresponding protocol message format defined so it can be included in the `User` message. Sources of remaining *.proto* files are too extensive to be listed in the thesis text and are therefore stored on the attached CD-ROM (Appendix D, p.83).

The serialized byte stream obtained from the built `DatabaseData` message object is additionally compressed with the GZIP algorithm before it is sent to the cryptographic and data integrity unit. Compression serves two purposes – for one it conserves disk space and for one it partially hides patterns in the “plaintext“ (the byte stream) that could be exploited for a cryptanalytic attack ([7]). The compressed byte stream represents the input for the cryptographic and data integrity unit. The cryptographic unit is responsible for encrypting and decrypting the byte stream with a provided 256 bit password using the 256 bit Advanced Encryption Standard (AES) symmetric algorithm. The same password is used in the data integrity unit that creates a signature by employing a hash-based message authentication code (HMAC). The signature will serve as a proof of data integrity and authenticity when loading the database file back into memory.

After the encryption is done and the signature is calculated the byte streams are stored in the appropriate fields of the `Database` message along with other properties such the database scheme version and a string describing the HMAC algorithm used for the signature. The `Database` *.proto* message is shown also in Listing 2.2. Finally the `Database` message is built and its byte stream form is saved onto a storage device (e.g. hard-drive).

Reading the database means reverting the whole process where instead of converting protocol buffer messages into byte streams the byte streams are parsed



**Figure 2.4:** Database component security scheme

into the messages. Also the HMAC signature/hash is built again from the plain-text (`DatabaseData` message byte stream) and the password and compared to the one stored in the `Database` message. If the hashes do not match the saved byte stream of the `Database` object has been either damaged or tampered with and the loading process should stop with an error message.

```

1  message DatabaseData {
2      optional int64 lastUpdateTimestamp = 1;
3      repeated User users = 2;
4  }

6  message Database {
7      required int32 version = 1;
8      required bytes data = 2;
9      required string dataSignatureAlgorithm = 3;
10     required bytes dataSignature = 4;
11     optional string primeIndexMap = 5;
12 }

```

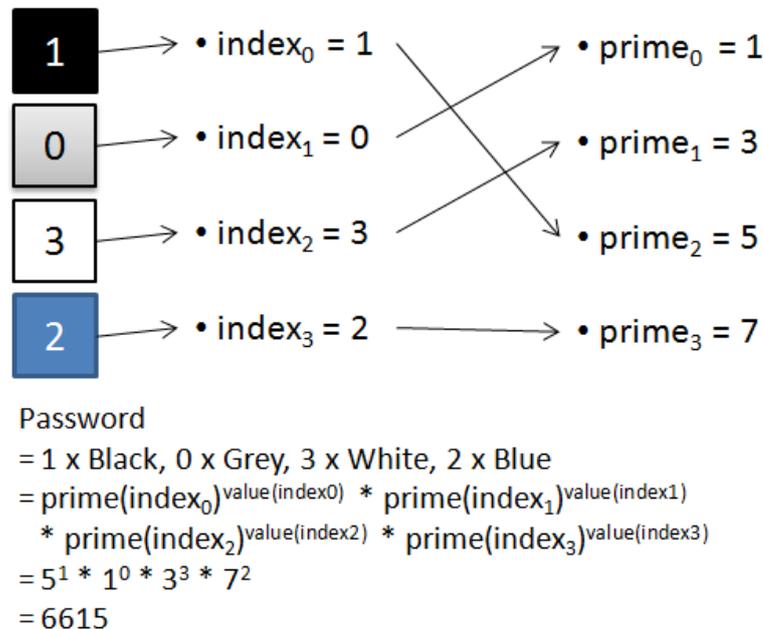
**Listing 2.2:** Database wrapper in Google protocol buffers code

The password used in both ways in the cryptographic and data integrity unit is always the same due to the symmetric nature of the algorithms used in both units. It is 256 bits strong and a result of the SHA-256 hashing function that combines three inputs. The first input is the username the user enters at application set-up and serves as the cryptographic salt together with the third input which is the current device's identification (ID) number. It is up on the HAL implementation to return a unique ID number of the device, for instance on desktop PCs this may be the serial number or on mobile phones this could be the International Mobile Equipment Identity (IMEI) number. It may be empty if no such number can be determined.

Since the first and last inputs are static the middle (second) part of the password has to be dynamic. The second part is therefore the actual password the user has to choose at set-up time and then always enter on a login screen before the database is to be decrypted and loaded. Thereby his identity (or at least

knowledge of the password) is verified. The password is not a simple personal identification number (PIN) or a string pass phrase but a mapping between pre-selected prime number indexes and their power. More exactly the user has to enter the power of each “index“. An index is a sequence number which is at set-up time mapped randomly to a prime number. The middle password part is then calculated as the product of all primes to the power of the number entered by the user (the password). It is the responsibility of the Prime-index mapper component to write the initial random mapping to the database object and supply it to the password calculation method on every following database loading process. The mapping is to be stored in the `primeIndexMap` field of the `Database` message as a serialized string which must have the form of the index for a given prime in order separated with the “|“ character. For example the serialized string “1|2|0|3“ would mean that prime with the ID zero (0) is assigned to index one (1), prime with ID 1 is assigned to index 2, ID 2 to 0 and so on... Figure 2.5 illustrates this example mapping together with an example calculation for the middle part of the password using colors to identify indexes.

At first glance this may seem overly complicated however it is in short no more than an algorithm that utilizes the Chinese remainder theorem ([8]) to build a big prime product number (password) that is hard to factorize and therefore hard to break<sup>5</sup>. In addition salts are added to prevent rainbow attacks as the final password is a SHA-256 hash due to the requirement of the cryptographic and data integrity units which need a 256 bit password to function properly. Listing 2.3 shows the whole process in pseudo-code.



**Figure 2.5:** Example password calculation and prime-index map (“1|2|0|3“ in serialized form)

<sup>5</sup>Factorization is believed to require super-polynomial time in the number of digits. See also section 2.5

```

1 byte[] calculateFinalPassword(
2     byte[] username,
3     Map<Integer, Integer> password,
4     Map<Integer, Integer> primeIndexMap)
5 {
6     Integer pwd = 1;
7     for (Integer index : password.keySet())
8     {
9         Integer indexCount = password.get(index);
10        if (indexCount != null && indexCount > 0)
11        {
12            Integer prime = PrimeIndexMapper.getPrimeById(
13                primeIndexMap.get(index));
14            prime = prime ^ indexCount;
15            pwd = pwd * prime;
16        }
17    }
18    byte[] deviceId = HAL.getSystemInfo().getDeviceId();
19
20    byte[] finalDbPassword =
21        Bytes.concat(username, pwd.toByteArray(), deviceId);
22    finalDbPassword = SHA256(finalDbPassword);
23    return finalDbPassword;
24 }

```

**Listing 2.3:** Final password calculating function (Java code)

Figure 2.5 uses small primes for illustration purposes, however the primes used in the final implementation have to be big enough to make a factorization near to impossible (at the current state of hardware and algorithms). It may use for example the Mersenne primes 12 to 19 (which corresponds to a range from  $2^{127}-1$  up to  $2^{4253}-1$ , see [9]) or any other primes as long as they are big enough and equal on all platform implementations to ensure portability of the database file. The primes should be part either of the static configurations of the application or included in the `Database` message (which would mean to add an optional field). This applies also to adding more primes (and therefore indexes) than eight which is the minimum. The password itself should consist of at least 3 primes with a count of all index values greater or equal to six to ensure sufficient security.

Using the described approach has an added benefit besides the harder factorization of the password than using a simple PIN. As the password consists only of a mapping between the index and the number of times it has been selected it may be presented to the user in various forms. The form used in the example used colors but other forms are thinkable such as mapping the indexes to a PIN pad or a pattern-screen lock.

The username and the prime-index mapping are set up at the initial run of the application. They may change in time if the application is running and the new prime-index mapping is stored so the database can be encrypted with the new encryption key or password respectively. The initial set-up is to be recognized by the database component by listening to additional events to the ones mentioned in subsection 2.2.4. Namely the `SECURITY_USER_FIRST_SETUP`,

SECURITY\_USER\_AUTH and SECURITY\_USER\_LOGOUT event types have to be registered by the component. It is recommended that the database component uses holder objects as parameters for the event types due to the limitation of events being able to carry only a single argument.

SECURITY\_USER\_FIRST\_SETUP is to be sent after the user fills all required fields to initialize the database the first time. It has to carry at least the username, the password as an integer-to-integer map (for each index and its power) and credentials for the SIP account (see 2.4.3). SECURITY\_USER\_AUTH events have to carry the username and password as in the previous event and are to be sent on each application start-up to log in the user. Log-in in this context means that if the final (hashed) password is correct the decrypting and loading of the database file succeeds and the component has to respond with the SECURITY\_USER\_AUTH\_OK event so the application may start up its regular operations. Otherwise the component has to respond with a SECURITY\_USER\_AUTH\_FAILED event which should signal the user that either the entered password was wrong or any other cause that may have caused this error such as a malformed database file.

## 2.4 Secure data transfer

Just as secure data storage is important to the architecture design so is data transfer. This chapter discusses the proposed design to provide just that starting with an analysis of available channels and methods for data transportation in subsection 2.4.1. As data security is not thinkable without data encryption subsection 2.4.2 handles ways of exchanging keys. With an emphasis on secure VoIP calls sections 2.4.3 and 2.4.4 focus on the data transmission and how it is controlled. However the approach shown will not be exclusive to voice calls and generic enough to incorporate video, text messaging and other data transfers if desirable.

### 2.4.1 Available channels

In this chapter an analysis of available data exchange channels is discussed. A channel in this context is any way two devices with the secure communicator application installed may exchange binary data. As the main feature of the secure communicator is to carry encrypted voice data first the needed bandwidth required for a channel needed to be calculated. For this purpose any full-duplex channel capable of continuous streaming with a bandwidth above 128 kbit/s (kb/s, or 16 kB/s) will be considered compliant. The number is derived from a simple calculation where the highest used bitrate by usually used audio codecs is 64 kbit/s demanded by the wide-band G.722 codec. All other codecs use a smaller or equal bitrate as depicted for example in [10]. To be on the safe side this number has been doubled to cover data transmission overhead introduced by wrapping the voice samples into packets and by the taken security measures as described in subsection 2.4.4. Of course transmitting other types of data such as video calls or larger files may require a faster connection but for now only the critical voice call feature is to be considered.

Since the data transfer is encrypted slower or even one way channels that could be used for exchanging encryption keys of two parties are considered. For this any channel capable of transmitting a 2 kB block of data is to be considered applicable (2 kB is the defined upper bound for a serialized Contact entity which holds the public keys and other contact information as described in the following chapter). In addition streaming capabilities of the channel are not a prerequisite because the key is exchanged only once in a single transaction e.g. taking a picture of a QR code with a camera.

Table 2.2 summarizes channels available on most current smartphones and desktop computers. The table is split into a listing of channels capable of a simultaneous two way communication and one way only channels. The latter case is hence a (non-exhaustive) list of actuator/sensor combinations. The data transfer speeds and range values presented in the table are taken from the official specifications and may vary in some cases greatly depending on the exact hardware device and protocol or technology version used. Also factors like signal quality influence the overall speed but the numbers give at least an approximation of the order in which the channel operates.

The protocols column lists standardized protocols or data formats which can be used on the particular channel. The “Network“ value means the channel supports network protocols such as TCP/IP or UDP. The last column indicates by an OK sign (“√“) or an “x“ sign whether the given channel meets or meets not the given bandwidth criteria for voice and encryption key/contact information transfers.

It is obvious that the “traditional“ connection types using an ethernet/DSL, wireless (Wi-Fi) or 3G mobile network meet the set requirements. In case of Wi-Fi however it depends greatly on what connection is the Wi-Fi router hooked up to. For example if the router is bound to a slow 64 kb/s ISDN modem the channel is effectively unusable, although in that case it is thinkable that if both parties are in the given range only the local network is used or an ad-hoc network is created. 2G mobile networks may theoretically be usable for voice data depending on the exact network type (EDGE vs. GPRS) but it is questionable whether the theoretical data rate will be achieved in practice. Also network latency may become an issue in this case.

While the remaining three channels (USB, Bluetooth and NFC) fulfill the bandwidth limitation for voice calls they are strongly limited by their range and cannot be recommended for voice data since it would be easier just to speak to the remote party in person. However they may be used for safely transmitting other types of binary data such as documents and public keys. USB is in this point of view a special case as it could be used only as a proxy channel for a shared internet connection (USB-tethering).

The one way channels suffer from the same range restriction as the previous three and are even slower. Thereby they should be used for offline key exchange only.

Regarding the final implementation for Android smartphones it should use at least one of the listed one way channels for a convenient way to exchange keys

Type	Approximate data transfer speed (down/up)	Protocols	Range	Usable for voice/key exchange
<b>Two way channels</b>				
<b>Ethernet</b>	24/1.3 Mb/s (ADSL2+)	Network	N/A (cable)	✓/✓
<b>Wireless</b>	11/11 Mb/s (802.11b)	Network	~30m	✓/✓
<b>3G mobile network</b>	7.2/2.88 Mb/s (HSDPA)	Network	Operator coverage	✓/✓
<b>2G mobile network</b>	236.8/236.8 kb/s EDGE Class 10	Network	Operator coverage	✓/✓
<b>Bluetooth</b>	2.1 Mb/s (v2.0, EDR)	L2CAP (others on top)	< 100m	(✓)/✓
<b>USB</b>	480 Mb/s shared (USB 2.0)	USB protocol	N/A (cable)	(✓)/✓
<b>NFC</b>	424 kb/s	LLCP (others on top)	< 0.2m	(✓)/✓
<b>One way channels using actuator/sensor combinations</b>				
<b>Vibrator/ Accelerometer</b>	N/A	Custom	0m	x/✓
<b>Display/ Camera</b>	N/A	Custom, QR codes, ...	< 0.5m	x/✓
<b>Speaker/ Microphone</b>	367/0 kB/s (est. at 44.1 kHz)	Custom, DTMF	< 1m	x/✓

**Table 2.2:** Comparison of data transfer channels

when people meet in person. A realizable method seems to be scanning QR codes from a second device's screen (provided both devices possess a camera). Other combinations or methods may be possible and are encouraged. For voice data transmissions the fast enough wireless and 3G mobile networks should be used preferably as smartphones usually do not provide Ethernet cable jacks.

## 2.4.2 Public keys exchange

A goal of the thesis was to secure the data exchange between two endpoints without a central authority that would provide key exchange or identity verifica-

tion which by principle would be the weak point in the overall security scheme. In response to that requirement the decision fell immediately on public key cryptography that can provide the needed decentralized authentication and public key infrastructure. The following text assumes knowledge of both topics. An overview on public key cryptography can be obtained from [11] or [12].

There exist many standards and approaches to asymmetric key cryptography and exchange and one that would fit both the security requirements and the architecture of the secure communicator had to be selected. As such the seasoned and freely available OpenPGP standard according to RFC 4880 was chosen due to its proven reliability and stable implementations. The alternative choice, the X.509 standard (RFC 2510) was disqualified due to the need of central certificate authority. The Diffie-Hellman (D-H) key exchange method on the other hand, whether standalone or included in the ZRTP (RFC 6189) standard, is not really structured for authentication purposes and would serve only for key agreement of the data transmission. Since identity verification must be done at an earlier stage (signalling the data transfer) both of these solutions were inadequate. In comparison to that the Pretty Good Privacy (PGP) suite and its OpenPGP standard provide a complete cryptographic system.

This concludes that every user that wishes to be part of the secure communicator network must possess a public PGP key ring together with its secret (private) counterpart. Due to the principle of public key cryptography the public part needs to be distributed to each party that the user wishes to communicate with before the communication takes place. Since the architecture cannot rely on a central Public Key Infrastructure (PKI) authority (server) to do that, the exchange has to be facilitated in channels suitable for public key exchange as discussed in the previous chapter. The proposition is therefore to use a common data format for the key exchange that is versatile enough for all channels in consideration. Such a format is defined as follows:

Contact information along with the public PGP key ring is stored in the `Contact` protocol buffer message as displayed in Listing 2.4. This message is identical to the database entity for contacts which allows for easy import and export of contact information directly into or from the database – including our contact information which is of the same type (stored in the `userinfo` field of the `User` message). Public PGP key rings are to be stored as byte encoded arrays in the `publicPGPKeyRing` field. The `optional` (including `repeated`) fields of the message may be left blank but all fields should be editable in the GUI after the contact message has been transferred from one user to another. The transportation channel is free to be chosen by the implementation however it must use a built `Contact` protocol buffer message wrapped into a Base 64 encoded string. Base 64 encoding ensures that the built message in form of binary data may be transported over channels that use textual unicode or even ASCII encoding modes of operation.

The public PGP key ring must contain at least two keys. The first key must

be a master signing key while the second key must be an encryption sub-key<sup>6</sup>. The keys however must not be exclusively only for signing or encryption, they may employ general algorithms suitable for both operations. Both keys should be at least 1024 bit strong and use the RSA encryption algorithm by default. The digital signature algorithm (DSA) should also be supported (because of imported key rings). Additional keys with arbitrary properties are allowed in the key ring and are to be ignored. When importing a contact message into the database, the presence, the expiration time and correctness of both required keys has to be checked.

