6CCS3PRJ Final Year
Decentralised End-to-End Encrypted Messaging App

Final Year Dissertation

Author: Habib Rehman

Supervisor: Costas Iliopoulos

April 23, 2020

Abstract

In an increasingly technologically-interconnected world, messaging apps are more ubiquitous than ever. A number of them support them now support End-to-End Encryption (E2EE) using the Signal Protocol to allow billions of people to communicate privately and securely. However, the protocol has serious security vulnerabilities. While it attempts to provide some protection against those vulnerabilities through in-person verifications for example, they come at such an expense of usability that this protection is practically rendered as futile.

This project presents the design, implementation and analysis of a decentralised end-to-end encrypted messaging app that addresses those vulnerabilities to enable people to communicate with full security and privacy without any sacrifice in usability.

Contents

Chapter 1

Introduction

In an ever-connected world, messaging apps are more ubiquitous than ever. Due to increasing privacy and security concerns as well as technological advancements [11], a number of them support them now support end-to-end encryption (E2EE) to ensure that only the original sender and intended recipient can read messages sent in a conversation. The messages can contain any arbitrary data such as images, videos and files available exclusively to the two parties in conversation. While there are many E2EE messaging apps, the most used ones are WhatsApp (by Facebook) with 1.6 billion monthly active users and Facebook Messenger with 1.3 billion monthly active users [3], and embody the widely used design of E2EE.

All of the existing E2EE messaging solutions are centralised which means all data is stored and transmitted via a central server. Due to that, in order to use the messaging service, consumers are necessitated to give all their personal data, agree to their terms of its use and trust them to act ethically with it. As history has shown, however, they trust is inevitably abused for profit in such with a system as evident by the recent Facebook–Cambridge Analytica data scandal where Facebook sold the personal data and private messages of millions of its users to Cambridge Analytica without their consent which used it for political advertising purposes influencing major international elections [7, 8, 9] and by the numerous data-sharing deals which Facebook has profited from [1].

As we shall see, existing E2EE implementations have serious security vulnerabilities that when exploited can compromise the consumer's security and privacy by allowing governments, corporations, hackers and other bad actors to eavesdrop on their private conversations. We shall see how their security can practically be rendered as futile through the continues physical effort it demands from the consumer to be effective.

1.1 Aim and objectives

We aim to:

> ***Develop an end-to-end encryption messaging app that enables people to communicate with full security and privacy without any sacrifice in usability***

This highlights our primary objectives:

- **Security**: uses the state-of-the-art security.
- **Privacy**: guards the user's privacy requiring the least amount of personal data possible.
- **Usability**: is easy to use; security or privacy do not come at the expense of usability.

1.2 Scope

Our sole focus is private E2EE messaging that occurs between two parties. This excludes group E2EE messaging which is an entirely different problem.

Our scope of E2EE is also limited to messages containing text and files. This means we will not consider E2EE in voice calls or video calls.

1.3 Software Platform and Target OS

The messaging application will target desktop over mobile as the technologies needed such as peer-to-peer are fully supported and much more mature making the application more stable. Moreover, desktop APIs are high-level and streamlined making them easier to use whereas mobile ones (Android and iOS) have great disparities and are very low-level making development very difficult as well as slow.

Chapter 2

Background

2.1 End-to-End Encryption (E2EE)

End-to-end encryption (E2EE) of messages is used ensure that only the original sender and intended recipient can read messages sent in a conversation and no other party including the messaging service. This involves essentially locking (i.e. encrypting) the messages with a set of unique keys (i.e. encryption keys) that only the intended recipient and sender possess to unlock them and read the messages. The messages can contain any arbitrary data such as text, images, videos and files available exclusively to the two parties in conversation.

While there are countless E2EE implementations, they all use the general E2EE design of deriving a different encryption key for each message and authenticating (verification of the source) it using *public-key cryptography*. Public-key cryptography is a method of cryptography where every party generates a *key pair* consisting of a *private key* and a *public key*. As their name implies, the private key is kept a secret like a password while the public key is published openly like a username representing their identity. A party can cryptographically "sign" some information using their private key which can be cryptographically verified by any other party using the party's public key to truly originate from the party providing authentication.

The most secure and *state-of-the-art* E2EE design is the open-source *Signal Protocol* by Open Whisper Systems [5]. It is also the most widely used design of E2EE having been implemented in the two most globally used messaging apps: WhatsApp (by Facebook) with 1.6 billion monthly active users and Facebook Messenger with 1.3 billion monthly active users [3, 14, 16]. Hence, they will be the focus of our analysis of existing solutions.

We will proceed to delineate the Signal Protocol.

2.2 The Signal Protocol

2.2.1 Security properties

The Signal Protocol offers the following security properties: [4]:

- **confidentiality**: only the sender and intended recipient can read the message
- **forward secrecy**: compromise of the current message does not compromise past messages
- **post-compromise security**: compromise of the current message does not compromise future messages
- **identity verification**: confirming that the public key of a user is truly owned by the user.
- **data integrity**: assurance of the accuracy and consistency of the message.
- **data authentication**: assurance of the message's source (i.e. the sender).

The last three security properties (identity verification, data authentication and data integrity) are provided by through the use of public-key cryptography.

2.2.2 Overview

The Signal Protocol [15] is composed of the following main steps:

1. **Registration**
2. **Session setup**
3. **Messaging**

We will proceed to delineate these.

2.2.3 Registration

At installation, the parties who want to securely message each other, both register with the central messaging server using some personal information which is usually their *phone number* in the implementations (WhatsApp, Messenger, ...).

The following set of key pairs are then generated and their public keys are uploaded to the central server:

1. **Identity Key Pair**: A long-term Curve25519 key pair representing the “identity” of the party. Referred to as simply the *Identity Key*.
2. **Signed Pre Key**: A medium-term Curve25519 key pair, signed by the Identity Key, and rotated on a periodic timed basis
3. **One-Time Pre Keys**: A queue of Curve25519 key pairs for one time use, generated at install time, and replenished on demand.

These are generated per-device and kept on the device (only public part is transmitted). This means that they are ephemeral as they only exist for a short time or exist while the app is installed.

All messages sent are cryptographically signed using the Identity Key and all incoming message’s signatures are validated providing authentication in Signal Protocol.

2.2.4 Session setup

The Signal Protocol allows two parties to exchange encrypted messages based on a ephemeral shared secret key, the *initial root key*. This shared secret key is agreed upon by both parties by a *key agreement protocol*.

Implementations use *Extended Triple Diffie-Hellman (X3DH)* where the initiating party requests and receives a set of the other party’s public keys from the central server and uses them to setup a session by derive the initial root key from them. We won’t go into further details of X3DH as it is irrelevant for our project.

2.2.5 Messaging

Once an initial root key is established, the parties use the *Double Ratchet algorithm* to send and receive E2EE messages.

In the Double Ratchet algorithm, every message sent is encrypted using a fresh message encryption key derived from an ephemeral shared secret key which is refreshed periodically. This means earlier message encryption keys cannot be computed from later ones providing a high degree of forward secrecy. The algorithm can visualised by two “ratchets” that are moved forwards by every party with every new message encryption key and new ephemeral shared secret key, respectively. A ratchet is a simple device that only moves forward, one step at a time, much like the algorithm. In the algorithm, the “ratchets” described are really *Key Derivation Function (KDF) chains*.

2.2.6 Key Derivation Function (KDF) chains

A Key Derivation Function (KDF) is defined to be a cryptographic function that takes a secret and random KDF key and arbitrary input data and yields some output. A KDF's output is indistinguishable from random given the key is unknown, and it provides a secure cryptographic hash of its key and input data given the key is known and random.

The output of a KDF can be split into two parts:

- **KDF key**

- **Output key**

A KDF chain can be formed when the KDF key of a KDF output is continuously used as an input to another KDF as shown in the figure below [15]

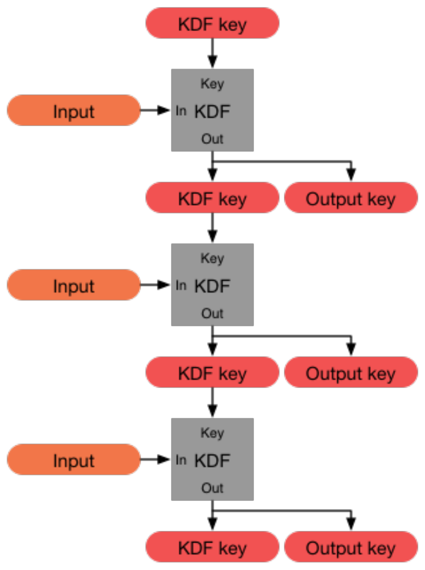


Figure 2.1: A KDF chain

A KDF chain has the following security properties [4]:

- **Resilience**: The output keys are unintelligible to an adversary without the KDF keys even if they compromise the KDF inputs.

- **Forward security**: The output keys from the past are unintelligible to an adversary who compromises the KDF key at some time.

- **Post-compromise security**: Future output keys are unintelligible to an adversary who compromises the KDF key at some time, given that the future inputs have accumulated enough entropy.