The `TrustLevel` attribute is an enumeration with the following allowed values: `NONE`, `LOW`, `MEDIUM` and `HIGH`. It indicates subjective trust the user applies to a given contact. When exporting the contacts message belonging to the current user the value should be overwritten to `NONE` so the recipient is made aware to customize the trust setting. It has no effect on the encryption or application logic and serves for informational purposes only. In the future this field may be used to establish a PGP web-of-trust model.

```

1 message Contact {
2   required int64      id = 1;
3   required string    username = 2;
4   required string    sipUsername = 3;
5   required bytes     publicPGPKeyRing = 4;
6   optional TrustLevel trustLevel = 5;
7   optional bytes     avatar = 6;
8   optional string    email = 7;
9   repeated PhoneNumber phoneNumbers = 8;
10 }

```

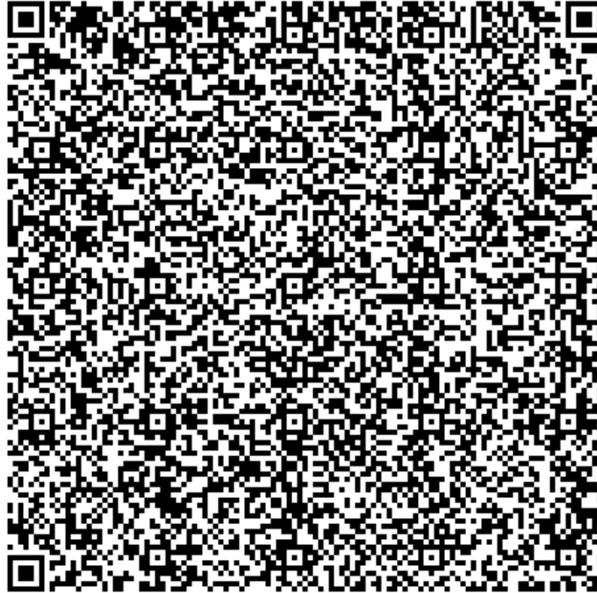
**Listing 2.4:** Protocol buffer message for a contact (shortened)

Figure 2.6 contains a sample contact QR tag with the Base 64 encoded `Contact` message. Sharing contact information through QR codes implies clearing the `avatar` field of the `Contact` message due to the limit of data a QR code can carry.

Using QR codes in conjunction with the display as the key exchange channel eliminates the risk of a Man-in-the-middle (MITM) attack because the users have to be in physical contact range to scan the code with their device. Using other transport channels the exchange itself may or may not be encrypted with a temporary session key (such as a PIN similar to when two Bluetooth devices are connecting) to prevent Man-in-the-middle (MITM) attacks. This however is not a “hard“ requirement as the users can always check the incoming PGP key ring’s fingerprint.

Users however must be given the choice to exchange the whole contact or just import the public PGP key ring part into an existing or a newly created contact.

<sup>6</sup>Note that using two keys is also the recommended approach in other OpenPGP implementations such as GnuPG or the PGP suite.



**Figure 2.6:** Sample QR code with contact/PGP data

This is needed in order to replace revoked or expired PGP keys and various day-to-day reasons (mainly backwards compatibility). Exchanging plain key rings must be though dealt with by the user himself and is outside the scope of this architectural proposition.

Each public PGP key ring naturally needs a secret counterpart to create signatures and decrypt incoming data. The secret PGP key ring must be either imported together with the public one or created anew. This has to happen together with the initial database creation so the fields in the database can be correctly used at first start-up already. Users should be able to import any existing PGP key ring as long as it fulfils the specifications on the master signing and encryption sub-key and of course has the according credentials for the given key ring. The secret key ring is to be stored in database in the `pgpSecretKeyRing` field of the `User` database entity along with the PGP username (`pgpUsername` field) and password (`pgpPassword` field).

The next chapters explain how and when both of the key rings are put to use.

### 2.4.3 Signalling plane

Before a secure data transmission can start there are three things to be considered in the architecture. First the application needs to locate the remote party the user wishes to communicate with and signal it a request to establish a data connection. Secondary the identity of the remote party needs to be verified (on both ends) and only then the transmission may start if accepted. At last the data transmission has to be encrypted and prior to that session keys must be exchanged. Session keys are in contrast to asymmetric public keys used only once per data transfer session and hence their name. Due to the expected amount of data it is a) not possible to use the comparatively slow asymmetric cryptographic algorithms directly and b) it is safer to use a random temporary, one-shot key

(provided using a secure random number generator). Therefore session keys are destined for symmetric cryptographic algorithms such as AES.

Because of this distinction the data transmission can be split in two planes. The signalling plane is responsible for locating, contacting and verifying the remote user and uses asymmetric cryptography (OpenPGP more precisely). Meanwhile the media plane is the lowest layer where the data itself is exchanged in encrypted form using symmetric encryption algorithms.

It is not a coincidence that the terms signalling and media plane are used as they are common to the Session Initiation Protocol (SIP). SIP is the protocol of choice which the secure communicator uses in the signalling layer. Alternatives to SIP will be discussed at the end of this chapter. Describing the detailed functioning of the Session initiation protocol is outside the scope of this thesis text so please refer to exhaustive guides to SIP in general, its message flows and response codes in [13] and [14]. Full specifications are located in the RFC 3261 [15].

A SIP proxy/registrar server is the backbone of the SIP signalling plane as it keeps a registry of all connected user agent clients (UAC) and routes requests between them. UACs in this case are active users who have started the secure communicator application and are able to participate in a communication. It is important to note that while by using a SIP server a central element is introduced into the architecture it has only a supporting function. In regards to the overall security scheme the server itself has no substantial effect on authentication or data encryption mechanisms except that it relays the messages required to establish a data transfer session.

Since the secure communicator must cope with SIP related situations the architecture contains a number of events associated with SIP. A controller – the SIP stack – must be responsible for interpreting and translating internal events into SIP requests and responses and of course the other way around. Basically there are four of these situations the SIP stack has to consider:

1. Initializing the SIP stack
2. Registering the UAC at the SIP server. This includes registration refreshes before the registration entry expires in the registry.
3. Incoming call or other data transfer.
4. Outgoing call or other data transfer.

Because SIP is a state based protocol the SIP stack object has to keep track of its state to properly respond to requests. Figure 2.7 shows a state diagram of a SIP stack according to the four situations and how the stack switches between them based on received or sent SIP messages. The state transitions are conditioned by an incoming SIP event denoted in square brackets (“[]”). The event may have three origins – a user action, an incoming SIP request or an incoming SIP response (numeric code) to a previous request. An initialized SIP stack begins in the `STACK_INITIALIZED` state from where it automatically tries to

register itself at the SIP registrar server. Should the registration succeed the stack idles in the IDLE state until a registration refresh fails, an incoming call is signaled (INVITE type message) or an outgoing call is requested by the user. The WAIT\_PROV and WAIT\_FINAL states are used on outgoing calls whereas the RINGING and WAIT\_ACK states are used on incoming calls. The ESTABLISHED event is common for both occasions and indicates that the session was successfully established and the media packets containing voice data are being sent and received.

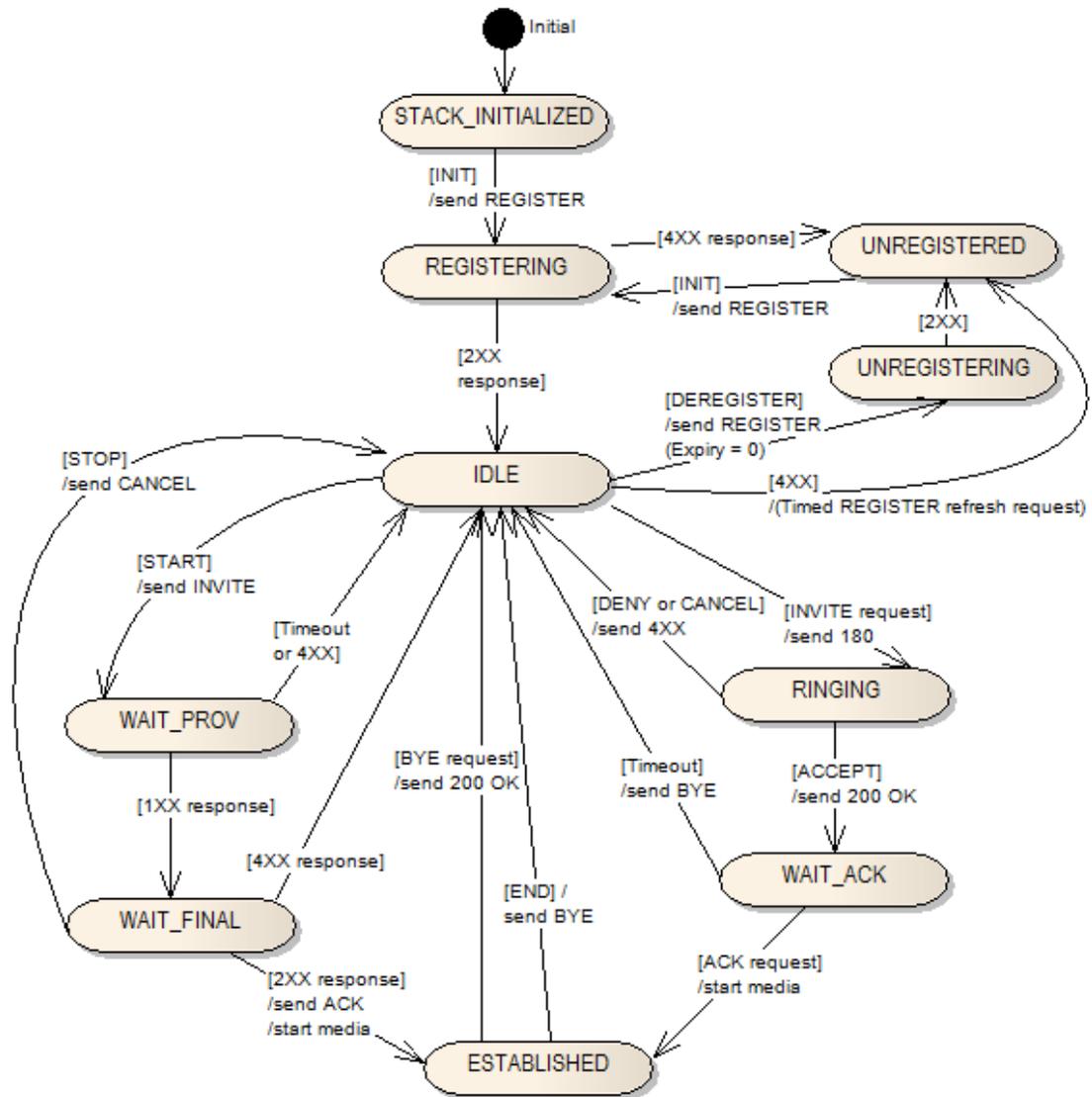


Figure 2.7: SIP stack state diagram according to SIP events

From the applications perspective however the SIP stack becomes operational after an initiating SIP\_INIT event which has to be sent after the user logs in successfully (on SECURITY\_USER\_AUTH\_OK event response) so the SIP stack may read the user's preferences from the database. The controller should load any required configurations, classes and libraries in order to be ready to receive or send SIP messages. If the initialization succeeds or the stack is already loaded the SIP\_INIT\_OK event has to be sent in response. Otherwise the SIP\_INIT\_FAILED

event has to be sent. This applies also to the SIP\_RESET event which (if the stack was loaded before) should de-register the UAC from the server, unload the stack, end the current call in progress (if any) and reinitialize it as if a SIP\_INIT event would be sent. The SIP\_RESET event should be triggered by an incoming NET\_CONNECTION\_ESTABLISHED event which signals that the underlying network connection has changed and the SIP stack needs to be reinitialized with a new IP address and a corresponding registration request.

Figure 2.8 extends the original state diagram in Figure 2.7 to display the SIP stack state transitions from the application's point of view (that is in form of events). This means that every state transition from the STACK\_INITIALIZED state upwards represents a mapping between SIP messages and application events and vice versa. The square brackets indicate the direction of the event where [in] stands for incoming events and [out] for outgoing events (from the stack's perspective).

To not overload the diagram it is missing some transitions from the in-call states to the IDLE state on a CALL\_ERROR event which indicates a software or hardware failure. The same applies for the SIP\_RESET event where transitions from all states (except the starting one) to STACK\_NOT\_INITIALIZED should be present.

The security aspect of this signalling architecture is integrated in the IDLE to WAIT\_ACK state transitions logic for incoming calls and the WAIT\_FINAL to ESTABLISHED state transition logic for outgoing calls. This is due the fact that the application logic of these transitions operates on SIP INVITE messages that are used for establishing a SIP session. To these messages (or the 200 OK responses to them) a session description protocol (SDP, RFC 4566) message body is attached which is usually used to negotiate the destination IP address and port for media packets as well as the audio codec to be used and other session parameters. It is here, before the media transmission itself, where the architecture's security related measures are hooked into. The security hook has two tasks – a) exchange of session cryptographic keys and parameters and b) the authentication of the remote party that sent the invite or the positive response to it. This is achieved by a little modification of the purpose of the key (“k:”) field in the sent SDP messages (or “piggy backing“ if you will). While this field is normally also used for key exchange the data format the design prescribes is based on an encrypted and Base 64 encoded `SipCryptoSpecsAttachment` prototype buffer message. The message as shown in the source file Listing 2.5 carries all cryptographic parameters needed for securing the data sent in the media plane (basically task a)).

Note that the fields defining the data integrity hashing algorithm for the data packets are the optional `digestAlgorithm` and `hMacAlgorithm/hMacKey` fields and only one of them must be set at a time (since only one method can be used at once). The session key and the initialization vector must be randomly generated by a secure random number generator and their size must match the algorithm used (e.g. 256 bit). Now because the `SipCryptoSpecsAttachment`



```

1 message SipCryptoSpecsAttachment {
2     required string cryptoAlgorithm = 1; // Symmetric algorithm
3     required string blockMode = 2;     // Block mode
4     required string padMode = 3;      // Padding mode
5     required bytes  sessionKey = 4;    // Session key
6     optional bytes  iv = 5;           // Initialization vector

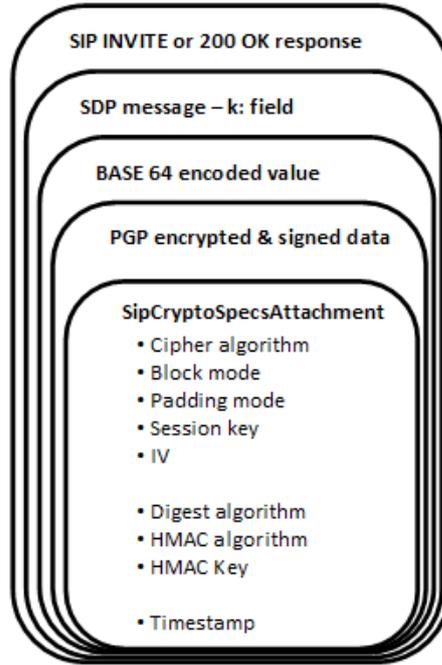
8     // Digest or HMAC (optional)
9     optional string digestAlgorithm = 6;
10    optional string hMacAlgorithm = 7;
11    optional bytes  hMacKey = 8;
12    required int64  timestamp = 9;
13 }

```

**Listing 2.5:** SipCryptoSpecsAttachment protocol buffer message

The basic idea of secure session establishment is therefore based on piggy-backing session data and using asymmetric PGP encryption and signatures to secure it. The complete process of the whole security hook is more complicated and thereby recapitulated in the following list. From the session initiator's point of view the responsibilities of the SIP stack are in order:

1. Retrieve the contact the user tries to establish a data session to from the database.
2. Build a `SipCryptoSpecsAttachment` message and fill the fields based on the user's data encryption preferences. Update the `timestamp` field with the current timestamp (section 2.5 explains why).
3. Build an INVITE message with a SDP message body. Set the needed SDP fields (audio codec etc.) and fill the "k:" field value with the encrypted and signed `SipCryptoSpecsAttachment` object as described previously.
4. Send the INVITE to the contact's SIP address through the SIP server and wait for provisional (1XX) or final (200 to 600) response. Notify the SIP stack to not accept any messages that are not from the address we try to contact.



**Figure 2.9:** Session key exchange message layers

Now on the receiver side, assuming his SIP stack is idling and no other session is in progress, the process is as follows:

1. On an incoming INVITE request check that the sender is in our database. If yes, retrieve the contact information and continue in 2). If not, deny the call automatically with a 433 (anonymity disallowed) error response code.
2. Decode the SDP message body. If none is found or it contains errors, deny the call with a 421 (extension required) response code. If the audio codec is not supported, deny it with 411 (unsupported media type) response code.
3. Decrypt the `SipCryptoSpecsAttachment` from the key field of the SDP attachment. If none is found return again with 421 code. If the decryption with the user's private PGP encryption sub-key fails signal it to the remote party with a 493 (undecipherable) error code and go back to idle. The same applies if the PGP signature (including data integrity protection) could not be verified or the signature's PGP key ID could not be matched to the remote contact's public signing key.
4. The time difference between the current time and the `timestamp` field value of the decrypted `SipCryptoSpecsAttachment` object must be less than 1 hour (should be configurable). If it is more, show a replay attack warning to the user in point 5).
5. Only if all conditions were met signal the user an incoming call and send a provisional 180 response.
6. Should the user accept the call, send a 200 OK response with a new SDP/`SipCryptoSpecsAttachment` attachment. The new `SipCryptoSpecsAttachment` object is to be built similarly to the previous description. The

audio codec value must be copied from the incoming session description parameters.

7. Notify the SIP stack to not accept any messages that are not from the address that tried to contact us. Wait for the final acknowledgment (ACK) request from the session initiator. Start a timeout which expires after 30 seconds by default (should be configurable). If no ACK confirmation is received the timer must set the SIP stack back to idle and notify the user.

Back on the sender's side the stack is waiting for a response to continue the process. Should the receiver deny a call for some reason end the connection effort and notify the user with an error message corresponding to the received 4XX code. If a positive (200 OK) response is received the stack must exercise the same security checks as described on the receiver's side (points 1. to 4. in the previous list) only instead if the INVITE request the 200 OK response is processed. Also any failures do not result in sending 4XX codes back to the receiver but instead letting the ACK timer on the receiver side expire. In this case the call must be terminated immediately on the user's side with an appropriate error message. If the receiver's response is verified the session is to be considered as established by sending the ACK request.

### **Additional security measures and SIP server analysis**

Additional security measures to the otherwise insecure (plain-text) session initiation protocol include that each SIP message should be communicated over a secured TLS connection established to the SIP server. As a second level fail-safe each message must be signed with a PGP certificate created by the message sender and verified by the receiver to prevent an attacker of sending false commands. This insures message authenticity and integrity. The certificates should be basically a hash value computed from the attributes of the SIP message and must include a current (universal time) timestamp to prevent message replay attacks. Messages that have a valid certificate but a timestamp older than one hour must be discarded automatically. For call initiating messages the application should display the normal call accept/end screen but with a warning saying that the request is outdated and therefore probably forged.

At the moment the PGP signature conditions should not apply to REGISTER requests which are negotiated only with the SIP server as this would require rewriting portions of the server code in order to check for these signatures. The SIP registration process should be executed in the standardized way using digest authentication (with MD5 hashes) as defined in the SIP RFC document ([15]). See the security analysis section 2.5 on how this may impact the safety of the signalling plane. Removing this requirement has also the effect of a free and interchangeable selection of the SIP server software. After some research the Open SIP Server or just OpenSIPS (formerly OpenSER, [16]) has been favored over the lightweight MjSip proxy ([17]) which was the first choice for prototype applications. OpenSIPS is in contrast to MjSip maintained by a professional community and provides in addition to the standard components carrier-grade features including SIP presence, messaging or load-balancing. Nevertheless the

most critical functionality is that it can serve also as a SIP and media NAT traversal unit which will be important for the media plane as explained in the next chapter. In general though the final implementation is more or less free to choose the SIP server implementation as long as it supports NAT traversal and is conform to the RFC standards. It might be even possible to widen this freedom of choice to the whole signalling protocol selection since the architecture is bound to events and not explicitly SIP messages. Would the architecture use for example XMPP or H.323 signalling protocols only the mapping between events and the appropriate message processing stack would have to change (disregarding some minor data format and GUI changes). The media plane implementation is abstracted from the signalling protocol and requires only secure session establishment and NAT traversal. In the end the reasons for selecting SIP were its maturity, broad selection of server software and the compatibility option to most existing VoIP applications. Also the option to use SIP alongside XMPP in the future exists. A comprehensive comparison of SIP to XMPP can be found at [18].

#### 2.4.4 Media plane

After the session has been established the application can be sure that the remote party is authenticated, has accepted the transmission request and both ends possess the correct session description parameters including session keys. By sending (or receiving respectively) the final ACK request the data transmission on the negotiated ports must start. For the transmission itself the most commonly used protocol in conjunction with SIP will be used - the Real-Time transmission protocol (RTP). RTP is specified in RFC 3550 [19] and belongs to the user datagram protocol (UDP) family.

The design will use the protocol “as-is“ with minor adjustments to the payload format which has to be encrypted. It therefore does not adhere to the RTP payload specifications as stated in RFC 3551. Instead, as anywhere else in the architecture design, encrypted Google protocol buffer messages will be used as the RTP packets payload data format (Listing 2.6).

```
1 message RtpPacketPayload {
2   required bytes data = 1;
3   repeated int32 framesSizes = 2 [packed=true];
4   optional bytes signature = 3;
5   optional int64 timeCreated = 4;
6 }
```

**Listing 2.6:** RtpPacketPayload protocol buffer message

The format is generic enough to carry voice, video or any other binary formats requested by the application. All data is to be stored in the byte array data field of the message. The data is basically anything what would be in the original payload for an RTP packet if no encryption would be used, e.g. recorded audio or video frames. When frames are used they are all packed one behind the other in the data field. To split them on the receiver end correctly the `framesSizes` field is an array of integers which define the size of each frame in bytes. If no

frames are used it may be left empty or contain only a single value. An additional timestamp is stored in the `timeCreated` field to enable recorded calls playback.

After these fields are set a signature of the data is to be computed and stored in the signature field. This is to detect modifications of the packets on their way to the receiver. Signatures are in the form of hashes generated either by digest or the more secure HMAC algorithms. The message is then to be built and encrypted with the session key obtained from the SDP attachment sent by the corresponding remote party in the negotiation process. The encrypted body serves then as the RTP packet payload.

For calls the transmission should take place in at least two parallel processes, one for processing incoming data and one for sending the recorded audio. Both processes should be started in their own thread in order not to block the `EventHandler` and to process audio recording as well as playback simultaneously. The processing of incoming RTP packets can be decomposed in the following steps:

1. Receive a RTP packet and queue it to prevent packet loss. The packets should be sorted into a queue according to the RTP timestamp to catch out of order packet deliveries.
2. Take a packet from the queue, decrypt the payload with the session key and parse the `RtpPacketPayload` message from the bytes. If it cannot be decrypted or the message cannot be built throw the packet away.
3. Calculate a hash of the `data` and `timeCreated` fields using the negotiated algorithm and compare it against the hash in the signature field. If they do not match throw the packet away as it has been damaged or tampered.
4. Check that the timestamp in the `timeCreated` field is in the allowed range (e.g.  $\pm 10$  minutes from now) to prevent replay attacks.
5. If the payload has the correct format and the origin of the sender and packet has been verified split the `data` into frames with help of the `framesSizes` array.
6. Decode each frame using the audio codec defined in the RTP payload type header field or as established in the SDP body of the INVITE SIP message (they have to match). The result should be pulse-coded modulation (PCM) audio samples.
7. Obtain the audio playback device through HAL and play the samples. Repeat from point 1.) until a `CALL_END` or `CALL_ERROR` event is received by the process. The process has of course to register itself as a runtime listener in the event handler first.

Sending audio data is basically the reversed process of the previous one:

1. Record raw PCM samples from the recording device obtained through HAL and encode them with the previously agreed on audio codec to frames. Remember each frame size in an integer array and concatenate the frames into a single byte array.

2. After there are enough frames to fill the RTP payload build an `RtpPacketPayload` message and fill the `data` and `framesSizes` fields. The final RTP packet must not be bigger than 1500 bytes which is the usual standard maximum transmission unit (MTU) for UDP packets. Therefore build a packet when the concatenated frames are about 1 kB in total to leave room for headers and other fields (plus some reserved space to be on the safe side).
3. Store the current timestamp in the `timeCreated` field. Calculate a hash of the `data` and `timeCreated` fields and store the result in the signature field. If the hash algorithm is a HMAC use the session key.
4. Build the protocol buffer message and encrypt it with the session key using a symmetric algorithm as specified by the user's preferences.
5. Create an RTP packet and set the correct payload type according to the used audio codec along with other header fields. Send the packet through an UDP socket to the receivers IP address.
6. Stop on `CALL_END` or `CALL_ERROR` event. Send a `CALL_ERROR` event in case the process is interrupted in some unexpected way.

A significant role in exchanging voice data has the audio codec. Audio codecs usually compress the raw PCM samples or at least convert them to a standard format (such as G.711). The implementation must support at least the narrow-band speex codec [10] and both A-law and  $\mu$ -law variants of the G.711 wideband telephone codec. The user should be able to select a preferred codec for outgoing calls (due to the SDP offer/accept mechanism the codec selection depends on the settings of the caller, not the callee).

Encryption settings used in the processes on the other hand are defined separately by each user and exchanged along with the session key in the SDP message body (Figure 2.9). As a result the symmetric encryption algorithm for outgoing data transmissions may be different or may have different specs than the incoming transmission. Users should be able to select their preferred symmetric algorithm from a predefined set and if applicable the cipher's key size, block mode and padding mode. The default preference should be set AES algorithm with a 128 bit key size in cipher block chaining (CBC) mode with public-key cryptography standard #7 (PKCS7) padding.

The same logic applies to the hash algorithms where one user may select digest hashes only and the other a HMAC algorithm. See also Table B.1 (p.78) for the full names and settings of the user preferences.

This is beneficial e.g. in comparison to the secure variant of RTP (SRTP) as each user defines the level of security for his data. Also SRTP uses only 128 bit AES encryption with a SHA-1 HMAC (RFC 3711). The described approach supports a variety of other algorithms, many of them cryptographically stronger than the ones used by SRTP and it is only up to the implementation to include them. This was the main reason why a custom approach of data security was selected over using the SRTP standard in spite compatibility issues (see section 2.6).

The presented solution is versatile and may be easily modified for transferring other data types. For example to include video calls the process logic would stay the same, the only thing that would change is the used codecs and playback-/recording devices. Nevertheless until UDP is used as the transportation layer NAT traversal is an issue that had to be solved in the architecture. For that the TURN (Traversal Using Relays around NAT) approach has been adopted. In short, it assumes a cooperation between the SIP server and a server software named “rtpproxy“ (which is included in OpenSIPS as a pluggable module). In conjunction they rewrite the destination IP addresses in the SDP message bodies to the IP address of the rtpproxy server. Effectively this means the endpoints are deceived on purpose to route all RTP traffic through the proxy instead of directly to the remote party. This has the advantage that the RTP traffic is capable of travelling through all types of NAT modes (even symmetric) at the cost of adding a delay and utilizing resources for the rtpproxy. Hence in this case the priority was set on 100% reachability rather than effectiveness. See [20] for full specification on the TURN solution using OpenSIPS and a comparison to the STUN (Simple Traversal of UDP through NAT) technology.

#### 2.4.5 Call history and recording

Each data transmission should be logged for informational and auditing purposes. As the architecture aims in this version mainly at voice calls, the call history should be kept in the predefined structure as shown in Figure 2.3 where every `User` entity has a list of `CallLogEntry` prototype buffer messages which model the database entity (Listing 2.7). A controller should listen to events starting with the `CALL_` prefix and create and save a `CallLogEntry` entity for each attempted or completed call. The entity has to be saved only after the call ends to correctly set the call duration attribute. Attempted calls (such as missed and denied calls or calls ended abruptly with a `CALL_ERROR` event) should have the duration field set to zero (0). The `timestamp` field should contain the UNIX timestamp of the time when the first call event type was received and updated when a call has been established (on `CALL_INCOMING_ESTABLISHED` or `CALL_OUTGOING_ESTABLISHED` event). For whatever reason the call ends the `CallLogEntry` entity has to be persisted in the database component with the correct call direction attribute value selected from the `CallDirection` enumeration. The `contactId` is a foreign key that holds the ID of the contact entity the call was directed to or from.

```

1  message CallLogEntry {
2
3      enum CallDirection {
4          INCOMING = 0;
5          INCOMING_MISSED = 1;
6          INCOMING_DENIED = 2;
7          OUTGOING = 3;
8          OUTGOING_CANCELED = 4;
9          OUTGOING_DENIED_OR_BUSY = 5;
10     }
11
12     required int64 contactId = 1; // foreign key to Contact.id
13     required int64 timestamp = 2;
14     required int32 duration = 3; // (in seconds)
15     required CallDirection direction = 4;
16     optional bool recorded = 5 [default = false];
17 }

```

**Listing 2.7:** CallLogEntry protocol buffer message

The recorded flag has to be positive if the call or a part of it was recorded by the user. The basic concept of call recording is that received and sent RTP packet payloads holding the voice data (encrypted `RtpPacketPayload` message, see previous chapter) are cloned to two arrays for incoming and outgoing packets. This starts when the user is in-call and presses a recording button which results in sending a `CALL_CURRENT_RECORD_START` event. A controller has to be responsible for processing events related to call recording as well as the recording logic. On receiving the event the controller should signal the media transmission processes to send it a clone of each packet payload (in the plain encrypted form). Success or failure of initializing the recording process has to be signaled to the GUI with an `OK/ERROR` event (see Table A.1, p.77). The recording may not be continuous and can be stopped or paused with the `CALL_CURRENT_RECORD_STOP` event. Repeated sending of the events should cause the recording process being resumed or paused until the call ends. Not later than when the call ends the recorded packets have to be put in the `RecordedCallHolder` prototype message including a SHA-256 HMAC as the signature. The HMAC signature should use the database password and hash the serialized `RecordedCallPackets` object which holds the arrays with the RTP packet payloads. The `RecordedCallHolder` object should then be serialized and flushed to an external storage drive. On success metadata has to be written to the database. The metadata consists of copies of the voice session properties including SDP message bodies and cryptographic parameters for both data transmission directions. See the `RecordedCall` message definition for details. All the aforesaid prototype buffer message definitions can be found in the attachments section (Appendix D, p.83).

Note that recording a call is local to the endpoint. The other endpoint is not notified of the recording. Also the metadata records in the database as well as the recording file should perish if the corresponding user or call history entry is removed from the database.

Users must also be able to replay their recorded calls. The playback is initiated by the `CALL_RECORDED_PLAYBACK_START` event type with the `timestamp`

of the recording as the argument which is also the primary key of the `CallLogEntry` entity. The responsible controller needs to check the HMAC signature of the whole recording file and while in playback check also each replayed RTP packet payload's hash to ensure that the recording was not modified as it is not stored in the database (only the metadata is). `CALL_RECORDED_PLAYBACK_PAUSE` and `CALL_RECORDED_PLAYBACK_STOP` events should pause/resume and stop the playback.

## 2.5 Security analysis

This chapter summarizes the security measures proposed by each component of the architecture design and will look into possible attack vectors and security threats (marked bold throughout the following text). It assumes at least basic understanding of current cryptography systems and algorithms as well as computer security terminology (although this applies to the thesis as a whole).

As the cornerstone of the proposed security scheme is aimed at asymmetric cryptography or more precisely on the Open PGP specification it is assumed that as of today no fundamental publicly known security flaw in the principles of the used (a)symmetric cipher algorithms and the PGP scheme itself is known. Of course this may not apply to the program code implementations of the algorithms which may or may not be flawed in terms of security. Furthermore it would be outside the scope of this text to analyse each cipher and cryptosystem one by one so please refer to the numerous existing publications on that topic (see for example [21] which analyses PGP's security). Instead the focus of the security analysis lies on the usage of these secure algorithms in a given context and whether they provide a compact defence system against known attacks. To summarize, the following assumptions and simplifications are made in this analysis:

- A “cryptographically strong random number generator“ (RNG) is available. This means it is not predictable or suggestible.
- Public key asymmetric cryptography using RSA and PGP is perfectly secure if used correctly (a strong enough password provided).
- The AES symmetric encryption and SHA-2 HMAC algorithms are perfectly secure to the degree of the quality of the password and the random number generator used.
- No quantum computers are available to the attacker which would weaken or render useless the deployed algorithms.
- Used third party libraries or software and cryptographic function implementations contain no back-doors or security flaws.

An additional note on the RNG: Since a “true“ RNG might not to be available to an implementation it is assumed that the weaker “cryptographically strong“ random number generator is sufficient. A custom RNG implementation may be developed in the final implementation e.g. by collecting and combining entropy from files on the hard drive and built-in sensors like the gyroscope or compass

(magnet field strength) sensor. However this is discouraged unless it can be proven that the data is truly random and the RNG complies at least with the “cryptographically strong” specification as described in the FIPS 140-2: Security Requirements for Cryptographic Modules and RFC 1750: Randomness Recommendations for Security documents. Instead proper use of existing and verified solutions such as Java’s SecureRandom class is recommended (see e.g. [22]).