The compliant KDF that Signal implementations use is the *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* which utilises the cryptographically-secure Hashed Message Authentication Code (HMAC) algorithm with the SHA256 hash function [17] defined as follows:

```
KDF = HKDF = HMAC-SHA256(input, KDF key)
```

Listing 2.1: KDF definition

2.2.7 Double Ratchet session

A Double Ratchet session between two parties involves each party storing three KDF chains:

- **Root chain**: A common KDF chain between both parties from which their own chains are derived. The KDF key is called **Root key**.
- **Sending chain**: The party's own KDF chain. The KDF key is called **Sending chain key**.
- **Receiving chain**: The recipient party's KDF (sending) chain. The KDF key is called **Receiving chain key**.

As both parties exchange messages and with them their new ephemeral Diffie-Hellman public keys, they generate new shared secret keys that become inputs to the root chain. The output keys that are derived from the root chain become new KDF keys for the sending and receiving chains. One such cycle is referred to as a *Diffie-Hellman ratchet*.

The sending chain is advanced with each new message sent and the receiving chain is advanced with each new message received for each party. The output keys of the sending chain are used to encrypt messages and the output keys of the receiving chain are used to decrypt messages. One such cycle is referred to as a *Symmetric-key ratchet*.

2.2.8 Symmetric-key ratchet

A unique ephemeral message key is used to encrypt every message sent and decrypt one received. The message keys are output keys from the sending and receiving KDF chains. The KDF keys for these chains are referred to as *chain keys* as shown below.

The purpose of the sending and receiving chains is to simply ensure that each message is encrypted with a unique key. Therefore, their KDF inputs are simply constants and do not provide any post-compromise security.

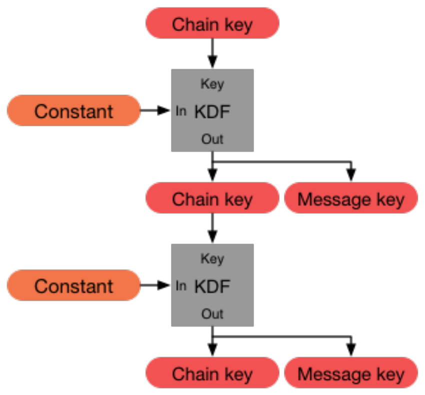


Figure 2.2: Sending and receiving KDF chains

The symmetric key ratchet proceeds as follows:

```
Message Key = HKDF(Chain Key, 0x01)
Chain Key = HKDF(Chain Key, 0x02)
```

Listing 2.2: Symmetric-key ratchet procedure

where `HKDF` is as defined at Listing 2.1

The message key is first derived from the chain key which is then updated causing it to "ratchet" forward.

The message encryption/decryption key is derived from this message key in addition to a secret key that is used with a Hashed Message Authentication Code function (HMAC) to simultaneously provide data integrity and the data authentication for the message.

2.2.9 Diffie-Hellman ratchet

If a party's sending and receiving chain keys are compromised by an adversary then all future message keys can be computed by the adversary which compromises all future messages. The Double Ratchet inhibits this from occurring by combining the symmetric-key ratchet with a Diffie-Hellman ratchet which updates the chains based on the Diffie-Hellman shared secrets.

Periodically, a DH ratchet proceeds with each party generating a new ephemeral Diffie-Hellman key pair which becomes their current ratchet key pair. Each message sent by a party includes their current ratchet public key. When a new ratchet public key is received from the other party say Bob by a party say Alice, the Alice's current ratchet private key is used with the Bob's ratchet public key to derive a new receiving chain and update the root chain. Then, when Alice wants to send a message, Alice replaces her current ratchet key pair with a newly generated ephemeral DH key pair and combines it with Bob's ratchet public key deriving a new sending chain and root chain. This continues in a "ping-pong" pattern of the parties taking turns in replacing ratchet key pairs and updating the chains as illustrated in the following diagrams [15].

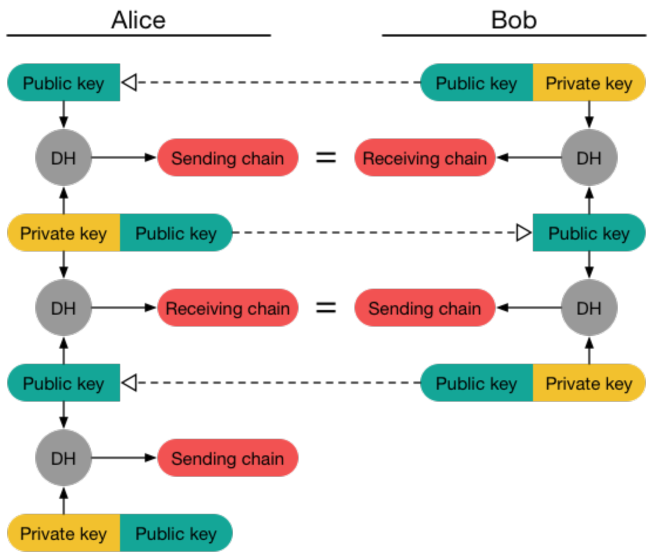


Figure 2.3: Diffie-Hellman ratchet "ping-pong"

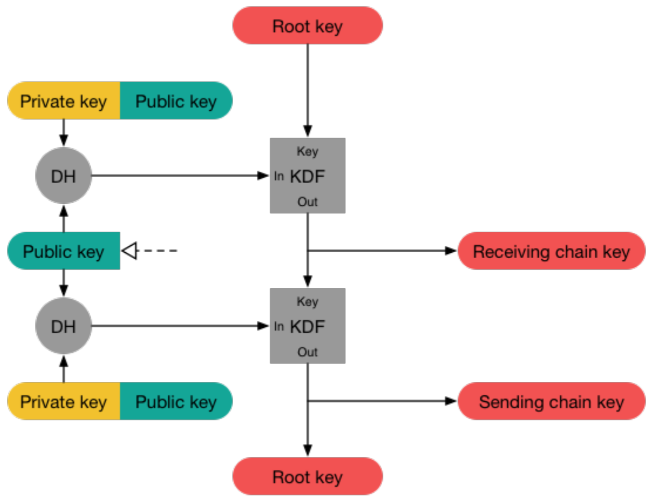


Figure 2.4: Diffie-Hellman ratchets updating the chains

The new chains are calculated as follows [17]:

```
Ephemeral Shared Secret = ECDH(Sender DH Key, Receiver DH Key)
Chain Key, Root Key = HKDF(Root Key, Ephemeral Shared Secret)
```

Listing 2.3: Diffie-Hellman ratchet chain update procedure

where `ECDH` is the Elliptic-curve Diffie–Hellman function which derives the DH shared secret and `HKDF` is as defined at Listing 2.1

N.B. as mentioned in the beginning §2.2, the initial value of the `Root Key` is derived through a key agreement protocol.

If an adversary compromises any DH private key, the adversary can only eavesdrop for a limited time as the DH private key is ephemeral and is replaced soon enough. As more DH ratchets are performed, increasing randomness is added to the E2EE, continually achieving perfect forward secrecy as well as post-compromise security.

2.2.10 Double ratchet

Combining the symmetric-key ratchet with the DH ratchet gives the Double Ratchet:

- When a message is sent or received, a symmetric-key ratchet is performed to the sending or receiving chain to derive the message key as defined in Listing 2.2

- When a new ratchet public key is received, a DH ratchet is performed before the symmetric-key ratchet to update the chains as defined in Listing 2.3

2.3 Existing solutions

While both implement E2EE via the Signal Protocol, Only WhatsApp uses E2EE by default for all conversations whereas Facebook Messenger requires the user to manually initiate a E2EE conversation for every other user that they wish to securely converse with. This fact already constitutes a major security vulnerability for those messaging apps as all conversation by default are in the clear without any protection meaning an adversary can easily intercept and read their messages. Moreover, lay users are unlikely to expend the effort to manually initiate a E2EE conversation if they can conveniently converse without it or may use the default conversations thinking its secure due to their lack of expertise in security. This turns security into a “feature” rather than an essential requirement.

Furthermore, they require a *phone number* and other personal information to be used that is all stored on a central server. Even if the messages are E2EE, there’s still a great deal of data that can be collected about their them (sent time, recipient, etc.) since all messages are relayed via the central server. Additionally, since both apps are owned by Facebook, the phone number can used to identify their Facebook profile and collect more data to sell as they previously have [1, 8, 9] profoundly undermining the user’s privacy.

As we have learned in the Signal Protocol (§2.2), all the messages are cryptographically signed and authenticated using a ephemeral identity key pair whose public key is uploaded to a central server. If a user say Alice wants to send an E2EE message to another user say Bob, then it requests Bob’s public keys from the central server. The approach employed here is a based on a *centralised Public Key Infrastructure (PKI)* approach where the trust is centralised to a central entity, the central server here, which manages and distributes the public keys. The trust is centralised here in the sense that a user requesting the public key of another user from the central server trusts the central server to return the true public key of the user requested. However, this design presents a major security vulnerability as it is susceptible to a **Man-in-the-Middle Attack (MITM)** wherein the communication between two parties is monitored and may be modified by an unauthorised third party[1].

The unauthorised third party in this case of a MITM is the central server which acts as a relay between Alice and Bob. When the Alice ask for the public keys of Bob to send an

[1]See https://www.cloudflare.com/learning/security/threats/man-in-the-middle-attack/

E2EE message, the central server can simply generate a new set of key pairs which it controls and give its own public keys back to Alice telling it that they are Bob's public keys. When Alice sends an encrypted message to Bob via the central server, the central server can decrypt it using its own keys, modify and re-encrypt it using Bob's actual public keys before relaying the message to Bob. The central server has then compromised the E2EE between Alice and Bob covertly eavesdropping on all their conversations even modifying their messages. This is can be done for all conversations on the messaging service compromising them all. Moreover, should the central server get compromised by an adversary, the adversary can compromise all conversations the same way. No implementation guards against this as they are all centralised.