Now when the security of the RNG is established almost all security features unveil themselves from the **user’s password** and its strength. It is therefore to be analysed first. As a reminder, the user’s password is the result of the SHA-256 hash method which input is the concatenated bytes of the username (at least 3 characters), a large factorized number ( $>2^{762}$ ) and the serial number of a device (may be empty). The factorized number is calculated as the product of at least six Mersenne prime numbers starting at the smallest used Mersenne prime 12. Its value is  $2^{127}-1$  so the declared power 762 of the factorized number comes from the requirement of using at least 6 numbers for the password ( $6*127$ , see 2.3). The final password is used to encrypt the database file with AES-256 and to create a SHA-256 HMAC that ensures the integrity of the file.

Supposed an attacker managed to obtain the **database file** there are two ways he could crack the user’s password. The first way is a direct brute-force attack on one of the algorithms which is mathematically proven to fail with current (or even near-future) hardware equipment. Failing in this sense means that while this attack would theoretically yield a result it is impractical in the amount of time needed to obtain it. This is also where the described password derivation method shows its strengths. For once it does not use a character passphrase so the attacker is unable to employ a dictionary attack which could dramatically speed up the brute-force attack. The user simply cannot enter a weak password (such as “password“). Secondly unique (per-user) cryptographic salts are used thereby disabling pre-calculated rainbow table attacks on the SHA hash. From this perspective the approach to store data is perfectly secure.

A smarter attacker though may choose a different way of breaking the user’s password. Since the only dynamic part in the final password is the number gained from the prime product part the attacker may focus only at it with a more directed brute-force (factorization) attack. The attack would consist of trying out all prime-index (or color if you will) combinations of the password which for a password of the length 8 is less than  $8^8$  combinations (eight primes<sup>^</sup>eight color selections). Though this way assumes the attacker has in addition to the database file itself a) knowledge of the inner workings of the password derivation algorithm, b) the prime numbers used and c) the user’s username and his device’s serial number, it is not to be dismissed as these information may be easy to come by.

While the second attack vector is significantly faster than the first approach and realizable in practice the system still provides a relatively high level of security. It assumes extensive knowledge of the application implementation and access to the needed data, plus for the attacker to try all password combinations which only for password lengths 6 to 16 means to try out somewhat less than  $\sum_{i=6}^{16} 8^i =$

$3.21685687631872 \times 10^{14}$  combinations. Meanwhile there is nothing preventing the user to still have a simple but more effective password e.g. 5 colors with 5 clicks each (about  $4.3 \times 10^{21}$  combinations in total) or even more. As an added benefit the user is prevented from entering dictionary-like passwords and the color-based password inputting mechanism presents a small added value to the security scheme (similar to gesture unlocks on mobile phones).

The impact of an eventual password breach on the security system would be fatal as the database file contains all private data including the PGP private key ring and the password to it. Even worse, such a security breach would mean not only loss of data confidentiality but allows the attacker to impersonate the victim as identity and message authentication is based on PGP signatures. Moreover with a proper set-up **man-in-the-middle (MITM) attacks** could be possible to eavesdrop on calls because session keys for the media plane are exchanged in PGP encrypted objects.

MITM attacks apply also to the **public key exchange** channels which do not require the public key rings and contact information to be encrypted. While an attacker might in some cases intercept these messages and exchange the keys for forged ones the impact is next to none and requires only that the users double-check the fingerprints of the received keys (generating a fingerprint collision for two different OpenPGP key rings is very, very unlikely).

Using **OpenPGP key rings** as the central piece of the security scheme opens also another attack vector. Should the user store (or export) his private key ring somewhere outside of the database the attacker may try to brute-force the key ring password. The security level in this case greatly depends on the key ring password strength. For PGP key rings generated by the application this is not an issue as the key ring password is essentially the same as the database password (only additionally Base 64 encoded so the user may enter it as a textual passphrase when using other PGP related software) therefore the password strength analysis of the previous case is applicable. However the application should allow the import of existing key rings as long as they fulfil certain requirements. The requirements are that the key ring must contain at least two keys that are each at least 1024 bit strong (one master signing and one encryption sub key, see 2.4.2). This requirement is enforced in order not to weaken the security scheme with weak imported PGP keys. The problem in this case is that no requirement on the key ring password is set so the user can reuse his existing key ring which he has potentially already distributed and is being trusted to by other parties. Hence the key ring security is a responsibility of the user. Regardless of where the responsibility lies, should such a custom imported key ring's password be obtained by the attacker the security impact would be almost as great as in the previous case. While the attacker would not gain access to the database (and therefore no stored private data including recorded calls) he still could impersonate the victim or eavesdrop on his communications.

Eavesdropping VoIP systems is possible only by intercepting voice data packets which in the given architecture are properly encrypted. Therefore for an attacker to eavesdrop in the given architecture design the messages in the **SIP signalling**

**plane** that carry the session parameters for the data exchange must be captured first. SIP in general is substantially insecure as for example analysed in the official U.S. National Institute of Standards and Technology (NIST) report ([23]). Due to its plain text nature it is unprotected and susceptible to any imaginable attack ranging from message forgery/tampering and false SIP registrations up to loss of data confidentiality. The NIST report suggests using TLS sockets to secure the signalling plane which is also a first-level measure employed by the architecture. While TLS alone should be enough to secure SIP the proposed architecture goes a step further and adds another layer of security. The main reason is that the attacker still has three opportunities to circumvent the encryption used on TLS channels. For one he could use a 0-day exploit on the TLS stack to gain access to the plain-text messages. Furthermore he could seize the **SIP server** instance and use it to capture the messages since the encryption is only between the UAC endpoint and the server. At last he could exploit a situation that prohibits the use of TLS in favour of UDP socket for some reason (e.g. NAT traversal issues). Concurring from these attacker's options the architecture employs a second-level safety measure based on PGP. This includes each SIP message being signed with a PGP certificate so it is not modifiable without the modification not being detected by the endpoint in which case the message is immediately thrown away by the SIP stack. Also the cryptographic parameters for a session are encrypted with the public keys of each participant making them practically unreadable to the attacker. Basically the SIP stack in the architecture makes no assumptions of the security of the SIP message transport and/or the SIP server and relies purely on PGP for message authenticity and integrity (see 2.4.3). So even if the attacker has managed to exploit the TLS security layer (or the SIP server) the added security measures prevent him from modifying the SIP messages or reading the piggy-backed symmetric session keys from the SDP message body.

It is thereof safe to say that unless the attacker has no access to the private PGP key rings of the participants or a fundamental flaw in the OpenPGP cryptosystem is discovered that the data confidentiality in the signalling plane is ensured. On the other hand anonymity of the participants can be secured only if the TLS layer is properly employed and the SIP server is secured. Otherwise the attacker, whilst still unable to decrypt the contents of the data session, can read the SIP messages and obtain potentially useful call "metadata" so to speak such as the (IP) location of the user or on when and by who a call was placed.

All the employed security measure however cannot prevent a **denial of service (DoS)** attack. Should the IP address of a user (and his SIP stack) be available to the attacker with enough network bandwidth and computing power he is able to effectively exclude the user from any communication (even if the messages are thrown away at once). This applies also to the SIP server which in which case the DoS would affect all registered users. Also in case the SIP server is taken over by an attacker he may just simply discard or redirect any received SIP message from a given set of users. This is a general problem of network based services and no final (endpoint) solution is known to the author. A proper DoS defensive configuration of the SIP server may alleviate the issue a bit but only in case the SIP DoS attack is routed through the SIP server and does not hold if the attacker targets the user's endpoint directly.

The **media plane** uses by default at least AES-128 symmetric data encryption and message digests or HMACs for integrity checks and authenticity of the encrypted packet data. With different (random) session keys for both directions given a secure RNG no computational attack is plausible from this attack vector besides the mentioned interception of the session keys. This applies also to other symmetric algorithms eventually provided by the application and selectable by the user at each endpoint separately. The user is simply not given the option to choose an algorithm that is considered weak or insecure. For performance reasons however the user might choose not to create a key secured HMAC for each RTP packet sent but only a hash digest such as MD5. The hashes are on the receiver's end used to verify the data integrity of the arriving packet and in case of HMAC's also the authenticity of the packet's origin (since the HMAC is created with the session key). Although using digest hashes may increase performance and does not affect data confidentiality it opens an attacker a window to tamper with the integrity of the transmitted data. By a MITM attack on the local network or the rtp proxy part of the SIP server (2.4.4) the attacker might destroy, garble or replace the data packets and create a new digest stamp to hide the modification. This could in addition to data loss result in **buffer overflows** in the application and/or potential code injection (though probably not in Java). Thereby on sufficiently capable hardware the HMAC option should always be used. Packet HMACs are also a countermeasure against **replay attacks** from which an attacker might obtain potentially useful information or simply block the user's line. This applies also to the signalling plane.

The last threat in this area is again a DoS attack that could prevent the users from exchanging data.

The last section of the analysis will discuss security threats originating from outside the architecture's range. Attacks on the **underlying operating system** belong in this classification. An attacker with sufficient OS rights (e.g. root privileges) might easily do a memory (RAM) dump of the running application's memory space. Since all data from the database is in runtime loaded into the memory a simple data analysis would reveal all confidential data including passwords to the attacker. The security fallout in this case is the same as in the first analysed case where the attacker cracked the password. Note that this applies not only to desktop systems but mobile platforms as well. For example the iOS can be "jailbroken" and for Android smartphones a number of privilege escalation ("root") exploits is available. In addition **hardware attacks** targeting the memory physical memory modules are possible, though a great deal of expertise is required (such as a cryogenic freeze and imminent data dump of the RAM modules). Simpler but not less effective software and hardware hacks include malicious **keylogger** software in some form that might record the user's clicks and input or just simply take a screenshot of the password after it has been entered. In this regard the application is fully exposed to the security measures of the underlying system and of the user administering the system. Some even more "**trivial**" attacks fall in this category such as a planted bug, eavesdropping with a directed microphone array or plainly catching the user in the act of entering his password.

**User behaviour** in general is a great security risk as a much of the system's

security relies on it. Users might select relatively weak passwords (e.g. for the imported PGP key ring) or leave the running application unattended so an attacker has unlimited access to it in the user's absence. And of course the greatest security architecture can fail if the user writes down his password on a Stick-it note next to the device or succumbs to a social engineering attack.

Disregarding the attacks the architecture is by principle unable to fight back, under the conditions listed at the beginning of this analysis the security architecture provides a great deal of security on multiple levels and relies only on the endpoints clients. The only 3<sup>rd</sup> party involved in the scheme is the SIP server and should it be compromised the impact is limited (see SIP TLS circumvention attacks). But even if data confidentiality in the signalling plane is ensured it is still one of the most vulnerable points of the architecture (indifferent whether with or without the server) as it is susceptible to DoS and user identity/location identification attacks. The absolutely most vulnerable point is the user who generates or imports weak passwords into the otherwise safe system.

## 2.6 Compatibility

To summarize, implementations of the architecture as it was proposed in the text will be compatible between each other indifferent of the underlying platforms. The key reasons for that are the usage of universal network protocols such as SIP and RTP and mainly the employment of Google protocol buffers as the data format for everything the application persists or exchanges. Programming libraries for protocol buffers are available on a number of platforms including all the key ones. Because of that it is possible to export the database including the recorded call files from an Android smartphone and import it into an iPhone<sup>TM</sup> without any customizations on the data (which might not be possible using e.g. native SQLite databases). The protocol buffer library even takes care of little and big endian conversions when transferring data between platforms with a different endianness.

Using established codecs for audio or video transfers is another key point as they not only compress the data (in most cases) but guarantee that each platform can decode the frames to a format it understands.

Of course deployment of custom solutions comes at a price. While the audio codecs used (speex, G.711) are compatible to existing applications (see 2.8), the RTP packets payloads are not as they are packaged into protocol buffer messages which none of the applications would understand. Also the encryption algorithms used are standardized on all platforms but no standard security mechanisms such as ZRTP/SRTP were adopted. This is mostly due to the limited functionality and limited variety of encryption algorithms in contrast to the requirements of the proposed secure communicator. Another factor is that development of ZRTP is stalled at draft stage (RFC 6189) and there are only few freely available and stable implementations for even less platforms (the latter includes also SRTP). However using SIP for signalling has the added benefit that most of the existing implementations depend on it also and can be seen as the common denominator.

So while the architecture in the current design is not compatible with existing applications because of the mentioned reasons it could be extended with a compatibility mode to allow non-encrypted calls or calls using ZRTP/SRTP. The signalling layer would then have to change only marginally by temporarily disabling or modifying the authentication and key exchange mechanisms.

## 2.7 Android platform limitations

The prototype implementation as described in chapter 3 will be developed for the Google Android platform for smartphones. Constrained hardware resources such as memory and processing (CPU) performance on mobile platforms are given by the limited space and battery life. Android is no exception to this. Preliminary benchmarks have shown that available hardware at the time of writing may not be capable of simultaneous encrypted data transfers in satisfactory quality or speed. The analysis therefore focused on the primary function of the secure communicator, which is making secure calls. This is also where the implementation's focus should lie.

Another limitation of Android is that the Universal Serial Bus (USB) interface is not programmatically accessible through the platforms API at levels<sup>7</sup> below 12 which narrows the list available channels for data exchange. Furthermore the networking channels that are left are limited by Android's connectivity manager<sup>8</sup> which allows using only one channel at a time. The manager does not even allow a manual selection between e.g. 2G/3G or wireless connection despite the fact that they are established on distinct network adapters.

The listed concerns have been considered in the architecture proposal and should be considered during the implementation and evaluation phase of the final application prototype.

## 2.8 Related work

A lot of research already went into cryptography and secured (VoIP) communication and the thesis leverages this knowledge to build a new and safe architecture of a secure communicator. For example Phil Zimmerman, the creator of the PGP standards which are also used as a component of the architecture in this thesis, tried to approach the mentioned security issues by creating the PGPfone software and later on the ZRTP standard. His Zfone project's software was probably the first VoIP client that used ZRTP. The project however is now discontinued and only a limited software development kit (SDK) can be obtained, not the software alone. He is now actively part of the Silent Circle venture that – at the time of writing – is planning to provide encrypted call and messaging services for mobile devices. Silent Circle is based on Zfone and there are surprisingly only a handful

---

<sup>7</sup>See <http://d.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels> for explanation on Android API levels.

<sup>8</sup>Implemented by the `ConnectivityManager` class as defined in <http://d.android.com/reference/android/net/ConnectivityManager.html>

of other implemented secure VoIP solutions that do exist for mobile platforms. The vast majority of them use TLS sockets to secure the SIP signalling layer and/or ZRTP/SRTP standards to secure the media plane. This approach being a standard has its advantages, but it has also drawbacks as pointed out in section 2.6. As an example CSipSimple has to be named as the only open-source (GPL) solution with these parameters. Proprietary solutions include the iCall software exclusively for the iOS platform or Bria that is also available for Android and others. The popular Skype software offers call encryption, however it is suspected that calls made through it can be monitored by government agencies (taken from e.g. [24]).

In comparison the number of desktop VoIP clients or clients that use no data encryption is greatly larger, e.g. SipDroid or Viber to name a few. A clearly arranged table with an overview of all related software products may be found at [25].

The proposed architecture tries to walk down a different path than the listed solutions and the prototype implementation will try to confirm that it is not a dead end.

## 3. Prototype implementation

The prototype implementation of the architecture as described in the previous chapters is designated for the Google Android platform to see whether the approach is viable on one of the currently leading mobile platforms. The goal is to show that it is well possible to build such a communicator application that is as secure as it is usable on current smartphones. This chapter gives insight into the application structure from a software development point of view beginning with the project source codes structure in section 3.1. The major components as pointed out in chapter 2 are described accordingly in subsections of section 3.2. If not explicitly stated otherwise the implementation of the components sticks straightforwardly to the design in the analytical part of the thesis. As a detailed description of the source code would yield no added value for this thesis the text focuses on the general features of the implementation. If needed, implementation details can be reviewed in the program documentation or directly in the source code of the application. Both are to be found in the appendices section together with a short user manual (Appendix C).

Implementation specific security issues are discussed in section 3.3. The last section 3.4 contains instructions on how to test and build the application (although a binary build is present on the attached CD-ROM). A final evaluation on the capabilities and eventual shortcomings of the prototype is given in the final chapter 4.

### 3.1 Project and package organization

The secure communicator application has been nicknamed with the working title “CryptDroid“ and given a logo (Figure 3.1) along with an internet domain (cryptdroid.net). The domain is used for aliasing the IP address of the SIP server and SIP account registrations. From now on the application prototype may be referred to simply as CryptDroid.



**Figure 3.1:** CryptDroid project logo

CryptDroid was developed using the Eclipse integrated development environment (IDE) with the Android Development Tools (ADT, see [26] and [27]) plugin installed. The application itself is composed of three separate Eclipse Java projects which mimic the basic layout of the architecture and three supporting projects. The application’s projects are as follows:

1. **cryptdroid\_3rdparty\_lib** – Stores the used 3<sup>rd</sup> party libraries as well as the compiled .jar file of the SIP library from the *cryptdroid\_sip* project. See 3.1.1.

2. **cryptdroid\_base** – Contains the java `com.cryptdroid.base` package with the core logic of the application. The package source files contain no direct references to Android classes or interfaces and form therefore the base module that may be reused in implementations for other platforms that support Java (e.g. a desktop PC version). It also contains the `com.cryptdroid.-android` java package with the Android platform specific implementation of HAL and some of the logic components (controllers). Additional 3<sup>rd</sup> party libraries that are distributed in the form of Java source files and need to be compiled are also packaged in this project.
3. **cryptdroid\_ui** – This is the main project that depends on the previous two and on the Android software development kit (SDK). The Eclipse project has the Android project trait which allows it to contain the individual screens of the graphical interface (Android activities) of the application. The Activities are placed in the `com.cryptdroid.ui` package. The project is also used to build the final binary distributable (Android .apk file) as described in section 3.4.

The three supporting projects are not compiled into the final distributable and host supplementary data:

1. **cryptdroid\_sip** – A Java project containing the refactored source files of the JAIN SIP library (see 3.2.4). An Ant build script is included to build the JAR library file which is then placed into the *cryptdroid\_3rdparty\_lib* project.
2. **cryptdroid\_test** – Test project with the jUnit test source files.
3. **cryptdroid\_doc** – Strictly speaking this is not a project but only a folder with the OpenSIPS configuration files, program documentation (JavaDoc) and a digital copy of the thesis' text.

### 3.1.1 Used libraries

In cases where the application uses well known algorithms and/or approaches it would be inefficient if it did not reuse existing code in form of libraries. Such an area is for example logging or code testing frameworks. On the other hand some libraries (e.g. JAIN SIP and Google protocol buffers) are needed to implement the functionality set by the architecture. Table 3.1 provides a comprehensive overview of all used libraries in the CryptDroid application.

Since the Android framework already contains parts of some of the libraries two libraries had to be renamed (refactored) in order to not cause Java duplicate class name conflicts when they are to be included into the application. The first library is the widely used Bouncy Castle Java security provider library<sup>1</sup> that contains implementations of all cryptographic algorithms used in the application as well as PGP. The Android framework uses parts of this library itself therefore the alternate Spongy Castle<sup>2</sup> had to be included. It is identical to Bouncy

---

<sup>1</sup><http://www.bouncycastle.org>

<sup>2</sup><https://github.com/rtyley/spongycastle>

Library name	Purpose	JAR file(s)	Version
<b>Android annotations</b>	Simplifies Android GUI development with annotations	androidannotations-2.6.jar androidannotations-2.6-api.jar	2.6
<b>Bouncy/Spongy Castle</b>	Cryptographic library with PGP and commonly used ciphers implementation	bcprov-jdk16-1.46.jar scprov-jdk16-146.jar	1.46
<b>Google Guava</b>	General purpose library	guava-11.0.1.jar	11.0.1
<b>Google Protocol Buffers</b>	Implementation of the GPB uniform data format (see 2.2.4)	protobuf-java-2.4.1-lite.jar	2.4.1
<b>SLF4J</b>	Logging framework for Android/desktop version	slf4j-android-1.6.1.jar slf4j-api-1.6.6.jar slf4j-jdk14-1.6.6.jar	1.6.1-RC1 /1.6.6
<b>JAIN-SIP</b>	Refactored JAIN SIP library implementation	jain-sip-sdp-1.2.1111.jar	1.2
<b>jUnit</b>	Java testing framework	junit-4.10.jar	4.1
<b>jlibRTP</b>	RTP transmission library	None, included in cryptdroid_base	0.2.2
<b>org.sipdroid.codecs</b>	Audio codecs package from the SipDroid project	None, included in cryptdroid_base	2.7

**Table 3.1:** Used 3<sup>rd</sup> party libraries

Castle with the exception of the package name (`org.spongycastle` instead of `org.bouncycastle`) and the provider name (“SC“ instead of “BC“).

The second library is the JAIN SIP library<sup>3</sup> which Android includes from version 2.3 upwards (API level 9). Since the application targets API level 7 (Android 2.1) the library has to be included but without causing conflicts on newer Android versions. The *cryptdroid\_sip* project solves this by refactoring a copy of the library checked out from a Subversion (SVN) server of the original JAIN SIP project. It prepends a “com.cryptdroid“ prefix to all package names in the original project and is built into a JAR file that is referenced by the *cryptdroid\_base* project.

---

<sup>3</sup><http://jsip.java.net>

## 3.2 Implementation specifics of components

### 3.2.1 The Events and services system

The events system as explained in subsection 2.2.1 is implemented one to one to the specified design by the application's `com.cryptdroid.base.events` package. The `EventHandler` is a singleton class responsible for receiving and distributing events to registered listeners. Events are created in the `EventFactory`. Static controllers and the event types they want to listen to are registered by the `StartupManager` class which is called by the `BackendService` class on application start-up.

The `BackendService` is an implementation of the Android's `Service` class and thereof must have been placed in the *cryptdroid.ui* project. It represents the heart of the application in the sense that this is main entry point of the CryptDroid application. This service is present for the whole lifetime of the application and basically holds all event listeners from the logic layer. The controller instances could not have been placed into an Android activity since the Android operating system may kill `Activity` classes almost at will (e.g. to save battery life or free up memory) which in turn would remove all references to the controllers from memory. Such an effect is not desirable since the controllers need to be always present to receive events (such as incoming calls) even when the application is not active on the user's screen. This distinction into services and activities is important in Android since activities are only short-lived screens (objects) while services are the only classes guaranteed not to be killed by the OS. Therefore placing the logic layer into a service is only a consequence to the given restrictions of the underlying operating system. Aside from the main effect of the application being able to run in background the service also cleanly separates the presentational layer (the activities) and the logic layer (the controllers) by acting as a middleman between them. The activities can access the service through their own instance of the `PresentationHandler` class. The `PresentationHandler` class is a helper class that automatically connects the activity to the service and registers the activity as a runtime event listener (if case the activity wants to receive some events). On the other hand when the activity is destroyed it de-registers it from the service and the event handler.

The `BackendService` object has one additional responsibility which is to inform the application when the network channels or connectivity changes. By listening to native Android events it detects changes in the network configuration such as when the user switches from a mobile network (e.g. 3G) to a wireless one or connects to a network the first time. This is usually accompanied by a temporary network connection loss and by obtaining a new IP address. These situations are signaled to the application with the `NET_CONNECTION_LOST` or `NET_CONNECTION_ESTABLISHED` events respectively so it can take appropriate measures (such as reloading the SIP stack with the newly acquired IP address). All event types are defined in the `EventTypeEnum` enumeration Java object and summarized in Table A.1 (p.77).

### 3.2.2 Hardware abstraction

Hardware abstraction classes representing underlying devices and services (see 2.2.2) are placed in the `com.cryptdroid.android.hal` package together with a factory (`HALAndroidImpl` class) for creating these objects. Since the `com.cryptdroid.base` package should not reference directly any objects from the Android SDK and the HAL system should be generic for multiple platforms only the HAL interfaces and a proxy HAL factory (`HAL` class) from the `com.cryptdroid.base.hal` package are used in the logic layer. The abstraction was done only for services needed in the core logic. See the program documentation on the HAL package for details on which services these are.

The android specific package in the *cryptdroid.base* project contains not only HAL specific implementations but also a `com.cryptdroid.android.controllers` package for controllers that are specific to the Android platform. It is designed in first place to hold controllers that make sense only on Android. Controllers that require a direct access to the platform's API due to the way how the Android platform handles the application's resources and would be cumbersome to write using only HAL interfaces may also be placed here. For now only the controller handling call ringtones is located here (`RingtonesSingletonController`).

### 3.2.3 Cryptography and CryptoBoxes

The application security scheme heavily relies on the usage of cryptographic algorithms as described in section 2.5. In the logic layer they are used through the standardized `javax.crypto.*` a `java.security.*` Java interface packages. The implementation used is provided by the refactored Bouncy Castle library Java cryptography provider `Spongy Castle` which also provides an OpenPGP (RFC 4880) implementation. All programmatic access to these packages is however wrapped around with custom classes in the `com.cryptdroid.base.crypto` package to prevent misconfiguration and unnecessary boilerplate code. The package main classes are `CryptoBoxSymmetric`, `CryptoBoxFactory` and `CryptoBoxSpec` where `CryptoBox` is an acronym for a cryptographic black box object. This means that once a `CryptoBox` object is configured it can be used as a "black box" to encrypt, decrypt or hash data without the respective code worrying too much about the inside mechanisms of it. The configuration is created and stored in immutable `CryptoBoxSpec` objects which are then passed to the `CryptoBoxFactory`. The factory checks the validity of the configuration and if it was correct returns a fully configured `CryptoBox` object. If not it throws an exception. For now only `CryptoBoxes` for symmetric algorithms are implemented (`CryptoBoxSymmetric`) and its derivations with Digest or HMAC algorithms (`CryptoBoxSymmetricDigest` or `CryptoBoxSymmetricHMAC`).

A configuration is created by providing parameters to the `CryptoBoxSpec` object constructors. All parameters are enumeration objects from the `com.cryptdroid.base.crypto.algorithms` package which contain all supported cryptographic algorithms, padding and block modes, key sizes and hash digest and HMAC algorithms. `CryptoBoxSpec` objects can be also created in the factory by calling the helper methods `getCryptoBoxSpecsFromPreferences()` and `adaptCryptoBoxSpecsFromSipAttachment()`. As their names already give away the first method creates specs based on the user's preferences whereas the latter bases

the specification on the parameters received from a remote user to whom a SIP voice call is being established. The created CryptoBox objects are immutable and provide an easy but flexible way to handle the required cryptographic tasks related to symmetric cryptography and hashing.

Asymmetric cryptography is used only in PGP related tasks such as identity verification of the remote party and transferring session keys (see 2.4.2). Due to their separate nature PGP classes have an own `com.cryptdroid.base.crypto.-pgp` package but the base principle of wrapping the access to the underlying classes stays the same. The PGP class provides methods for encrypting and decrypting data and handles PGP signatures as well.

Another important class is located in the `com.cryptdroid.base.crypto` package and that is the `Primes` class. It holds the Mersenne prime numbers #12 to #19 used in the authentication scheme as requested in section 2.3 and provides methods to retrieve or store a prime to index mapping.

### 3.2.4 The SIP stack and OpenSIPS

The SIP related logic used in the prototype is built in top of the JAIN SIP library. The JAIN SIP reference implementation according to [15] is a public domain product of the Advanced Networking Technologies Division at the National Institute of Standards and Technology (NIST, an agency of the United States Department of Commerce). It is the de-facto standard library for many SIP applications and was in a refactored form (see 3.1.1) chosen as the SIP stack for CryptDroid. A drawback of the library is that it was not possible to develop a properly functioning prototype communicating a SIP session through TCP/TLS sockets. While TLS is officially supported in JAIN SIP the library code execution mostly failed with an exception. Additionally NAT traversal using TURN was not possible when using TLS though it was properly configured for UDP. As this would hinder all incoming communication from the SIP server it was decided to use the less secure UDP sockets for all SIP messaging. The negative security impact of this solutions is thanks to the communicator security scheme limited as discussed in section 2.5.

The SIP stack implementation in the application is formed mainly by three classes that have their home in the `com.cryptdroid.base.sip` package (with the exception of the controller). The stack configuration is held by the `SipCfg` class which includes static properties (such as the transport protocol used) and dynamic properties like the SIP account name and password to register with on the SIP registrar server. The configuration file is passed to a newly created `SipStackListener` object the first time a SIP\_INIT or every time a SIP\_RESET event is received by the `SipSingletonController`. The controller is responsible for managing the SIP stack instance and translating CryptDroid events to SIP events (messages). `SipStackListener` is the actual SIP message processor responsible for handling SIP requests and responses and translating them back into CryptDroid type events. To listen to SIP events it implements the JAIN-SIP `SipListener` interface and to send the application's events it holds a

reference to a `SipStackCallback` interface which is conveniently implemented by `SipSingletonController` that thereby closes the event loop.

Note that the SIP stack is also reloaded each time a `NET_CONNECTION_ESTABLISHED` event is received in order to update the stack and the registration entry on the registrar with the newly acquired IP address.

Because the application cannot be used meaningfully without a SIP account a SSL secured website at <https://www.cryptdroid.net/register/> has been set up to facilitate SIP account registrations. The user is notified about that fact in the initial set-up phase of the application and must configure an account before he may proceed. The registration website is fairly simple and contains only one form for entering the desired username and password. After submitting the form a SIP account is created in the OpenSIPS server's database.

The mentioned domain name aliases a virtual machine hosting not only the website but also the OpenSIPS and rtpproxy applications. OpenSIPS serves as the registrar and proxy according to the architecture requirements as specified in subsection 2.4.3. All configuration files needed to set up the applications may be found in the `cryptdroid_doc/conf` directory on the attached CD-ROM.

### 3.2.5 Calls

Voice calls are the most critical feature of the secure communicator not only from the user's point of view but also regarding the number of processes involved. Once a call is established and signaled to the `CallSingletonController` listener it fires up five processes. They divide the necessary responsibilities belonging to a voice call ranging from audio recording on the sending site to audio playback on the receiving side as they were specified in subsection 2.4.4. Figure 3.2 depicts these processes in a one way audio stream situation from the caller to the callee. Of course in the implementation both ends have to mirror the depicted situation in order to send and receive audio data at the same time therefore running all five processes in parallel.

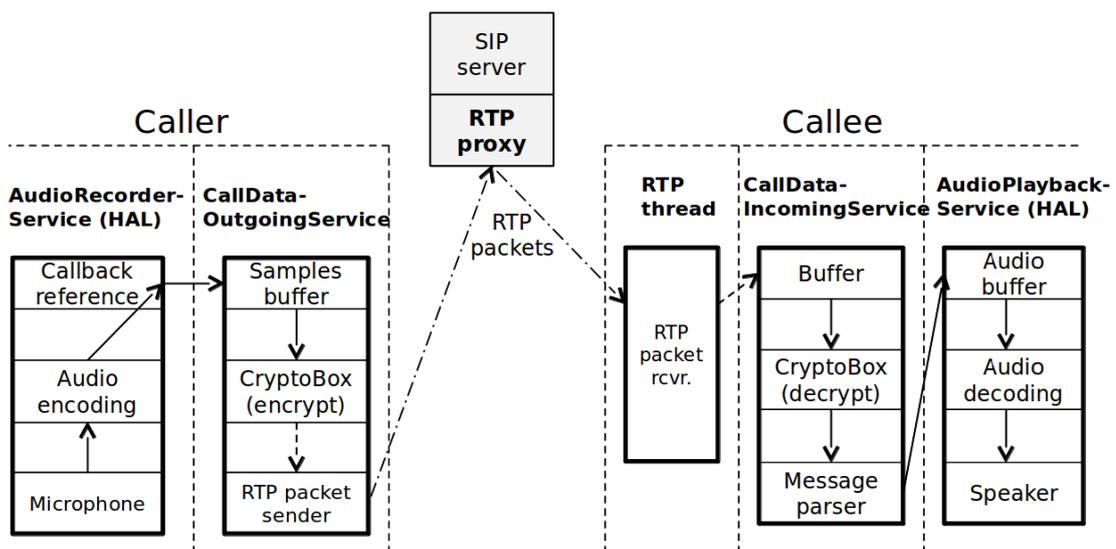


Figure 3.2: Call processes

In the shown situation the call is already established and the RTP proxy has been appropriately configured by the SIP server to forward all RTP packets from the caller to the callee to circumvent symmetric NAT devices. Full arrows represent the packet payload (recorded audio frames wrapped into an encrypted protocol buffer message) whereas dotted arrows represent whole RTP datagram packets. Dotted lines on the other separate processes so each dotted box represents a single thread.

The path of an audio sample carrying the recorded voice begins in the `AudioRecorderService` which is implemented by the `AudioPlaybackServiceAndroidImpl` HAL object. It records raw (WAV) audio samples and encodes them with an audio codec depending on the user's preferences. For performance reasons codec implementations use native (C/C++) code called through a Java native interface (JNI) wrapper for. The codec implementations are borrowed from the SipDroid project<sup>4</sup> and the corresponding wrappers are hence located in the `org.sipdroid.codecs` Java package. Implemented codecs include speex, the G.711 a-law/ $\mu$ -law variants, G.722 and the standard GSM codec as it is used in voice calls over the classic GSM network.

After a sample has been encoded it is sent to a callback reference of the `AudioRecorderReadCallback` interface type. The callback is implemented by the `CallDataOutgoingService` which queues the recorded frames into a buffer (blocking queue). The service is a separate thread that takes multiple frames from the buffer, wraps them into the `RtpPacketPayload` message which it in turn encrypts in a pre-configured `CryptoBox` based again on the user's preferences (see Table B.1, p.78). The encrypted bytes form the payload of a new RTP packet which is then send over the network through the RTP proxy to the callee. This producer-consumer scheme continues until all processes are tore down by the `CallSingletonController` after a `CALL_END` or `CALL_ERROR` event is received.

On the receiving side the sending process is mirrored. For receiving (but also sending) RTP packets the `jlibrtp` library is used (`org.jlibrtp` package) which spawns its own receiver thread. A consumer for that receiving thread is implemented by the `CallDataIncomingService`. Analogue to the outgoing service it buffers the incoming packets and a thread polls them from the buffer. Once the packet payload is decrypted it is split back into audio frames (still encoded). The frames are passed to the `AudioPlaybackService`'s additional audio buffer to prevent a stuttering sound. Once the playback thread registers frames in the buffer it decodes them back to pulse code modulated (PCM) samples and sends them to the Android speaker hardware API.