The ephemeral identity key pair presents another security vulnerability as it allows a MITM to be performed by an external adversary, say Mallory. When Alice uninstalls the app or her device becomes inaccessible to her (i.e. damaged, lost, etc.), Alice has to generate a new identity key pair and uploading the new public key to the central server upon logging into the app again. To universally verify the identity of the user and universally identify the user upon log in, the central server uses the user's phone number where a SMS verification is performed by sending a ephemeral secret code in a SMS text message to verify the user's identity. However, this process has been shown to be easily compromised by an adversary [12]. Even with a perfect implementation of SMS verification, SMS text messages have been shown to be vulnerable to hijacking due known flaws in the cell networks [2]. Since there is no conclusive way for other users in conversation with the Alice and even the central server to know whether the new identity key truly belongs to Alice, Mallory can generate her own identity key and exploit this reset mechanism to pretend to be Alice to every other user (like Bob) that Alice was already in conversation with and the other parties to Alice. Mallory has then compromised the E2EE encryption between Alice and the other parties Alice was in conversation being able to eavesdrop and even modify their messages.

Implementations attempt to guard against the ephemeral identity key pair MITM through optional in-person verifications of the identity key. Parties can optionally verify their E2EE session after one has been established by each user scanning the other user's QR code of their encoded identity key and vice versa [17]. This must be done in-person between any given pair of parties for every device they own since there is an identity key per device. This exacerbated by the fact that this process must be repeated again every time the identity key for a user changes when, for example, they login from a new device. As result, numerous manual in-

[2]See https://www.theverge.com/2017/9/18/16328172/sms-two-factor-authentication-hack-password-bitcoin

person verifications may be continuously required for the E2EE to be effective. Considering the continues physical effort this demands, the E2EE can be practically be rendered as futile.

Chapter 3

Requirements

This project had numerous requirements including nonfunctional requirements, functional requirements and hardware requirements. I used to various software engineering methods as well as design techniques to derive a set of requirements which have been divided up into the appropriate sections as follows.

3.1 Nonfunctional Requirements

The system should not only be optimised for security and privacy but also be optimised for performance and usability. The system has to satisfy a number of nonfunctional requirements. The system has to:

1. Provide privacy.
2. Be secure.
3. Be efficient.
4. Be reliable.
5. Be easy to to use.

3.2 Functional Requirements

The system needs to be satisfy the functional requirements. The system must:

1. Provide the user with a GUI to interact with the system.

2. Enable the user to start a conversation with another user.
3. Enable the user to send and receive E2EE messages with all the security properties of existing systems (as defined in §2.2.1).
4. Enable the user to send and receive E2EE files including images.
5. Be secure against the man-in-the-middle attack vulnerabilities of existing systems (as delineated in §2.3).
6. Be secure without requiring any in-person verifications.

3.3 Hardware Requirements

Since the system's target are major desktop platforms, it has the following hardware requirements:

1. The computer must be running on operating system that is Linux (Linux-based), Windows or MacOS
2. The computer must have sufficient RAM to run the system
3. The computer must have sufficient storage to accommodate the system
4. The computer must have sufficient storage to accommodate the content the user creates in the system (text messages, images, files, etc.)
5. The computer must be connected to the internet

Chapter 4

Design & Specification

This chapter discusses the design decisions made in the process of developing the system and delineates the design of each part of the system. The motivation behind every design decision made was either the overall aim of the system or one of the aforementioned requirements.

4.1 The Solution

The root cause of the vulnerabilities we highlighted with existing solutions in §2.3 is their centralised architectures as it necessitates all users to trust a central server unconditionally in order to use the messaging service. As shown, this compromises the user's privacy and security.

This can be addressed by using a decentralised architecture instead where no central server needs to be trusted. Using such an architecture, users would be able message each other directly, *peer-to-peer*, instead of using a central server to relay their messages. This would not only provide strong privacy to users but also make the system and messaging more efficient thereby making them faster. Additionally, a decentralised architecture significantly reduces the cost of operating the messaging service with the massive overhead of storing and relaying messages being eliminated.

Furthermore, the ephemeral identity key MITM vulnerability can be resolved by using a *Decentralised Public Key Infrastructure (DPKI)* involving long-lived established identities instead of the centralised one. We will use use *Pretty Good Privacy (PGP)* as our DPKI as it the most widely used email encryption system in the world [18] allowing users who already use it to simply use their existing established identity. PGP is based on public-key cryptography using a portable long-lived key pair called a *PGP key* to represent the identity of a user universally.

These PGP keys are shared without a central server directly between users. Replacing the ephemeral identity key with a PGP key that is long-lived and portable eliminates the reset mechanism of existing solutions that makes the system susceptible to the MITM. Moreover, no in-person verification is required as the user identity is already established and long-lived.

We explain our usage of it in the following.

4.2 Pretty Good Privacy (PGP)

PGP employs a decentralised trust model known as *Web of Trust* in which any user can act as a *Certificate Authority (CA)* [2], an entity that issues *digital certificates*. The idea of a web of trust is that a user verifies the identity of another user and decides to trust the other user to trust users for the user. This is done by the other user issuing a *digital certificate*, a digital document that certifies the user has verified the identity of another and trusts it. So if one verifies that Alice's certificate is actually for Alice and one verifies that Bob's certificate is from Bob, Alice and Bob can both verify that Charlie is in fact Charlie for one. Due to this, no extensive in-person verifications are required like with the Signal Protocol.

Since PGP uses public-key cryptography, PGP provides identity verification, data authentication and data integrity [2] which, when coupled with the Signal Protocol, provides the system with all the security properties (as defined in §2.2.1) demanded by our requirements.

While PGP can be and is usually used to encrypt data as in the case of email, we will only use it to sign and authenticate all communication between parties. This is because PGP encryption does not offer any of the security properties that are demanded by our requirements while the Signal Protocol does and always produces a long output for even small inputs [6] making its encryption much less secure and efficient.

We will use OpenPGP, the open standard of PGP encryption software.

4.3 System Architecture

As explained in §4.1, the system will have a decentralised architecture for increased the user privacy and efficiency. While the system architecture is decentralised, a server is still needed to facilitate the users in finding and connecting with each other.

The system consists of two major sub-systems:

- **Application** – a client that provides all the main interface and functionality of the system directly to the end user (like messaging)

- **Server** - a server that runs in the background enabling users to establish chats and connections with each other

4.4 Application

4.4.1 Framework

To better satisfy our requirements, we will use the **ElectronJS**[1] framework over native desktop application development or other cross-platform frameworks like *Swing* for the following reasons[2]:

- **Single code base**: A single code base for all desktop platforms (Linux, Windows and macOS). Code once, distribute everywhere.
- **Easy-to-use high-level APIs**: The Electron runtime provides high-level APIs for native operating system functionality that abstract the low-level complexities and are easy to use speeding up development.
- **Rapid development**: Electron uses JavaScript and web technologies with the GUI being defined in HTML & CSS, making it as rapid as web development.
- **Powerful GUI**: Since the GUI is defined in HTML & CSS, a much more enhanced and consistent user interface (and experience) can be delivered like on the web.
- **Most libraries**: JavaScript offers the greatest number of software libraries of any language in the world through its software registry NPM[3] which further accelerate development.
- **Automatic updates and easy installers**: offers automatic update for the application and ships the built application with convenient installers that bundle all application components together.

The framework runtime environment consists of two parts[4]:

- **Main** - the *back-end*, the main process running the main app code in a *Node.js* environment. Code is written in *JavaScript*.

[1]See https://www.electronjs.org
[2]See https://dzone.com/articles/what-is-electron-amp-why-should-we-use-it
[3]See https://www.npmjs.com
[4]See https://www.electronjs.org/docs/tutorial/application-architecture

- **Renderer** – the *front-end*, renderer process(es) running the Graphical User Interface code in a *Google Chrome browser-based (Chromium)* environment. Code is written in *JavaScript*, *HTML* and *CSS*.

Hence, the project structure will reflect this.

Both the backend and frontend processes run independently (i.e. have separate address spaces). Thus, *Inter-Process Communication (IPC)* will be used to communicate between the backend and frontend processes through *event-driven programming*[5].

4.4.2 Graphical User Interface Framework

To better satisfy our requirements, we will use the **React**[6] JavaScript framework to build the GUI over just plain JavaScript/HTML/CSS and other frameworks for the following reasons:

- **Component-Based**: GUI is declared as *components* that manage their own state and can be composed to build complex GUIs.
- **Reusable**: *components* can be reused to build modular and maintainable GUIs.
- **Declarative**: *components* are declarative views that make code more predictable and easier to debug.
- **Efficient**: React efficiently updates and renders just the right *components* when their data changes.

With React, all the GUI is split into React *components* recursively until the GUI is as modular as it can be. Components are synonymous to functions that take as input some data and use it to render some GUI. This input can can be `props` (properties) which is readonly data and/or `state` which is dynamic data (may change over time). The most important difference between `state` and `props` is that `props` are passed from a parent component, but `state` is managed by the component itself. Every time any of this data changes, React efficiently updates and renders just the appropriate *components*, efficiently keeping the GUI up-to-date.

The GUI is defined in the Graphical User Interface section (§4.6).