Note that all classes and interfaces beginning with "Call" are located in the `com.cryptdroid.base.controllers.call` package. This package includes besides the formerly mentioned services and controllers another two controllers. One is responsible for keeping track of the call history (`CallHistorySingletonController`) by listening to all call related events and creating and saving the according database entries as described in subsection 2.2.4. The other one is the

---

<sup>4</sup><https://code.google.com/p/sipdroid/>

**CallRecordingSingletonController.** When activated it taps into the incoming and outgoing data services from where it obtains copies (clones) of the RTP packet payloads before they have been decrypted. The signal is an event triggered by pressing the call recording button on the call screen. These packet copies are stored on the external storage while the metadata (session keys etc.) is kept in the database for playback purposes. The according protocol buffer messages are stored in the *cryptdroid\_base/src-proto* folder.

### 3.2.6 Database

The database according to the scheme in Figure 2.3 is modelled by multiple hash maps, lists and protocol buffer objects kept in the **UserDB** object. Once the database is loaded it resides only in memory and is written to disk only for persistence. Because it would have no sense at this time to have multiple databases the **UserDB** object is a singleton where all its public methods are synchronized. The synchronized method access ensures data integrity and atomicity of database operations. Before the public methods may be called the database has to be loaded (or created) otherwise they throw a **DatabaseNotOpenException** exception. For a full database API specification please refer to the JavaDoc program documentation. Since the database is part of the authentication scheme it is held in the `com.cryptdroid.base.controllers.security` package. Methods for creating, opening and saving the database are package private and can be called only from the **UserAuthenticationSingletonController** which resides in the same package and listens to authentication and database events. These events include the `SECURITY_USER_FIRST_SETUP` event for the initial creation of the database file, the password and a random prime-index mapping. The event is sent only once in the whole application lifetime and that is after the initial application setup. Further events are `SECURITY_USER_AUTH` which are sent from the login screen each time the user has entered his password (prime-index combination) and wants to log in. In both cases the response events are either `SECURITY_USER_AUTH_OK` or `SECURITY_USER_AUTH_FAILED`, depending on whether the database loading/creation was successful or not. Once created the database is loaded on succeeding runs by a `SECURITY_USER_AUTH` event with the entered credentials as the parameter.

The database is persisted to the internal storage on two occasions. The first is after receiving a `SYSTEM_EXIT` event which not only creates the database structure file and saves it to disk but also clears all database related variables and references. The second occasion is the reception of a `SYSTEM_PERSIST_DATA` event which saves the data to disk but does not unload the database from memory. To prevent data loss on application crashes or forced shut-downs by the operating system the event is sent after the user leaves the main screen or a significant data change took place (such as adding a contact). To save battery life it is not sent periodically because encryption is quite demanding on computational power.

Created database files may be exported from the application based on a user request from the GUI. The file is saved on an external storage device so it can be imported later after a data application wipe or on another device.

### 3.2.7 Graphical user interface (GUI)

The main project *cryptdroid-ui* is an Android type Eclipse project and contains all the activities (screens) and media resources of the application. As explained in subsection 3.2.1 the CryptDroid front end activities are connected through a `PresentationHandler` proxy object to the back end part. The back end (*cryptdroid-base* project) handles the application's logic and is stored in the `BackendService` object. When the service is running an icon and a persistent notification is shown in the Android status bar.

The screens itself are written directly to the Android platform API in order to fully utilize the framework's capabilities. In addition the code makes heavy use of the `androidannotations` library that removes a noticeable amount of boilerplate code. All activities and services are placed in the `com.cryptdroid.ui` package. A short description of technical features of each activity follows (see also Figure 3.3). For a short user guide please refer to Appendix C on page 79.

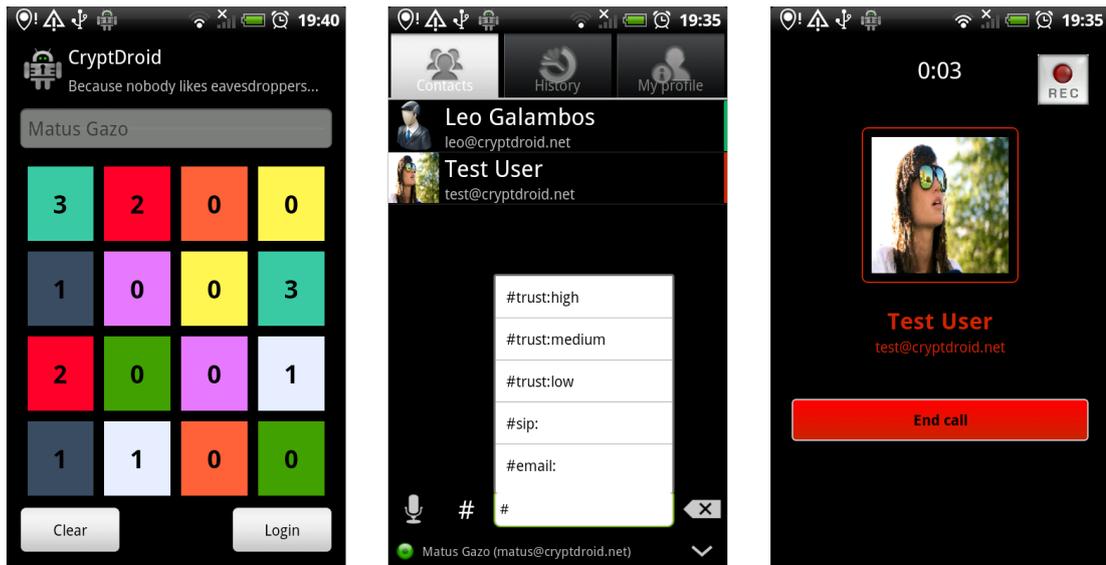
**SetupActivity** is the first screen the user sees after a fresh installation of CryptDroid. It contains seven sub-screens with instructions on how to properly set up the application including the SIP account, the PGP keyring and the color based password. After every field is correctly filled it sends the values as a parameter of the `SECURITY_USER_FIRST_SETUP` event to initialize the database. If the initialization succeeds the user is redirected to the home screen activity.

The other option to initialize a database is to import it from a previously exported database file. In this case a separate **DbImportActivity** is launched which guides the user through the process. Because the IMEI number of the device the original database was created is part of the salt of the database password the user must know and enter it.

**LoginActivity** is the first screen the user sees on each consecutive login. It serves only for entering the password and username. The username field is pre-filled and can be modified only if the username could not be read from application local preferences. The color buttons are randomly placed in two rows and their index value is read from the stored prime index mapping. After the credentials have been entered a `SECURITY_USER_AUTH` event is sent. On success the database is loaded, the client registers itself at the SIP server and redirects the user to the home screen (**HomeActivity**).

**HomeActivity** is the main screen of the application. From here the all functions of the communicator can be reached. It is divided into three tabs where the first contains a list of contacts, the second a call history and the final tab with the (editable) details of the currently logged in user. On the first two screens a name filter is available that filters the lists for a given name (or starting part of a name). When a hash sign (“#”) is entered into the filtering box an auto-complete menu shows available “tags“ (filters) to filter out entries based on various criteria e.g. the trustworthiness of a contact or the date when a call was made.

Pressing the menu button gives more options such as sharing the user's contact information (together with the public PGP key ring), getting to the preference



**Figure 3.3:** Application screens (From left: The login, home and in-call activities)

screen or exiting the application.

When selecting a contact or a call history item in the first or second screen a pop-up menu is shown with available actions. This serves mainly for establishing a voice call or short-cuts for emailing or sending a SMS. Selecting the call button sends triggers a `CALL_START` event which effectively results into sending a SIP invite request to the selected remote party.

**ContactActivity** is launched after a contact is selected for editing or adding. From there a new contact entity can either be created or imported. The implemented contact imports and exports include a file based variant where the contact file (.cdc) is stored on the device's storage or sent directly by email. A second variant uses QR codes that can be scanned by a 3<sup>rd</sup> party software named Barcode Scanner. The scan result must be a Base 64 encoded prototype buffer **Contact** entity (Listing 2.4).

PGP key rings may be also updated here on existing contacts to address key revocation or expiration issues.

**CallActivity** is a simple activity presented to the user while in-call and provides controls for accepting, ending or recording a call in progress.

**PreferencesActivity** is a standard Android layout preference screen that allows the user to set his preferences including SIP account credentials or audio and voice data encryption settings. Since Android provides an own preferences framework the shared and local type preferences as defined in Table B.1, (p.78) are copied to the Android implementation (**SharedPreferences** class). After editing the preference changes are persisted back to the database. The HAL interface **AppLocalPreferences** is responsible just for that.

The **DbExportActivity** may be started from this screen in order to export the current database in memory to an external file. The activity shows the user

the IMEI number of the device used as the database password salt so the user can import it on a different device with a different IMEI number.

### 3.3 Implementation specific security issues

This chapter is an extension of the original security analysis (2.5) taking into account specifics of the implementation and the Android OS. It will not deal with the concepts of Android security per-se as these are sufficiently handled in available publications ([28], [29]).

In general the greatest risk on Android systems is that applications may exploit vulnerabilities to gain root access (privilege escalation)<sup>5</sup>. Eventually some user's root their phones on purpose making these attacks even easier. A malicious application can then break out of its sandbox and access memory segments, preferences and files belonging to other applications (such as CryptDroid). The impact of this is fatal and was discussed in the original security analysis. Should it however have access only to the database file which is normally stored in the private application segment of the internal storage the attack is void as the file is properly encrypted and signed.

Access to the user's preferences (that are cloned to the Android preferences system only at runtime though) has a limited impact as the preferences (as listed in Table B.1, p.78) do not contain fatally sensitive information except the SIP account credentials. It is imaginable that the attacker changes the preferences (and the user might not even notice) to open a window for a different kind of attack e.g. by selecting a digest algorithm over a HMAC one for data integrity checks or a weaker encryption algorithm.

The biggest drawback of the implementation however is that for various reasons (see 3.2.4) TLS could not be used in the signalling plane with all the described negative effects this concludes. This does not necessarily mean the application is insecure as (voice) data confidentiality is untouched by this. A critical part of the whole solution, the secure random number generator (RNG) is taken from the Android implementation of `SecureRandom`. It employs the SHA1PRNG algorithm and is seeded from `/dev/urandom`. According to the documentation “[the] seed is unpredictable and appropriate for secure use”<sup>6</sup>.

### 3.4 Building and testing the application

While this thesis comes with a pre-packaged binary build of the prototype application stored on the attached CD-ROM (see Appendix D, p.83) it is possible to build CryptDroid from source. The preferred way is to import all projects into Eclipse and use the IDE to compile and export the application's APK file. An Ant *build.xml* script is included for building, testing and generating JavaDoc

---

<sup>5</sup>To test whether a device is vulnerable to root exploits the X-Ray tool can be used.  
<http://www.xray.io>

<sup>6</sup><http://d.android.com/reference/java/security/SecureRandom.html>

documentation from the command line. For that at least Ant version 1.8.2 is required (obtainable from <http://ant.apache.org>).

Prerequisite for both ways is the installation of the newest (revision 20.0.1) Android SDK found at <http://developer.android.com/sdk/>. Please follow the instructions on the website on how to install it. The installation directory of choice will be referred to as *\$INST\_DIR*. Once set up the SDK development packages must be downloaded separately by running the SDK manager application<sup>7</sup>. At least the “Android SDK Platform-tools“ and API level 16 (Android 4.1, the target level) packages must be selected and installed before proceeding. Further the binaries located in the *\$INST\_DIR/tools* and *\$INST\_DIR/platform-tools* folders must be in the respective PATH variables of the given operating system.

After setting up the required tools the build process is fairly straightforward. Copy all *cryptdroid\_\** folders from the attached CD-ROM to a directory on the system’s harddrive and enter *cryptdroid\_ui* (the main project) from the command line. Before running the Ant script modify the *local.properties* file’s *sdk.dir* value to point to the SDK installation directory (*\$INST\_DIR*). The script is now ready to be run and by typing ‘*ant -p*’ or simply ‘*ant*’ in a shell (still in the *cryptdroid\_ui* directory) all available ant tasks will be presented. Main tasks include:

- **test** – Runs the jUnit tests of the *cryptdroid\_test* project.
- **doc** – Generates JavaDoc into the *cryptdroid\_doc/javadoc* directory.
- **debug** – Compiles and builds a debug version of the APK installation file into the *cryptdroid\_ui/bin* directory.
- **release** – Compiles and builds a (unsigned) release version.
- **installd** or **installr** – Installs the built APK file onto a connected Android mobile device or a running emulator. On hardware devices the software debugging mode must be enabled for this.

Please refer to the short user manual in Appendix C (p.79) for instructions on how to set up a user account and work with the running application.

---

<sup>7</sup><http://developer.android.com/sdk/installing/adding-packages.html>

## 3.5 Prototype evaluation

The secure communicator architecture was successfully ported in form of a prototype application on the Google Android platform. It is capable of establishing secured and authenticated VoIP/SIP calls through all types of NAT devices and set-ups. In addition it has been tested under real-world conditions on multiple devices which are:

- HTC Desire – 576MB RAM, 1GHz ARM Scorpion CPU, Android 2.2,
- LG P500 – 512 MBRAM, 600MHz ARM 11 CPU, Android 2.3,
- Sony Xperia Mini – 512MB RAM, 1GHz ARM Scorpion CPU, Android 2.3.

To display the in-call performance of the application under real-world conditions in the thesis text a benchmark was performed using the first two devices in the list. By recording the calls with the built-in recording functionality enough data could be collected to measure the packet jitter and the time to process a single RTP packet (with a payload as defined in 2.4.4). Figure 3.4 shows the results of this benchmark. The jitter values in the table indicate the difference in time between arrivals of two related RTP packets. Negative jitter values were not considered since they do not induce lags and the average value would be zero. The processing values for outgoing packets (“OUT“) are measured from the point of where the process starts to create an RTP packet until the moment it is sent. It thereby represents the time needed to pack and encrypt the recorded audio frames and – if selected – to sign them. Values for processing incoming (“IN“) packets measure the reversed process. However this process is, by design, slower as packets are buffered on arrival and the measurement ends only after the packet’s raw audio samples have been submitted to the underlying audio playback device.

For every measurement type the minimum, average and maximum value is always obtained over an approximately 120 seconds interval (or about 400 RTP packets in each direction, the call duration) using a wireless network. The wireless network router itself was attached to a VDSL2 (16Mbit down/1Mbit up) line. Files related to the benchmark are stored on the attached CD-ROM in the *cryptdroid\_test/testFS* directory.

An interpretation of the results is that by considering the most important jitter value on average the application performs well. The standard jitter value is usually indicated at 30 ms for VoIP calls without distortions and artefacts. The application performs on the given network with an average jitter of about 15 ms. However due to network congestion and unstable latency times the jitter may on occasion reach up to an average of 110 ms, individually even over 150 ms. If the incoming packet buffer is not refilled in time this might be noticeable to the user as silence or voice stutter. Also noticeable is that while using the more computationally demanding HMACs for data verification does in general affect the average jitter value (to the worse), the impact in absolute numbers is relatively low as shown by the Processing OUT average values.



for public key exchange and since the defined data format is versatile an additional way to exchange the keys using QR codes was implemented.

Remaining considerations in this evaluation are that while the architecture proposes other types of data transfers, due to the limitations of available hardware the implementation legitimately focuses on voice calls only which on their self already utilize the full computing power available on the tested phones. This was a known limitation also pointed out in the analysis (2.7).

The most pressing issue is therefore the absence of the secured (TLS) transfer of SIP signalling messages which should be definitely addressed in the future according to the requirements of the analysis even though the negative security fallout is limited as analysed in section 2.5.

Also, while call quality was not the primary issue of the thesis nor the prototype implementation, the sensitive microphones produce an acoustic feedback loop which may impact the audio quality negatively. This also concludes the evaluation as of except the issues mentioned in the last paragraphs the prototype implementation delivers what was required by the architectural design.

# 4. Conclusion

This chapter completes the thesis by a short evaluation on the fulfilment of goals as they were defined in section 1.1 and gives an outlook in possible future enhancements.

In compliance with the first goal the thesis proposes in chapter 2 a secure architecture and analyses the available channels and means to do so. The architecture is well described and the event-based component system used has specified interfaces described by events or platform-independent messages. A security analysis was conducted in section 2.5 and thereof in the eyes of the author the analytical part of the thesis has been fulfilled. This applies also to the implementation related (second) goal as the final application is properly implemented to conduct secure VoIP calls. The prototype is ready to be installed on every Android device with version 2.1 upwards from a package found on the attached CD-ROM (Appendix D, p.83).