4.4.3 Storage

All application user data will be stored in a **LevelDB**[7] database to make storing and retrieving it. No cross-platform database solution for use exists by default so we will have to ship one

[5]See https://en.wikipedia.org/wiki/Event-driven_programming
[6]See https://reactjs.org
[7]See https://github.com/google/leveldb

with the app. We choose LevelDB as it is *reliable*, *fast*, uses *compression* by default and, most importantly, *lightweight*[8] which is important as it ship with the app.

As per our requirements, to ensure security, all private keys will be stored encrypted so that even if the database is compromised they are secure.

However, the *passphrase* of the PGP key which is used to decrypt the private key and, hence, is the most sensitive data will be stored and retrieved from the operating system's keychain. This delegates their security to the operating system's secure password management system and ensures system-wide protection of this sensitive data. Since the only safe alternative to this is prompting the user for it every time the app is launched, this also provides a profoundly better user experience and, hence, better satisfies our requirements.

4.4.4 End-to-end encryption

We will implement E2EE via the Signal Protocol as defined in §2.2 but with one major difference. As discussed above, to remedy existing vulnerabilities, we will be using PGP keys instead of identity keys. This means that we need to integrate it into the Signal Protocol and update it accordingly.

Using PGP, the existing session setup can be significantly simplified. In the updated key agreement protocol, to agree on the initial root key, each party generates an initial *Curve X25519* key pair and sends its public key, the *initial public key*), to the other party after signing it with their PGP key. On receiving the party's public key, the other party sets the initial root key to the Diffie-Hellman shared secrets derived from combining both of their keys (ECDH). Both party's perform a DH ratchet on sending their first message, after which, they use the Signal Protocol as defined in §2.2 (i.e. the Double Ratchet algorithm).

[8]See http://highscalability.com/blog/2011/8/10/leveldb-fast-and-lightweight-keyvalue-database-from-the-auth.html

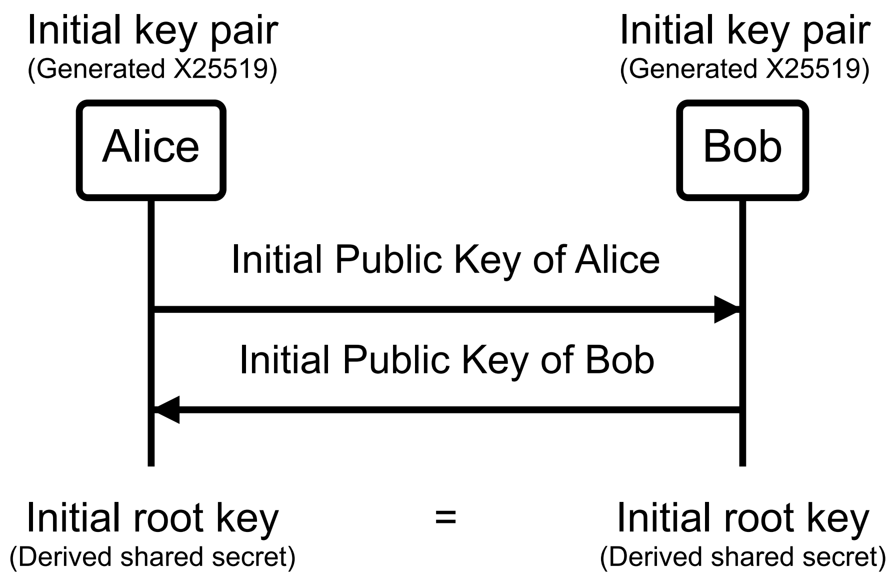


Figure 4.1: Key exchange protocol

4.4.5 Peer-to-Peer messaging

As we have mentioned in §4.1, we aim to decentralise the system especially messaging. After the initial chat and connection is established between two users using the *signalling protocol* as defined in §4.5.2, all communication between them is direct or *Peer-to-Peer (P2P)* meaning no third-party (like a server).

All messages from either user will be sent P2P from one app to another on top of being end-to-end encrypted to best guard their privacy. As existing solutions allow for images and files to be sent as well in their messages, the app will also send images and files P2P providing a rich user experience while providing the strongest privacy.

To initiate the E2EE messaging, a E2EE session has to be established as defined in §4.4.4. As shown in Figure 4.1, an initial key pair is generated and the initial public key is exchanged. This exchange occurs via a *key message*.

Within our framework, the **WebRTC**[9] (Web Real-Time Communication) P2P protocol will be used to provide this as it the standard for P2P on the web.

A p2p *message* should have following attributes:

- **type**: The type of the message, either `message` for a message or `key` for the user's initial

[9]See https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API

session public key (see §4.4.4).

- **timestamp**: The time the message was created.

- **signature**: The PGP signature of the message by the sender.

A message sent by a user should have the following additional attributes:

- **id**: The unique id of the message, can be the hash of the message.

- **content**: The content of the message, the message text or binary encoded as a hexadecimal string in the case of an image or file.

- **contentType**: The type of the content of the message, should be `text`, `image` or `file` appropriately.

A *key message* should have the following additional attributes:

- **key**: The *initial public key* of the sender (see §4.4.4).

4.5 Server

Runs in the background providing a protocol for the Application but no GUI.

It serves two main functions in the system:

- **Discovery**: allow a user to find and establish a chat with another user by their unique `userId`

- **Signalling**: relay all the connection information (signals) from one user to another user to enable a connection between them to be established

The server will be as *stateless* (i.e. will forget everything related to application state) as possible to reduce complexity making it less prone to bugs and providing a better separation of concerns (SoC).

4.5.1 Discovery

A `userId` is required for every user to universally identify and find them. This needs to be globally unique while being easy to use (as per our requirements). While we use a PGP key to represent the identity of a user and is globally unique, it is far too long [6] to be easy to use. However, upon a closer inspection of the OpenPGP specification, we find two potential candidates for the `userId`: the key ID and the key fingerprint [13]. Further research shows that the key IDs are susceptible to a collision attack with relatively small possible permutation space of 2^{64} meaning two or more different PGP keys can have the same key ID so the key ID is not globally unique [10]. Fortunately, the key fingerprint has a sufficiently large permutation space of $2^{1}60$ to prevent a collision attack from occurring making it globally unique. It is also only 40 characters long when encoded in hexadecimal making it easy to use and share. In fact, it is not only easy to use but it can be simply derived from the PGP key of the user one wants to message requiring no exchange of it if the PGP key is already possessed.

The fingerprint for the `userId` combined with the PGP key fulfil the function of universally identifying the user and universally verifying the identity of the user for which existing solutions use the user's *phone number* for as discussed in §2.3. Hence, we can drop the use of the user's phone number to guard the privacy of the user (as we set out in our requirements). Moreover, this no further information is stored on the server further guarding the privacy of the user.

Discovery leverages this `userId` by enabling users to establish a chat with another user by their unique `userId` in the *discovery protocol.* A chat between a user *Alice* and another user *Bob* is established via the discovery protocol as shown in the following diagram:

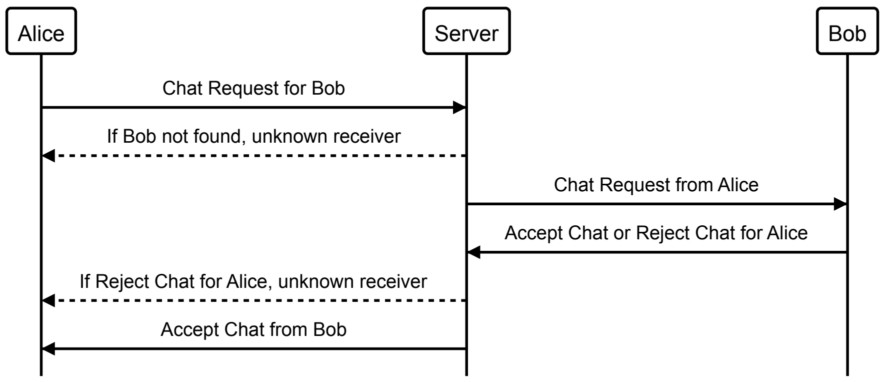

Alice
Server
Bob
Chat Request for Bob
If Bob not found, unknown receiver
Chat Request from Alice
Accept Chat or Reject Chat for Alice
If Reject Chat for Alice, unknown receiver
Accept Chat from Bob

Figure 4.2: Discovery protocol

4.5.2 Signalling

Relays all the low-level network connection information packets, *signals*, from one user to another user to enable an initial connection between them to be established. To enable this reliably, we will use **WebSocket (WS)** as opposed to **HTTP** (long pooling) for several reasons [11]. The first is that we are building a real-time application which means that latency is a central concern (as set out in our requirements). WS has much less overhead as it doesn't send extra

[10]See https://debian-administration.org/users/dkg/weblog/105
[11]See https://www.ably.io/blog/websockets-vs-long-polling

data like HTTP headers reducing latency significantly. It also provides completely bi-directional communication between the client and the server allowing messages to be streamed between them concurrently. Most importantly though, WS establishes a persistent connection between the server and the client which allows the server to keep track of all the users currently online in real-time and relay messages between users more reliably better meeting our requirements. When a connection with a user drops (i.e. user disconnects), the server instantly knows that the user has gone offline and can respond to message relay requests to the user properly.

Once a connection with the server is established by both users, signalling can be performed as follows to establish an initial connection between the users:

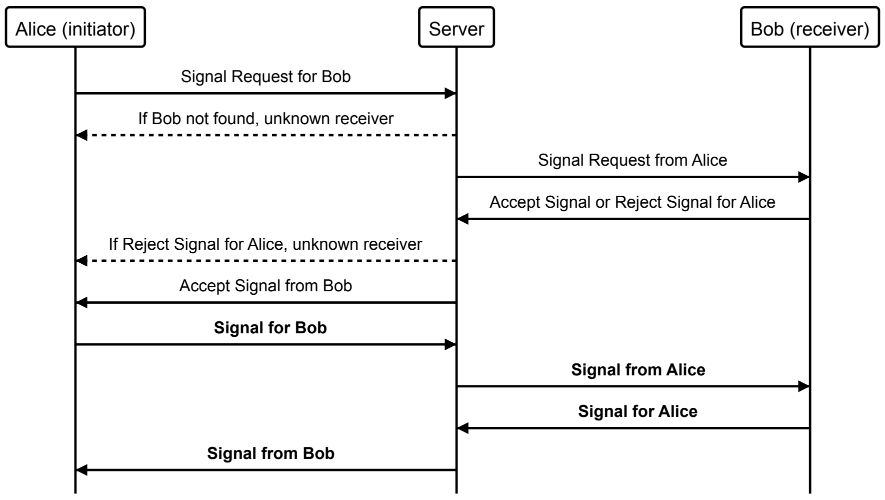

Alice (initiator)
Server
Bob (receiver)
Signal Request for Bob
If Bob not found, unknown receiver
Signal Request from Alice
Accept Signal or Reject Signal for Alice
If Reject Signal for Alice, unknown receiver
Accept Signal from Bob
Signal for Bob
Signal from Alice
Signal for Alice
Signal from Bob

Figure 4.3: Signalling protocol

As shown in the figure, an initial "handshake" is performed so both users can prepare to receive the signals and establish the connection. Multiple signals may be sent concurrently by both parties to each other via the server.

4.5.3 Authentication

A simple authentication will be performed when the user connects with the server to verify the identity of the user (i.e. their PGP key). To authenticate, the user will have to sign a timestamp with their PGP key and the server will verify their signature. The signature expires after *1 minute* (when the timestamp is older than that) so that if the signature is later compromised by an adversary it cannot be used to imitate the user and compromise the authentication.

4.6 Graphical User Interface

The user interface of the application is what allows the user to interact with the entire system. It should meet all requirements the we set out in §3. The user interface is explained as follows in the chronological order in which the user should experience it on the first run of the application.

4.6.1 Setup Identity

This is the first GUI screen that is seen by a user upon the running the application for the first time or if an identify isn't setup yet. It prompts the user to setup an identity represented by a PGP key giving the user the option to either create a new identity (PGP key) or import an existing identity (PGP key). Depending on the choice, the next GUI screen is either *Create identity* (§4.6.2) or *Import identity* (§4.6.3).

4.6.2 Create identity

This GUI screen is shown upon the user choosing to create an identity in the initial *Setup Identity* (§4.6.1) screen. The screen prompts them to specify all the details needed to generate their PGP key:

- **Name**: their name (the name by which the users they want to converse recognise them by).
- **Email (optional)**: optionally their email address.
- **Passphrase**: a secret phrase used to keep the private key of their PGP key secure, acts like password.
- **Key algorithm**: one of the PGP key algorithm as defined in the specification [13] defaults to `RSA-4096`[12].

When the user submits these details, the details are validated. If they are invalid, an appropriate error message is shown. If they are valid, the PGP key is generated using the details, securely saved internally and then displayed in a dismissible dialog including its public key and *encrypted* (using the specified *passphrase*) private key along with their `userId` so they can save or share it. On dismissing this dialog, the user is shown the main *Messenger* screen (§4.6.4)

[12]This is a more secure option than even what the *National Institute of Standards and Technology NIST* recommends (RSA-2048) https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf

4.6.3 Import identity

This GUI screen is shown upon the user choosing to import an identity in the initial *Setup Identity* (§4.6.1) screen. The screen prompts them to specify their existing PGP public and private key along with its passphrase if one is set. When the user submits these details, the details are validated. If they are invalid, an appropriate error message is shown. If the details are correct, the keys are securely saved internally and the user is shown the main *Messenger* screen (§4.6.4)

4.6.4 Messenger

This is the main GUI of the application as it shows all the chats and lets the user compose new chats and messages. If the identity is setup, this is the first screen shown on every run of the application. It uses the intuitive chat design that modern chat applications such as Facebook Messenger use. The screen is split into two major sections with a toolbar spanning across them at the top showing the actions the user can perform for each section which are delineated as follows.

Chat list

A a left sidebar with a list of all the chats created by the user. Every chat shows the name of the user the chat is with, the last message received from him/her and its time. When the user selects a chat by clicking on it, its list of messages are shown in the **Message list** section. There is also a delete button displayed next to the every chat that when clicked removes the chat along with all its data. In the toolbar, it has a compose chat button that leads to the *Compose chat* (§4.6.5) screen when clicked. A green circle is also shown on the chats where the user, whom the chat is with, is currently online as a visual feedback.

Message list

A list of messages of the chat currently selected by the user. There is a compose message input at the bottom of the section that allows the user to enter the message to and has buttons for sending an image or file on the right of input. In the toolbar, it has a chat info button at the top that leads to the *Chat info* (§4.6.6) screen when clicked.

This design makes everything the user needs readily accessible.

On the start of the application, a connection is established with every user for which a chat exists via the server through the signalling protocol (as shown in Figure 4.3).

4.6.5 Compose chat

This screen is shown when the user clicks the compose button in the *Messenger* screen (§4.6.4). The screen presents the user with an input for the `userId` or PGP key of the user that they want to chat to. If the input is a PGP key, the `userId` is derived from it. When the user submits the input, it is validated. If it is invalid, an appropriate error message is shown. If it is valid, the chat establishment protocol is performed to establish the chat as shown in Figure 4.2. Firstly, a chat request is sent to the `userId` specified via the server. If the user with the `userId` cannot be found or rejects the chat request, an error message is shown saying the user wasn't found for both cases to guard the other user's privacy. The chat is added to the *Messenger* screen (§4.6.4) and a connection is established via the server (as shown in Figure 4.3) if the chat request is accepted.

4.6.6 Chat info

This screen is shown when the user clicks the chat info button in the *Messenger* screen (§4.6.4). This screen shows all the details of the user the chat is with including their `userId`, full name and PGP (public) key. It also gives the user the option to delete the chat.

4.7 Libraries

This section discusses all the major third-party software that will be used to produce the system and accelerate its development.

4.7.1 React

React[13] is used to build a GUI that better meets the requirements demand (as explained in §4.4.2).

4.7.2 Simple-Peer

Simple-Peer is used to provide Peer-to-Peer messaging via WebRTC. Simple-Peer provides a Node.js interface for WebRTC via data channels that allow text and binary data to be sent over them.

[13]See https://reactjs.org

4.7.3 OpenPGP.js

OpenPGP.js[14] is a JavaScript implementation of the OpenPGP protocol [13] which is used to perform all PGP-related activities required by the application.

4.7.4 LevelDB

All application user data is stored in a LevelDB[15] database to make storing and retrieving it reliable and quick. LevelDB is a fast key-value NoSQL database built by Google.

4.7.5 TweetNaCl.js

Since the standard library of Node.js does not support *Elliptic-curve Diffie–Hellman* with *Curve X25519* needed for implementing the Signal Protocol (§2.2) yet, we will use the JavaScript port TweetNaCl.js[16] of the high-security cryptographic TweetNaCl library[17] to provide this functionality.

4.7.6 Moment.js

Moment.js[18] is used to parse, validate, manipulate, and display dates and times in the application.

4.7.7 KeyTar

KeyTar[19] is used to securely store and retrieve the PGP key passphrase. KeyTar is a library to get, add, replace, and delete passwords in a operating system's keychain. On macOS the passwords are managed by the Keychain, on Linux they are managed by the Secret Service API/libsecret, and on Windows they are managed by Credential Vault.

4.7.8 Ionicons

To provide the icons to depict the actions required by the GUI in the application, the Ionicon[20] icon font will be used.

[14]See https://github.com/openpgpjs/openpgpjs
[15]See https://github.com/google/leveldb
[16]See https://github.com/dchest/tweetnacl-js
[17]See http://tweetnacl.cr.yp.to
[18]See https://momentjs.com
[19]See https://github.com/atom/node-keytar
[20]See https://github.com/ionic-team/ionicons

Chapter 5

Implementation

In this chapter we will discuss how we developed and implemented our design as delineated in the previous chapter. We discuss our development approach first and then the implementation and any issues we encountered during it.

5.1 Development Approach

Please refer to Appendix A.1 for the class diagrams of the implemented system and Appendix C for the source code of the implemented system.

Since we used an iterative *Agile development* approach, the class diagram was not included in the Design and Specification chapter. This was due to many implementation-specific parts that had to be figured out gradually during development.