The last goal, the evaluation of the prototype, was done in an examination in section 3.5 with the result that despite some technical issues (e.g. the lack of use of TLS sockets in the signalling plane) it successfully followed the pattern set by the architecture and has proven its viability in real-world tests. The limitations of the prototype as well as of the architecture were properly analysed and listed.

To summarize, the thesis tried to tackle the problem of securing data and media transmissions by providing an architectural proposition of a secure communicator that can be implemented not only on current mobile platforms but in general as well. By that it (and the application) differs from most currently implemented security solutions for mobile devices which focus on one device or platform only. Compared to them, as also outlined in sections 2.6 and 2.8, it uses a greater variety of cryptographic algorithms and provides authentication mechanisms based on OpenPGP signatures.

## 4.1 Summary and future work

All in all it was established that all main goals of the thesis were fulfilled and that the proposed architecture is applicable for being implemented and deployed on real-world hardware and environment. There, even on mid-range mobile devices, it is capable of delivering acceptable results. And though the application is only a prototype it is already usable for secure VoIP call including authentication verification. Still, there is some work ahead to be done in order the application to reach industrial grade quality.

Foremost the weakest point of the application as of now is that TLS sockets could not be implemented. While this is no way fatal, to complete the security scheme as designated the issue in the JAIN-SIP stack should be investigated. If that should not be possible the whole SIP library part should be exchanged,

which given to the few real alternatives may result in using a different signalling protocol at all.

Further future enhancements may include optimizing the architecture and its security related parts by e.g. an independent third party evaluation (sometimes also referred to as a “hacking contest“) to discover (and fix) new attack vectors. On the implementation side of things a compatibility layer to existing ZRTP/SRTP communicator solutions and commercial VoIP/SIP providers (such as 802.cz) could widen the eventual user base. Also the architecture should be deployed and tested on other platforms.

From the user’s perspective it may be desirable to address the somewhat annoying acoustic feedback issue with an echo cancellation method. Along with the growing computational power of smartphones continuously adding concurrent features such as secured messaging and video calls is thinkable. For some less-experienced users cosmetic changes to the GUI may provide a greater user experience and by potentially fully abstracting the security scheme (especially the OpenPGP part) away from the user may lower the initial barrier to use secure software.

# Bibliography

- [1] WHITTAKER, Zack. *New Wikileaks files expose widespread mobile phone, email hacking capability*. ZDNet [online]. Cambridge, MA: CNET Networks, Inc [cit. 2012-07-28]. <http://www.zdnet.com/blog/london/new-wikileaks-files-expose-widespread-mobile-phone-email-hacking-capability/1218>
- [2] GAMMA, Erich and HELM, Richard and JOHNSON, Ralph and VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995, pp. 144-153. ISBN 0-201-63361-2.
- [3] GOOGLE INC. *Protocol Buffers: What Are Protocol Buffers?*. Protocol Buffers [online]. 2012, 2012-04-02 [cit. 2012-07-28]. <https://developers.google.com/protocol-buffers/>
- [4] ORACLE CORPORATION. *Versioning of Serializable Objects* Java Object Serialization Specification version 6.0 [online]. 2010 [cit. 2012-07-28]. <http://docs.oracle.com/javase/6/docs/platform/serialization/spec/version.html>
- [5] GOOGLE INC. *Developer Guide - Overview - Why not just use XML?*. Protocol Buffers [online]. 2012, 2012-04-02 [cit. 2012-07-28]. <https://developers.google.com/protocol-buffers/docs/overview>
- [6] ZETETIC. *Home - SQLCipher - Open Source Full Database Encryption for SQLite* [online]. 2012 [cit. 2012-07-28]. <http://sqlcipher.net>
- [7] An Introduction to Cryptography. pp. 16-17 [online]. Santa Clara, CA: Network Associates, Inc. and its Affiliated Companies, 1990-1999 [cit. 2012-07-28]. <ftp://ftp.pgpi.org/pub/pgp/6.5/docs/english/IntroToCrypto.pdf>
- [8] Chinese remainder theorem. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2012 [cit. 2012-07-28]. [http://en.wikipedia.org/wiki/Chinese\\_remainder\\_theorem](http://en.wikipedia.org/wiki/Chinese_remainder_theorem)
- [9] CALDWELL, Chris K. *Mersenne Primes: History, Theorems and Lists*. [online]. 1994–2006 [cit. 2012-07-28]. <http://primes.utm.edu/mersenne/>
- [10] XIPH.ORG. *Speex: A free codec for free speech*. [online]. [cit. 2012-07-28]. <http://www.speex.org/comparison/>
- [11] HELLMAN, Martin E. *An Overview of Public Key Cryptography*. IEEE Communications Magazine, May 2002, pp. 42–49. <http://www-ee.stanford.edu/~Ehellman/publications/31.pdf>
- [12] HOOK, David. *Beginning cryptography with Java*. Indianapolis, IN: Wiley Pub., 2005. ISBN 978-0764596339.
- [13] JOHNSTON, Alan B. *SIP: understanding the Session Initiation Protocol*. 2nd ed. Boston: Artech House, c2004, xxiii, 283 p. ISBN 15-805-3655-7.

- [14] PEREA, Rogelio Martínez. *Internet multimedia communication using SIP: a modern approach including java practice*. Amsterdam: Elsevier ; Morgan Kaufmann, 2008, xxiii, 576 p. ISBN 978-0-12-374300-8.
- [15] ROSENBERG, J. and SCHULZRINNE, H. and CAMARILLO, G. and JOHNSTON, A. and PETERSON, J. and SPARKS, R. and HANDLEY, M and SCHOOLER, E. *RFC3261 - SIP: Session Initiation Protocol*. IETF, 2002. <http://www.ietf.org/rfc/rfc3261>
- [16] VOICE SYSTEM. *OpenSIPS*. [online]. 2012-07-11 [cit. 2012-07-28]. <http://www.opensips.org>
- [17] *MjSip*. [online]. 2012-04-22 [cit. 2012-07-28]. <http://www.opensips.org>
- [18] SINGH, Kundan. *P2P-SIP: SIP vs XMPP or SIP and XMPP?*. [online]. 2009-11-09 [cit. 2012-07-28]. <http://p2p-sip.blogspot.cz/2009/11/sip-vs-xmpp-or-sip-and-xmpp.html>
- [19] SCHULZRINNE, H. and CASNER, S. and FREDERICK, R. and JACOBSON, V. *RFC 3550 - RTP: A Transport Protocol for Real-Time Applications*. IETF, 2003. <http://tools.ietf.org/html/rfc3550>
- [20] GONCALVES, Flavio E. *Building telephony systems with OpenSIPS 1.6: build scalable and robust telephony systems using SIP*. Birmingham, U.K: Packt Pub, 2010. ISBN 978-1-849510-74-5.
- [21] OTTERLOO, Sieuwert van. *A security analysis of Pretty Good Privacy*. [online]. 2001-09-07 [cit. 2012-07-28]. <http://www.bluering.nl/pgp/pgp.pdf>
- [22] SETHI, Amit. *Proper use of Java's SecureRandom*. [online]. 2009-08-14 [cit. 2012-07-28]. <http://www.cigital.com/justice-league-blog/2009/08/14/proper-use-of-javas-securerandom/>
- [23] KUHN, Richard D. and WALSH, Thomas J. and FRIES, Steffen. *Security Considerations for Voice Over IP Systems, Recommendations of the National Institute of Standards and Technology*. Gaithersburg, MD: National Institute of Standards and Technology, 2005. Special Publication 800-58. <http://csrc.nist.gov/publications/nistpubs/800-58/SP800-58-final.pdf>
- [24] NORRIS, Steven. *Microsoft and Skype set to allow backdoor eavesdropping*. [online]. 2011-11-07 [cit. 2012-07-28]. <http://memeburn.com/2011/07/microsoft-and-skype-set-to-allow-backdoor-eavesdropping/>
- [25] Comparison of VoIP software. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2012 [cit. 2012-07-28]. [http://en.wikipedia.org/wiki/Comparison\\_of\\_VoIP\\_software](http://en.wikipedia.org/wiki/Comparison_of_VoIP_software)
- [26] ECLIPSE FOUNDATION. *Eclipse - The Eclipse Foundation open source community website*. [online]. 2012 [cit. 2012-07-28]. <http://www.eclipse.org>
- [27] GOOGLE INC. *Developer Tools — Android Developers*. [online]. 2012 [cit. 2012-07-28]. <http://developer.android.com/tools/>

- [28] ENCK, William and ONGTANG, Machigar and MCDANIEL, Patrick. *Understanding Android Security*. IEEE Security and Privacy, vol. 7, no. 1, pp. 50-57, Jan/Feb 2009. doi:10.1109/MSP.2009.26
- [29] SHABTAI, Asaf and FLEDEL, Yuval and KANONOV, Uri and ELOVICI, Yuval and DOLEV, Shlomi and GLEZER, Chanan. *Google Android: A Comprehensive Security Assessment*. IEEE Security and Privacy, vol. 8, no. 2, pp. 35-44, Mar/Apr 2010. doi:10.1109/MSP.2010.2

# List of Illustrations

## Figures

2.1	High level architecture overview . . . . .	9
2.2	UML Diagram of EventHandler and EventFactory . . . . .	11
2.3	Database E-R diagram . . . . .	18
2.4	Database component security scheme . . . . .	20
2.5	Example password calculation . . . . .	21
2.6	Sample QR code with contact/PGP data . . . . .	28
2.7	SIP stack state diagram according to SIP events . . . . .	30
2.8	Application events according to the SIP stack state . . . . .	32
2.9	Session key exchange message layers . . . . .	34
3.1	CryptDroid project logo . . . . .	49
3.2	Call processes . . . . .	55
3.3	Application screens . . . . .	59
3.4	Application benchmark . . . . .	63
C.1	Application installation screens . . . . .	79
C.2	Application set-up screens . . . . .	80
C.3	Main screen tabs with sample data . . . . .	81
C.4	Contact quick options and in-call screen . . . . .	82
C.5	Log-in screen . . . . .	82

## Tables

2.1	Comparison of data formats . . . . .	16
2.2	Comparison of data transfer channels . . . . .	25
3.1	Used 3 <sup>rd</sup> party libraries . . . . .	51
A.1	Application event types . . . . .	77
B.1	Preference types . . . . .	78

## Listings

1.1	Sample code fragment . . . . .	6
2.1	Event processing pseudo-code . . . . .	11
2.2	Database wrapper in Google protocol buffers code . . . . .	20
2.3	Final password calculating function (Java code) . . . . .	22
2.4	Protocol buffer message for a contact (shortened) . . . . .	27

2.5	SipCryptoSpecsAttachment protocol buffer message . . . . .	33
2.6	RtpPacketPayload protocol buffer message . . . . .	36
2.7	CallLogEntry protocol buffer message . . . . .	40

# List of Abbreviations

AES	Advanced Encryption Standard. A specification for the symmetric encryption of electronic data adopted by the U.S. government and later on also worldwide.
API	Application Programming Interface. Provides the means to take advantage of software features, allowing dissimilar software products to interact upon one another.
CBC	Cipher Block Chaining. A cryptographic block cipher mode that provides confidentiality but not message integrity.
CORBA	The Common Object Request Broker Architecture is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.
CRUD	Create-Read-Update-Delete, the four basic operations on computer data storages.
D-H	Diffie–Hellman key exchange algorithm is the most widely used public key distribution system.
E-R	Entity – Relationship model/diagram is an abstract way to describe a database.
GPB	Google Protocol Buffers are a method of serializing structured data developed by Google Inc.
GSM	Global System for Mobile Communications, originally Groupe Spécial Mobile, is a standard developed by the European Telecommunications Standards Institute (ETSI) to describe technologies for second generation (2G) digital cellular networks.
GUI	A graphical User Interface is a type of user interface that allows users to interact with electronic devices using images rather than text commands.
HMAC	A key-dependent one-way hash function specifically intended for use with MAC (Message Authentication Code). Based upon RFC 2104.
HTTP	HyperText Transfer Protocol is a common protocol used to transfer documents between servers or from a server to a client.
IDE	Integrated development environment, a software application that provides comprehensive facilities to computer programmers for software development.
IDL	Interface description language, any computer language used to describe a software component’s interface.

IMEI	The International Mobile Equipment Identity is a unique number to identify mobile phones.
JNI	The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call native applications and libraries written in other languages such as C or assembly.
MITM	The man-in-the-middle attack in cryptography and computer security is a form of active eavesdropping in which the attacker makes independent connections with the victims and relays messages between them, making them believe that they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker.
NAT	Network Address Translator defined in RFC 1631. A router connecting two networks together; one designated as inside, is addressed with either private or obsolete addresses that need to be converted into legal addresses before packets are forwarded onto the other network (designated as outside).
NDK	Native Development Kit, an Android SDK for developing directly on a computing platform rather than the virtual machine.
NFC	Near Field Communication, a wireless communication technology.
NIST	The National Institute of Standards and Technology. A division of the U.S. Dept. of Commerce that publishes open, interoperability standards called FIPS.
OpenPGP	OpenPGP is a protocol for encrypting data using public key cryptography as defined in RFC 4880.
PGP	Pretty Good Privacy. An application and protocol (RFC 1991) for secure e-mail and file encryption developed by Phil R. Zimmermann.
PIN	Personal Identification Number. A password used to access an automated teller machine or other secured systems.
PKCS	PKCS is a group of public-key cryptography standards devised and published by RSA Security Inc.
PKI	Public Key Infrastructure. A widely available and accessible certificate system for obtaining an entity's public key with some degree of certainty of having the right key and that it has not been revoked.
RDBMS	Relational Database Management System. A system in which data is stored in tables and the relationships among the data are also stored in tables.
RFC	Request for Comment. IETF documents that specify Internet standards. Each RFC has an RFC number by which it is indexed and by which it can be retrieved from <a href="http://www.ietf.org">http://www.ietf.org</a> .

RNG	Random Number Generator. A computational or physical device designed to generate a sequence of numbers or symbols that lack any pattern.
RSA	Short for RSA Data Security, Inc.; or referring to the principals - Ron Rivest, Adi Shamir, and Len Adleman; or referring to the algorithm they invented. The RSA algorithm is used in public key cryptography and is based on the fact that it is easy to multiply two large prime numbers together, but hard to factor them out of the product.
RTP	Real-time Transport Protocol. A standardized packet format for delivering audio and video over the Internet.
SDK	Software Development Kit. Typically a set of software development tools that allows for the creation of applications for a certain software or hardware platform.
SHA-2	Secure Hash Algorithm. A set of cryptographic hash functions (SHA-224, SHA-256, SHA-384, SHA-512) designed by the U.S. National Security Agency (NSA) to replace its predecessor SHA-1.
SIP	Session Initiation Protocol, an IP network protocol typically used for VoIP telephony.
SQL	Structured Query Language. A special-purpose programming language designed for managing data in relational database management systems (RDBMS).
SRTP	The Secure Real-time Transport Protocol defines a profile of RTP intended to provide encryption, message authentication and integrity, and replay protection to the RTP data.
STUN	Session Traversal Utilities for NAT. A standardized set of methods, including a network protocol, used in NAT traversal for applications of real-time voice, video, messaging, and other interactive IP communications
SVN	Apache Subversion, a version control system.
TCP	Transmission Control Protocol, one of the core protocols of the Internet protocol suite.
TLS	Transport Layer Security. A secure network protocol and successor to Secure Sockets Layer (SSL).
TURN	Traversal Using Relays around NAT (TURN) is a protocol that allows for an element behind a network address translator (NAT) or firewall to receive incoming data over TCP or UDP connections.
UAC	User Agent Client, part of the Session Initiation Protocol user agent that is behaving like a client-server client (as opposed to UAS, User Agent Server).

UAS	User Agent Server, the part of a Session Initiation Protocol user agent that is behaving like a client-server server (as opposed to UAC, User Agent Client).
UDP	User Datagram Protocol, a network communications method.
VoIP	Voice over IP. Commonly refers to the communication protocols, technologies, methodologies, and transmission techniques involved in the delivery of voice communications and multimedia sessions over Internet Protocol (IP) networks, such as the Internet.
XML	Extensible Mark-up Language (XML) is a mark-up language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
XMPP	Extensible Messaging and Presence Protocol. An open-standard communications protocol for message-oriented middleware based on XML.
ZRTP	ZRTP is a cryptographic key-agreement protocol to negotiate the keys for encryption between two end points in a VoIP telephony call based on RTP. It uses D-H key exchange and the SRTP for encryption.