The "divide and conquer" paradigm was employed to successfully produce a maintainable and efficient system. This involved successively dividing the design into its component parts that were small enough to be completed within time-boxed iterations, *sprints*. Since development was iterative, many components were usually being developed and improved concurrently. An overview of the order of development is as follows:

1. The base of the App and the Setup screen: `index.js`, `main.js` (window), `App.js`, `Modal.js` and `SetupIdentityModal.js`. (§4.6.1)

2. The Create identity screen and PGP key generation implementation: `CreateIdentityModal.js`, `Crypto.js`. (§4.6.2)

3. The Storage of user information and PGP keys: `Crypto.js` (iteration). (§4.4.3)

4. The Import identity screen and PGP import implementation: `ImportIdentityModal.js` and `Crypto.js` (iteration). (§4.6.3)
5. The Chat list of Chat screen along with its base: `Messenger.js`, `ChatList.js` and `Chat.js`. (§4.6.4)
6. The Message list of Chat screen along with its Toolbar and Chat info screen: `MessageList.js`, `Message.js`, `Compose.js`, `Toolbar.js` and `ToolbarDropdown.js`. (§4.6.4, §4.6.6)
7. The Storage of Chats: `Chats.js`. (§4.4.3)
8. The base of the Server and the Discovery protocol: `server.js` and `Server.js`. (§4.5, §4.5.1)
9. The Server Signalling protocol: `Server.js` (iteration). (§4.5.2)
10. Implementation of Discovery and Signalling protocols in the App: `Server.js`. (§4.5.1, §4.5.2)
11. The Compose Chat screen, iterating: `ChatList.js` and `MessageList.js`. (§4.6.5)
12. Peer-to-peer messaging: `Peers.js`. (§4.4.5)
13. End-to-end encrypted messaging via the Signal Protocol: `Peers.js` and `Crypto.js`. (§2.2, §4.4.4)

The components were generally implemented in the chronological order they will be seen and used by the user as to test and iterate over the user experience along with the functionality. All the details in the Design chapter were used to implement the system.

As security and privacy is such a central concern, the security and privacy of the system was considered at every stage of development with it informing every decision made.

5.2 Implementation

The system was implemented as it is defined in the Design and Specification chapter. Any changes and deviations from that have been documented in §5.3 and §5.4. The following presents an overview of the implementation.

5.2.1 Project structure

Application

As mentioned in the Application section in Design and Specification chapter (§4.4), the Application consists of two parts: *Main*, the back-end; *Renderer*, the front-end. Considering this, the Application has the following folder structure:

- / **src**: contains all the source code
  - / **main** - contains all the *back-end* source code
    - / **lib** - contains all the *core modules* that provide all the Application functionality.
    - / **windows** - contains the code to create the *main window* which renders the **index.html**
    - / **index.js** - the entry point of the Application, implements the Application using the *core modules* to provide all the functionality and the *main window* to provide the GUI.
  - / **renderer** - contains all the *front-end* source code
    - / **lib** - contains common modules needed by the *React components*
    - / **components** - contains all the *React components* that make up the GUI.
    - / **index.js** - uses all the *React components* to create a *React application* and provide the GUI
  - / **config.js** - contains the Application development and production configurations that are resolved automatically
  - / **consts.js** - contains the Application-wide constants
- / **static**: contains all the static assets of the Application
  - / **scss**: contains all styles of the GUI written in *SASS*, a superset of *CSS* which is compiled down to it
  - / **fonts**: contains all the fonts used in the GUI (just an icon font)
  - / **index.html**: is a HTML file that renders the *React application* (see **src/renderer**) in the *main window* (see **src/main/windows**) to show the GUI

Server

- / **src**: contains all the source code

- / **lib** - contains all the *core modules* that provide all the Server functionality
  - / **Server.js** - the *Server class* that implements a WebSocket Server and provides all functionality.
  - / **utils.js** - utilities for the Server class
- / **schema** - contains all the formal descriptions of the protocol formats (i.e. schemas)
  - / **message.json** - a formal description of the protocol message format and structure.

/ **server.js**: the entry point of the Server, initiates the *Server class* with a *Node.js HTTP Server* and runs the Server

/ **config.js**: contains the Server development and production configurations that are resolved automatically

5.3 Implementation issues

When turning a design into a product, unforeseeable issues tend to always arise. Fortunately, every issue encountered has been through continuous iteration of the product and testing. Moreover, this process has also made the product much more reliable.

5.3.1 Connection race condition

A user connects with another user through the *signalling protocol* as defined in §4.5.2. In this process, one user has to be the *initiator* while the other user is the *receiver* in order for a connection to be established as per the WebRTC protocol[1]. When a new chat is established through the *discovery protocol* as defined in §4.5.1, the user who initiated the chat is the initiator while the other user is the receiver. However, on application start up, every user attempts to a connect to every other user it is in conversation with as the initiator which causes a *race condition*[2] as both try to be the initiator when the protocol requires only one to be. This, ultimately, results in both trying to connect each other as initiators which causes the connection to fail.

To address this, an initiator agreement strategy was required. A simple initiator strategy we implemented was that the user who first sent the request to connect was to be the initiator. This was determined using a timestamp of the time the request was sent that was sent along

[1]See https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity
[2]See https://techterms.com/definition/race_condition

with every signal request. So if the application had already sent a signal request to a user but then received a signal request from that user, the application would check the timestamp of the received signal request against its sent signal request's one. If the received signal request occurred before the sent one, the application would become the receiver by replying with a signal accept making the other user the initiator. Otherwise, the application would be the initiator while the other user is the receiver.

5.3.2 Image and file transmission issue

When sending a file (arbitrary or an image) by setting its hexadecimal-encoded contents to the message's content, the app crashed. Debugging the app, we found that the crash was due to a buffer overflow in the Simple-Peer library. The message size exceeded the Simple-Peer sending buffer size so it overflowed and caused the app to crash. Upon further inspection, we found that the maximum message size that the WebRTC protocol could handle was 64 KiB [10] and the file easily surpassed that limit which the buffer size was set to.

To resolve this, the file up had to be split up into small discrete chunks of a size that was smaller than the WebRTC maximum message size before being sent and then had to be assembled on the receiving end. Fortunately, *Node.js* provided a native way to accomplish this with *Streams* enabling the file to be *streamed* from the sender to the receiver (i.e. sent and received chunk by chunk over time). Besides just enabling file transmission, this approach was also significantly more efficient both in terms of memory and time. In terms of memory, the data is buffered in memory so less of the memory is used. It is time efficient as data is processed as soon as it is available, rather than waiting till the whole data payload is available to start. Moreover, the file contents was sent in binary instead of a hexadecimal-encoded string. Thus, the app and network throughput was increased significantly.

However, this meant that the whole sending and receiving architecture had to be re-engineered to be Stream-based.

Streams

There are four types of Streams in Node.js[3]:

- **Writable**: Writes data to a destination.
- **Readable**: Reads data from a source.

[3]See https://nodejs.org/api/stream.html

- **Duplex**: a combination of **Writable** and **Readable** so can read and write data.
- **Transform**: a **Duplex** stream which reads data and transforms it before writing it back.

A Readable stream can be *piped* with a Writable stream to create a flow of data, a chunk of data at a time, from the source to the destination. Once all the data and written, the streams are closed and no more data can be read or written. Furthermore, Transform streams can be piped in-between the Readable stream and Writable stream to transform the data in some way before it is written, creating a *pipeline*. So to send and receive files using Streams, the following pipelines were implemented.

Sending pipeline

To send a file, the following pipeline is executed:

1. Read stream reading the data from the file
2. Transform stream performing the encryption on the data
3. Duplex stream writing the data to the receiver remotely (via a *Data Channel*)

Receiving pipeline

To receive a file, the following pipeline is executed:

1. Duplex stream reading the data from the sender remotely (via a *Data Channel*)
2. Transform stream performing the decryption on the data
3. Writable stream writing the data to a file locally

Data Channels

To send messages between two users over WebRTC, a default WebRTC Data Channel is established between the users when they connect to each other. In the Simple-Peer library, this channel is implemented as a Duplex stream. However, since this is a single Duplex stream, the default Data Channel closes after a single file is sent as the close of the file stream triggers the close of this Duplex stream. As no more messages can be sent, this closes the connection between the users.

This issue was solved by Data Channel multiplexing: creating a new Data Channel for every file transmission. Since a Data Channel is implemented by a Duplex stream, we create a new Duplex stream to write the file data on the sender's end and read it from the reader's end. This new channel is destroyed after the file is transferred. Furthermore, we create a new Data Channel with a message that contains metadata about the file like the *file name* to prepare the receive to receive the file (e.g. create the appropriate local file path). Keeping with our requirements and in the best interest of privacy, we encrypt the file name to provide confidentiality.

5.3.3 Backpressure

While the new Streams-based implementation worked well for smaller files, it caused the app to crash with larger files. Deeper inspection revealed that this was due to a *backpressure* issue. The sender was sending "Too Much Data, Too Quickly" for the receiver causing a data buildup at the receiver's end where the receiver's buffer was full and not able to receive additional data.

Fortunately, this was easily fixed by throttling the sending pipeline which slowed down the amount of data that was sent giving the receiver more time and less data to process at a time. To accomplish this, we used the *brake*[4] third-party software package which is Transform stream (see §5.3.2) that was placed in the sending pipeline. It took two arguments:

- **period**: the interval at which data is sent in milliseconds, set to `50ms` based on testing.
- **rate**: the amount of data in bytes that is sent every **period**, set to `16 KiB` as it is the optimum WebRTC message size for transmitting data over a WebRTC Data Channel [10]

This gave us the following updated sending pipeline

Sending pipeline

To send a file, the following pipeline is executed:

1. Read stream reading the data from the file
2. Transform stream performing the encryption on the data
3. Transform stream throttling the pipeline to `16 KiB / 50ms`
4. Duplex stream writing the data to the receiver remotely (via a *Data Channel*)

[4]See https://github.com/substack/node-brake

5.3.4 Out-of-order messages

Since files took longer to send and were being sent in separate Data Channels, new smaller messages (like text messages) sent arrived much sooner and were displayed first. This resulted in the messages being displayed out of order at the receiving end.

In order to resolve this, messages had to be queued on both the sending and receiving end to ensure they were sent and received in the right order. This was accomplished by creating a generic task *Queue class* (see `Queue.js`) that executed one task after another has finished executing even if it failed.

In JavaScript, a "task" can be represented by a `Promise` [5] language construct that finishes as either:

- **fulfilled**: the task completed successfully.

- **rejected**: the task failed as an error occurred.

The sending method (`_send` in `Peer.js`) and receiving method (`_onDataChannel` and `_onMessage` in `Peer.js`) are both written as `async` functions and, thus, return a `Promise`. The Queue class leverages this language construct to provide an agnostic[6] way to queue the sending and receiving functionality. Every user has a sending and receiving queue that is an instance of the Queue class. Every message/file sent by the user is added to their sending queue while every message/file received is added to their receiving queue and removed once they finish. If an error occurs (i.e. the task is *rejected*), the error is passed up the application to be shown.

5.3.5 Server authentication

Implementing server authentication presented a real challenge as the WebSocket protocol does not provide any way (like HTTP does with *headers*) for the application to specify any user information to authenticate the user while connecting with the WebSocket server.

The only way to accomplish this seemed to be for the application to perform a separate HTTP authentication flow where it authenticated with a HTTP server obtaining an authorisation "ticket". This it would then send via a cookie as part of an initial handshake when opening the WebSocket connection[7]. This is far to complex for our authentication case (see §4.5.3).

Upon closer inspection of the WebSocket connection procedure, we discovered a novel way of specifying the authentication information could be as part of the WebSocket connection *uniform*

[5]See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[6]See https://whatis.techtarget.com/definition/agnostic
[7]See https://devcenter.heroku.com/articles/websocket-security

resource identifier (URI). Authentication is done by encoding the authentication information (the public key, signature and timestamp) as a *query string*[8] like so:

```
?publicKey=<public key>&timestamp=<timestamp>&signature=<signature>
```

Then appending this to the URI and connecting to the WebSocket server with it.

On the WebSocket server side, the query string is parsed and the authentication information is validated. If the signature is valid and less than *1 minute* old (as per see §4.5.3), the authentication is successful and the application is allowed to connect. Otherwise, the authentication fails and the connection is refused. This ensures strong security,

5.4 Changes and additions from design

5.4.1 Chat Info screen in the Messenger screen

While designing the GUI, it was discovered that most chat applications made the the Chat Info screen a part of the main screen that is the Messenger screen. So following this heuristic, we also integrated the Chat Info screen into the Messenger screen. It was implemented as a dropdown that showed up when the user clicked the chat info icon and disappeared when the user clicked on an action or clicked the chat info icon again.

5.4.2 Compose chat screen in the Messenger screen

While designing the GUI, it was discovered that most chat applications made the the Compose chat screen a part of the main screen that is the Messenger screen. So following this heuristic, we also integrated the Compose chat screen into the Messenger screen.

When the user clicks the compose chat icon, a new chat placeholder for the chat is created and the toolbar becomes an input for the `userId` or user's PGP key who the new chat is with. When this input is populated by pasting the details or entering them and functions just as described in the Compose chat design (see §4.6.5).

5.4.3 User Info screen

After implementing the GUI, I realised that there was no way for the user to see his own details so I added a User Info screen in the main Messenger screen. It is the same as the User Info screen but contains the user's info instead of the chat details. This is the same case for its actions.

[8] See https://en.wikipedia.org/wiki/Query_string

5.4.4 Deleting chats

In addition to composing chats, it was natural to implement the functionality to delete chats. This is done by simply clearing all data stored about a chat and disconnecting from the user whom the chat is with if still connected with them.

5.4.5 State

To store the application state, a State class was also added. It currently only stores the chat id (`userId`) of the chat most recently selected by the user. This so that when the user re-launches the application, the chat the user was engaging with before the user quit the application is shown.

5.4.6 Dark Mode

To improve application usability, a Dark Mode was added to the application. In Dark Mode, the system uses a darker colour palette for all the graphical user interface. Refer to Appendix A.2.2 to see Dark Mode in action. Dark Mode uses light text on a dark background which reduces eye strain in low light conditions and improves visibility for users with low vision and those who are sensitive to bright light[9].

Functionally, with the majority of the screen being dark, Dark Mode reduces screen glare thereby also reducing flickering and blue light. It also helps users focus on their work[10].

A *Nature* scientific report[11] that explores the effects of contrast polarity on human eye even finds that dark mode (negative polarity) is significantly less harmful to vision than light mode in the long-term. The report finds that dark mode essentially inhibits the development of myopia (i.e. nearsightedness) in the eyes while light mode quickens the development of myopia.

[9] See https://developer.android.com/guide/topics/ui/look-and-feel/darktheme
[10] See https://support.apple.com/en-us/HT208976
[11] See https://www.nature.com/articles/s41598-018-28904-x

Chapter 6

Legal, Social, Ethical and Professional Issues

While developing the project, significant effort was made to follow *Code of Conduct & Code of Good Practice* of the *British computer Society (BCS)* every step of the way to avoid unintended any legal or ethical consequences. The BCS informs us to keep the best interest of the public at the forefront while developing our system. We are responsible for safeguarding the privacy and data stored of the public who uses our system. This is upheld by the very fact that the system is decentralised which means that all data of a user is kept by the user meaning the user is in full control of their own data. Good ethics are sustained, as in the case of a law enforcement request for the data of a user, no data can be shared as none is kept on any central servers safeguarding the user's privacy and liberties. This also establishes a trust between the user and the system.

Professional issues are combated through matriculate and fault free back-end engineering of the program. This involved testing the system countless times under different scenarios and conditions thus providing reliability and proficiency of the systems execution. Such tests include the system being run on various computers with the various supported operating systems (Linux, Windows and macOS), increasing the reliability of the systems. Furthermore, the users wellbeing and the accessibility of the system were important concerns. In the consideration of these, the system introduces a Dark Mode wherein a darker colour palette for all the GUI is used allowing the user to operate the system for a longer time under various different lighting conditions without straining their eyes (see more in §5.4.6).

During the development of the system, many steps were taken to safeguard and protect all intellectual property and avoid it being unlawfully used. Not only is it highly unethical to use intellectual property without the consent of its creator but it is also illegal. Due to this, a directive has been put in place to only use material which is public domain and open-source. Throughly-tested third-party open-source software has been leveraged by the system to better address professional issues faced. While we may know and are knowledgeable on certain aspects of the system, we lack some on other areas so using well-tested open-source software can not only remedy this but also accelerate the development of the system. A particular emphasis was placed in using open-source software that was published under a *permissive license* such as the *MIT License* as it allows for the greatest benefit to be reaped from the software[1]. Although all the open-source material is free to use and often does not require any attribution, it is an important ethical practice to highlight any involvement therefore all the external material is stated and credited where appropriate.

[1]See https://www.cio.com/article/3120235/what-the-rise-of-permissive-open-source-licenses-means.html

Chapter 7

Evaluation

7.1 Testing

Testing is a key part of the software development process to ensure the quality and reliability of the system delivered. The system has to meet all the requirements defined in §3. Due to the decentralised nature of the system and the intricacy of simulating it in order to unit test the modules, unit testing was not viable. However, a number of methods were still used to test the system thoroughly which are as follows.

7.1.1 Non-functional testing

Non-functional testing was performed by evaluating how the system produced performs against our non-functional requirements as defined in §3.1:

1. **Privacy**: the system is decentralised so all user data is held by the user and all messages are directly transmitted between users so only they hold them. Messages are E2EE so only the parties in conversation can read the messages. Users are identified by their user id (derived from their PGP key) instead of their phone number or any other personal information so only a PGP key is required to use the system. This can simply be generated on in the application locally so no registration is required to use the system.

2. **Security**: the application implements end-to-end encryption for all messaging to securely exchange messages between parties. Since the application implements the Signal Protocol (see §2.2) to accomplish this, all its security features (resilience, forward security, post-compromise security) are provided for all the messages. Using PGP keys, the application

also offers identity verification, data authentication and data integrity for all messages. Moreover, the application securely stores the private key of the user PGP key in the database by encrypting it with the PGP key passphrase. This passphrase is securely stored in the operating system's keychain. Furthermore, the server implements authentication securely verifying every user's identity (PGP key) and verifying every message against it so no impersonation can occur.

3. **Efficiency**: the system performs all the functionality prescribed without wasting any resources. System optimisations were performed wherever it was possible by, for example, parallelising operations so they were performed concurrently. The system employs a decentralised architecture which is more efficient by-design. Messages and files are transmitted directly from one user to another, not through a server. Moreover, the application sends files and images in binary by streaming them from one user to another at an optimum rate which is efficient both in terms of memory and time which increases system and network throughput significantly (see §5.3.2). The application uses React to provide an efficient GUI. The server uses WebSockets instead of HTTP pooling for greater efficiency and throughput (see §4.5.2).