# Appendices

# A. Event Types

Event type	Argument (class)	Event response type	Action description
SYSTEM_EXIT			Application shut-down. No more events are processed.
SYSTEM_PERSIST_DATA			Save the database (but do not close).
SECURITY_USER_FIRST_SETUP	UserDBBuilder		Initialize the database for the first time
SECURITY_USER_AUTH	UserAuthData	SECURITY_USER_AUTH_OK SECURITY_USER_AUTH_FAILED	Authenticate a user and open database. Respond with OK if correct password, FAILED otherwise.
SECURITY_USER_LOGOUT			Logout the user, close database, but do not exit.
CALL_OUTGOING_START	Contact	CALL_OUTGOING_RINGING CALL_OUTGOING_DENIED CALL_OUTGOING_ESTABLISHED	Initialize a call to selected contact. Response can be that the remote phone is ringing, the call was denied or accepted and call was established (with CallInfo).
CALL_OUTGOING_CANCEL			Cancel our outgoing call attempt.
CALL_END			End current call or call attempt.
CALL_INCOMING_ACCEPT	CallInfo	CALL_INCOMING_ACCEPTED CALL_INCOMING_DENIED	Request the user to accept an incoming call. Response can be ACCEPTED or DENIED (with error code).
CALL_INCOMING_CANCELED			Incoming call was cancelled by remote party before it could have been accepted. Indicate missed call.
CALL_INCOMING_ESTABLISHED	CallInfo		Incoming call accepted by user and ACKed by remote party. Media transmissions should start here.
CALL_ERROR	[SIP response code]		Indicate call failure. End any call or call attempts.
CALL_HISTORY_MODIFIED			Indicate that call history has changed.
NET_CONNECTION_ESTABLISHED			Indicate that the underlying network has changed.
SIP_INIT or SIP_RESET		SIP_INIT_OK SIP_INIT_FAILED	Initialize the SIP stack and (re-)register. IF RESET, force stack unload and initialize. Respond with OK or FAILED.
SIP_DEREGISTER		SIP_REGISTRATION_OK SIP_REGISTRATION_FAILED	De-register from SIP server. Respond with OK or FAILED.
SIP_REGISTRATION_REFRESH_FAILED			Error, SIP registration session could not be prolonged.
CALL_CURRENT_RECORD_START	CallInfo	CALL_CURRENT_RECORD_STARTED_OK, CALL_CURRENT_RECORD_STARTED_ERROR	Start recording current call with given parameters. Respond with STARTED_OK or STARTED_ERROR.
CALL_CURRENT_RECORD_STOP		CALL_CURRENT_RECORD_STOPPED_OK CALL_CURRENT_RECORD_STOPPED_ERROR	Stop recording current call. Respond with STOPPED_OK or STOPPED_ERROR.
CALL_RECORDED_PLAYBACK_START	timestamp (long)	CALL_RECORDED_PLAYBACK_FINISHED	Start playback of recorded call with the given timestamp. On error send PLAYBACK_FINISHED.
CALL_RECORDED_PLAYBACK_PAUSE			Pause or resume the playback of a recorded call.
CALL_RECORDED_PLAYBACK_STOP		CALL_RECORDED_PLAYBACK_FINISHED	Stop the playback of a recorded call and free resources.

Table A.1: Application event types

## B. User Preference Types

Preference type	Default value	Type	Meaning
prefs_sip_proxy	cryptdroid.net:5060	String	SIP proxy server address
prefs_sip_account		String	SIP account user ID
prefs_sip_account_name		String	SIP account username
prefs_sip_account_pwd		String	SIP account password
prefs_sip_stack_port_auto	TRUE	Boolean	Auto-select port of the local SIP stack
prefs_sip_stack_port	5060	Integer	If the former property is false, use this port.
prefs_audio_micgain	TRUE	Boolean	Audio recording volume gain on/off
prefs_audio_preferred_codec	speex	String	Codec for media transmissions
prefs_crypto_call_alg	AES	String	Symmetric encryption algorithm used to encrypt media transmission
prefs_crypto_call_alg_block_mode	CBC	String	Algorithm block mode
prefs_crypto_call_alg_pad_mode	PKCS7-PADDING	String	Algorithm padding mode
prefs_crypto_call_alg_key_size	128	String	Algorithm key size (bits)
prefs_crypto_call_data_verification_method	HMAC	String	Call media data verification method. Can be 'Digest', 'HMAC', or 'None'.
prefs_crypto_call_digest_alg	MD4	String	If 'Digest' selected use this algorithm.
prefs_crypto_call_hmac_alg	HMAC/SHA256	String	If 'HMAC' selected use this algorithm.
prefs_last_used_username		String	Last username used to login the application (Local only)
prefs_crypto_prime_mapping		String	Prime-to-index mapping for database login (Local only)
prefs_db_was_imported	FALSE	Boolean	Set to true if DB was imported, after first valid login deleted (Local only)

**Table B.1:** Preference types

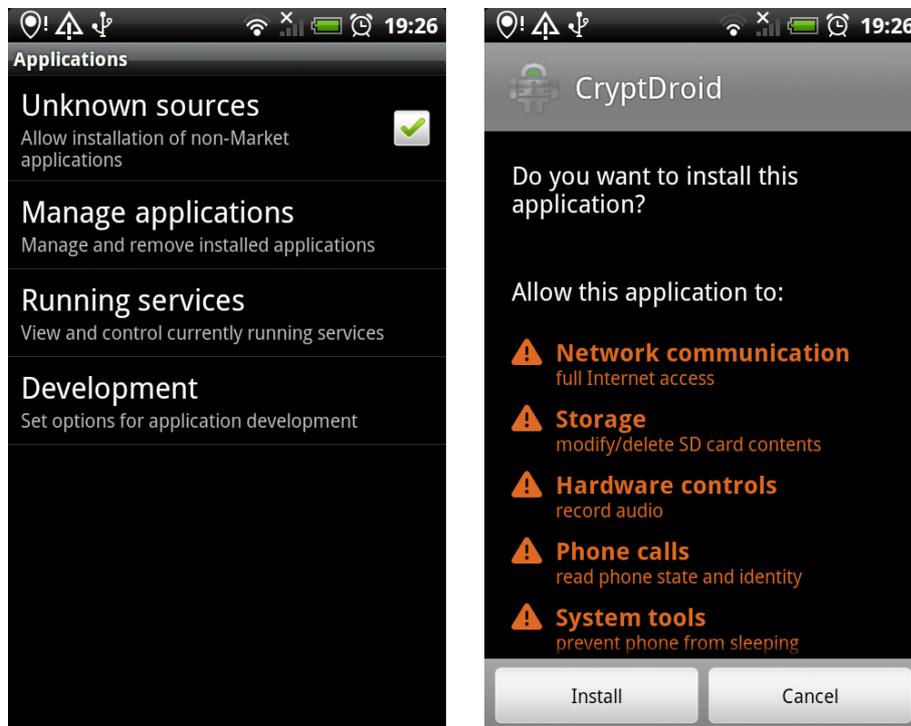
# C. User Manual

This chapter explains how to install the application and the most important screens of it. Before installing the application note that additional software such as a file manager (e.g. ES File Explorer) and a QR code scanner (Barcode Scanner application) obtainable from the Google Play store are recommended to take full advantage of the application’s capabilities.

To install the application its installation file (cryptdroid\_ui.apk) must be copied to the phone first. The simplest way is to download it in the phone’s browser by entering the address `https://cryptdroid.net/cryptdroid_ui.apk` which should save the .apk file to the phone. Alternatively the file can be copied from the attached CD-ROM (see Appendix D) to the phone’s external storage by transferring it over an attached USB cable or over a network service like FTP. Independently of the way before starting the installation the “Unknown sources“ preference must be checked in Android system preferences as shown in Figure C.1.

When installing from compiled sources as described in section 3.4 the “USB Debugging“ option in the “Development“ section must be ticked also.

Once set the installation is done easily by opening the folder containing the downloaded .apk file (usually SD-card/downloads) and selecting it. The installation process is self explanatory as shown on the second screen of Figure C.1.



**Figure C.1:** Application installation screens

After running the application for the first time it will show a series of set-up screens (Figure C.2). Please follow the on-screen instructions carefully to properly set up a CryptDroid account. On the (third) SIP account screen follow the link to

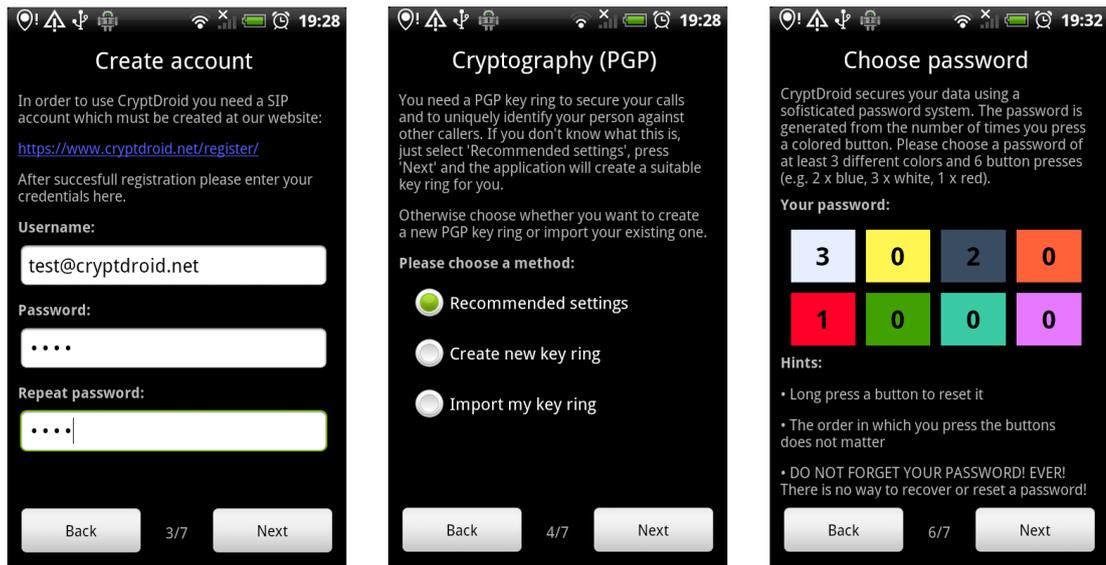


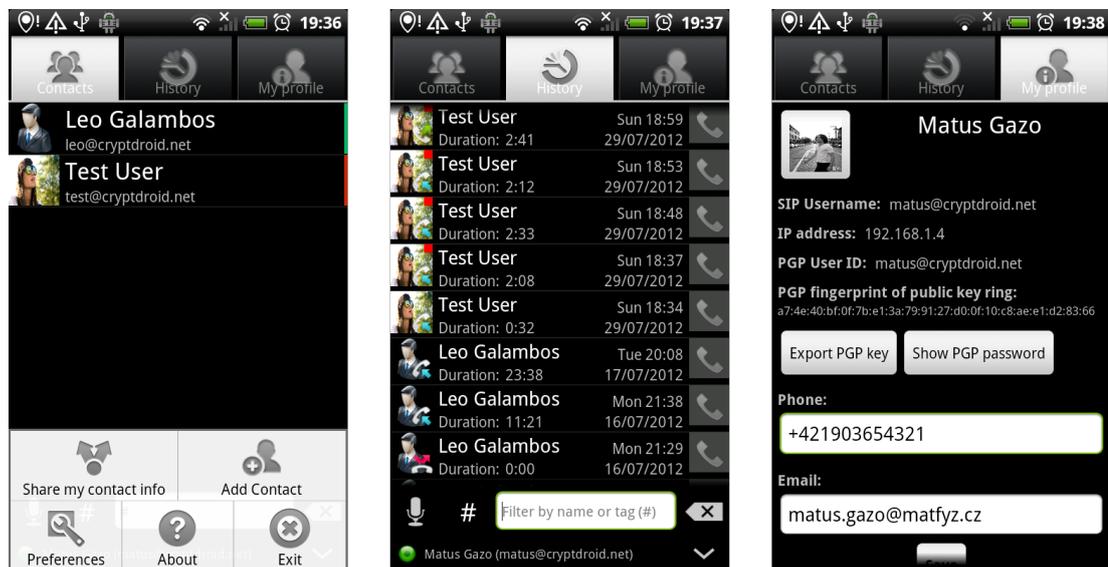
Figure C.2: Application set-up screens

<https://www.cryptdroid.net/register/> to open a browser window and create a SIP account on the CryptDroid server application. A more comfortable way if available is to open the registration form on a PC browser and perform the operation there. The credentials entered in the registration process need to be copied to the set-up screen. The last but one screen is designed for selecting a password of choice which must consist of at least three colors and six clicks in total. The selected password must be re-entered on the screen (note that the colors are positioned randomly here). By pressing the menu button in one of the setup screens a menu will show from where the process can be stopped at any time or an existing CryptDroid database file may be imported.

Should the the set-up finish correctly the main screen of the application should appear. It is divided into three tabs (Figure C.3). The first tab contains a list of contacts, the second tab holds information about made or missed calls (call history) and the last tab shows (editable) information about the CryptDroid account in use. The bar at the bottom of the screen displays the SIP account in use and a filter for filtering the contact and call history lists. The filter takes as input a username or a tag (starting with the hash “#“ sign). The icon next to the SIP username signaled whether the SIP account registration was successful (green icon) or not (red icon). If the icon is red please check the device’s internet connection and SIP account settings in the preferences screen.

To start using CryptDroid contact information (including the public PGP key rings) must be exchanged. By pressing the menu button from the main screen a menu will pop-up containing choices for sharing the own contact information, adding a contact, going to the preferences screen and exiting the application. Contact information can be exchanged through files with the .cdc suffix or QR codes. To handle QR code displaying and scanning the Barcode Scanner application must be installed on the phone. If it is not, CryptDroid will ask to install it. For starters try to scan the QR code in Figure 2.6 on page 28 by selecting

“Add contact“ from the menu and pressing the “From QR“ button. The QR code contains the author’s CryptDroid sample contact badge.



**Figure C.3:** Main screen tabs with sample data

Having imported a contact badge it will show in the contacts tab of the main screen. By clicking on it a quick action menu pops up from where a call can be placed by selecting the “Call“ option (Figure C.4). Long clicking a contact shows a context menu for deleting or editing the contact. While in-call it is possible to record it by selecting the “REC“ button.

It is possible to minimize the application by pressing the device’s home button. This does not exit the application and it is still possible to receive calls even while the application’s screen is not shown. The CryptDroid icon in the system status bar indicates a running application in the background. To exit it the appropriate menu option in the main screen must be selected. Should the application be restarted the login screen is presented (Figure C.5). After the password chosen at set-up time is entered correctly the main application screen will be shown again.

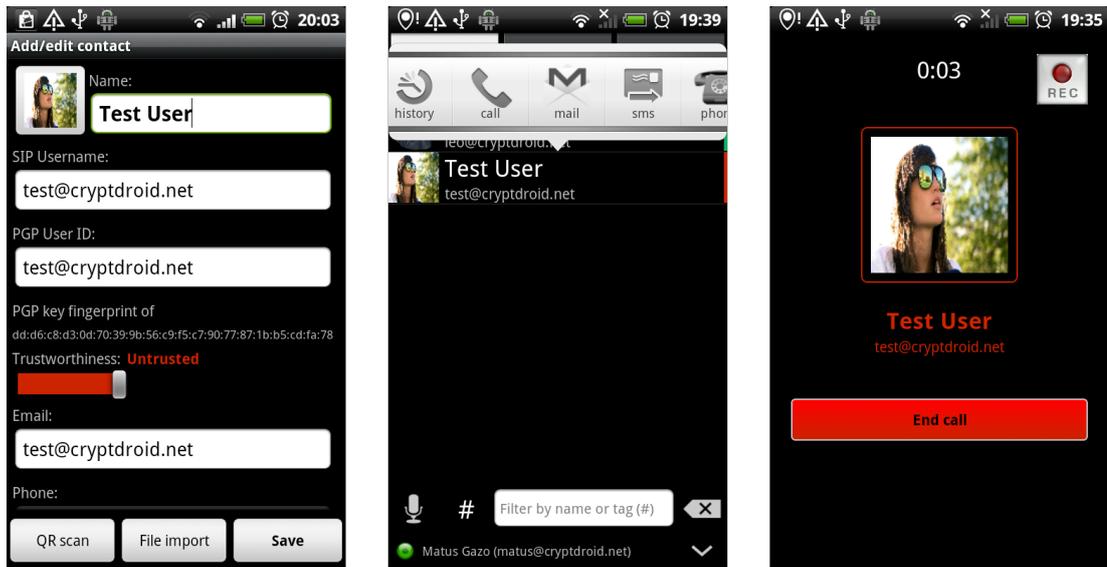


Figure C.4: Contact quick options and in-call screen



Figure C.5: Log-in screen

## D. CD-ROM Contents

Appended to this thesis is a CD-ROM containing the sources of the secure communicator prototype application (CryptDroid). Files on the media include six directories and three files:

- *cryptdroid\_\** folders – Contain source code files of the application as described in section 3.1.
- *cryptdroid\_ui.apk* – The final binary application distributable that can be installed on Android devices.
- *thesis.pdf* – A digital copy of this thesis text in PDF format.
- *README.txt* – File with the text of this appendix.

Program documentation is attached in form of Javadoc HTML pages. It can be viewed by opening the *cryptdroid\_doc/javadoc/index.html* file.

For building the application from source please refer to section 3.4. The Ant *build.xml* file is located in the *cryptdroid\_ui* directory.

Google protocol buffer message sources (*.proto* files, partly shown in listings) are stored in the *cryptdroid\_base/src-proto/* directory.

Configuration files for the OpenSIPS and rtpproxy software (see section 2.4.3) are in *cryptdroid\_doc/conf/*.