4. **Reliability**: the system performs as it is supposed to. The application validates all user input and shows an appropriate error message if it is invalid. The application also verifies the message's data integrity along with its authenticity using its signature for every message received. The application uses sending and receiving queues for both receiving and sending messages to ensure that the right order of the all messages is retained (see §5.3.4). The application also implements an initiator agreement strategy to avoid a connection race condition (see §5.3.1). The server validates all messages sent from the application using a standard *message schema* to ensure no fields are missing which can potentially crash the server.

5. **Usability**: the system is easy to use. No registration is required to use the system. The setup only involves one step which is generating a PGP key by entering the user's name and a passphrase or importing an existing PGP key. No in-person verification is required for security unlike with existing solutions. The GUI is not only a modern chat GUI which users are already familiar with but is also much more minimal and simple than them. The application is very responsive instantly responding to user interactions (showing appropriate response messages like an error message for invalid input) and state

changes (a green circle is shown next to the chats where the user is online). Furthermore, the application offers a Dark Mode for better accessibility allowing the user to operate the system for a longer time under various different lighting conditions without straining their eyes (see §5.4.6)

7.1.2 Functional testing

Functional testing was performed by evaluating how the system produced performs against our functional requirements as defined in §3.2:

1. **Provide the user with a GUI to interact with the system**: while the server does not provide a GUI, the application does provide a GUI that enables the user to interact with the entire system. The user can create chats, remove chats, see chat messages by selecting them, write messages, see messages and a lot more using the GUI as delineated in §4.6. Hence, this requirement is successfully met.

2. **Enable the user to start a conversation with another user**: the user can compose a new chat in the GUI by specifying the other user's `userId` or PGP key as delineated in §4.6.5 and shown in §B. If the provided details are valid, then a chat is established with the other user via the *discovery protocol* as defined in §4.5.1. Once a chat is established, a peer-to-peer connection is established and E2EE messaging is initiated with the other user as defined in §4.4.5. Hence, this requirement is successfully met.

3. **Enable the user to send and receive E2EE messages with all the security properties of existing systems (as defined in §2.2.1)**: as delineated in §4.4.5, the application establishes a peer-to-peer connection with every user whom there is a chat with in order to enable sending messages to and receiving messages from that user. Those messages are E2EE by the application using the Signal Protocol as delineated in §4.4.4. As discussed in §4.2, PGP public-key cryptography is coupled with the Signal Protocol to provide all the security properties of existing systems (confidentiality, forward secrecy, post-compromise security, identity verification, data authentication and data integrity). Hence, this requirement is successfully met.

4. **Enable the user to send and receive E2EE files including images**: the application sends and receives files including images by streaming them from one user to another user over a P2P Data Channel as explained in §5.3.2. The Signal Protocol is applied here just

as it is with the messages to provide E2EE. Furthermore, the file names are sent encrypted as well. Hence, this requirement is successfully met.

5. **Be secure against the man-in-the-middle attack vulnerabilities of existing systems (as delineated in §2.3)**: as explained in §4.1, the system protects against the man-in-the-middle attack (MITM) vulnerabilities of existing systems through its decentralised architecture. The system employs a Decentralised Public Key Infrastructure (DPKI) by using PGP, wherein a long-lived established user identity is represented by a PGP key. Trust in an user identity and its associated PGP key is already established decentrally through PGP's *Web of Trust* model. So a central server or an adversary cannot imitate a user identity as they cannot furnish the PGP key associated with it which only the user holds. This prevents a server or an adversary from performing a MITM which fixes the MITM vulnerabilities of existing systems. Hence, this requirement is successfully met.

6. **Be secure without requiring any in-person verifications**: as explained in §4.1 and above, the system uses PGP which represents a user identity as a long-lived PGP key. Trust in an user identity and its associated PGP key is already established decentrally through PGP's *Web of Trust* model. As the user identity is already established and long-lived, no in-person verification is required. Moreover, this PGP key is portable so can be used on multiple devices so the user identity is universal. Thus, no in-person verification is required even when the user logs into a new device. Since no in-person verification is even required, the security is not conditional any in-person verifications as it is with existing solutions. Hence, this requirement is successfully met.

7.1.3 Acceptance testing

Acceptance testing of the system was performed with some real users in a real world setting. I received the very positive feedback from my users which further reaffirmed that the system met the requirements. Here are some quotes highlighting that:

- *"The user interface is super clean and easy to use. I love it!"*
- *"The app sends my files really fast. I couldn't even tell they were encrypted."*
- *"This app has the easiest signup process of any messaging app I've ever used."*
- *"I really like that I don't have to provide my number to use it."*

- *“Everything just works with this app and it doesn’t even have to sell my data to make it happen.”*

7.2 Project Evaluation

As evident by the previous section, the system not only fulfils all the requirements but even exceeds them in some cases. Hence we have accomplished our aim and objectives set out in §1.1 for the project. However, this does not imply that application is impeccable as there are still great areas of improvement. These will be discussed in the next chapter.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The Signal Protocol is the state-of-the-art end-to-end encryption being used by billions of people to communicate privately and securely. However, this security is conditional on in-person verifications, an attempt by the protocol to provide some protection against its man-in-the-middle attack security vulnerabilities but comes at an expense of usability that renders it as ineffective. Fixing these security vulnerabilities using the decentralised architecture we have implemented in our system can ensure that those communications are unconditionally secure without no expense in usability. Moreover, our decentralised architecture provides strong user privacy and is more efficient which existing solutions can leverage.

The application of a decentralised architecture such as ours in other areas such as market economy and multimedia provides seemingly endless possibilities for innovation and advancement. As we have already seen in the area of market economy, cryptocurrency which employs a decentralised public key infrastructure like we do for currency ownership and uses it to offer strong privacy and security. We expect more such usage to emerge as privacy and security become increasingly greater concerns.

Many key points have been learned during the as a consequence of engaging in the project work. The use of *Streams* in *Node.js* provides a way to transfer and transform large amounts of data that is efficient in both memory and time that increase system throughput profoundly. In terms of memory, the data is buffered in memory so less of the memory is used. It is time efficient as data is processed as soon as it is available, rather than waiting till the whole data payload is available to start. Moreover, Streams provide an elegant way to solve the *backpressure* problem that occurs during data handling by simply allowing the rate of the Stream flow to be adjusted.

8.2 Future Work

While we have successfully met all the requirements we set out for the project, there still much room for improvements. Possible improvements include:

- **Offline messaging**: currently, due to the decentralised nature of the application, offline messaging is not supported. While it is a quite difficult task to accomplish with a decentralised architecture, it can still be accomplished by implementing a strategy to fairly and efficiently store the messages in the entire p2p network of users on some of their devices until the intended recipient of the message comes online. This would require the users to connect to each other arbitrarily to form an overlay p2p network. A sort of a *gossip protocol* may be used to disperse the offline messages among the network.
- **Full decentralisation**: While most of the communication on the system is fully decentralised, the discovery protocol and signalling protocol still occur over a central server as it is fairly challenging to decentralise them. However, if an overlay p2p network is created as discussed above, they could be performed using the *gossip protocol* to send the discovery/signalling messages and *distributed hash tables (DHT)* to store all the users in the network. The DHT would be stored by each user and contain all the users it knows in the network. The discovery/signalling messages are then dispersed to each user who forwards it to the intended recipient by the ones who know the intended recipient. However, this would make the protocols a lot more inefficient and slower but improve user privacy even offer anonymity for them.
- **Videos**: Allow videos to sent as a "video" file type and played back in the GUI.
- **Voice calls**: Allow end-to-end encrypted voice calls using *Voice over IP (VoIP)* from one user to another directly.
- **Video calls**: Allow end-to-end encrypted video calls using *Voice over IP (VoIP)* from one user to another directly.

References

[1] Facebook's data-sharing deals exposed. *BBC News*, Dec 2018.

[2] Perrig Adrian. How pgp works. *Carnegie Mellon University*, Jan 1999.

[3] J Clement. Most popular global mobile messenger apps as of october 2019, based on number of monthly active users. *Statista*, Nov 2019.

[4] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, Jul 2017.

[5] Douglas Crawford. Signal review, Jun 2018.

[6] Matthew Green. What's the matter with PGP?, Aug 2014.

[7] John Harris. The cambridge analytica saga is a scandal of facebook's own making — john harris. *The Guardian*, Mar 2018.

[8] Issie Lapowsky. Cambridge analytica could also access private facebook messages. *Wired*, Apr 2018.

[9] Issie Lapowsky. How cambridge analytica sparked the great privacy awakening. *Wired*, Mar 2019.

[10] Svetlin Mladenov. Webrtc data channel: Optimum message size. *Viblast*, Feb 2015.

[11] Nicole Perlroth. What is end-to-end encryption? another bull's-eye on big tech. *The New York Times*, Nov 2019.

[12] Sebastian Schrittwieser, Peter Fruehwirt, Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Markus Huber, and Edgar Weippl. Guess who is texting you? evaluating the security of smartphone messaging applications, Jan 2012.

[13] David Shaw, Lutz Donnerhacke, Rodney Thayer, Hal Finney, and Jon Callas. Openpgp message format. *Internet Engineering Task Force (IETF)*, Nov 2007.

[14] Open Whisper Systems. Facebook messenger deploys signal protocol for end-to-end encryption. *Signal Blog*, Jul 2016.

[15] Open Whisper Systems. Signal protocol specification. *Signal*, Nov 2016.

[16] Open Whisper Systems. Whatsapp's signal protocol integration is now complete. *Signal Blog*, Apr 2016.

[17] WhatsApp. Whatsapp security whitepaper. *WhatsApp*, Dec 2017.

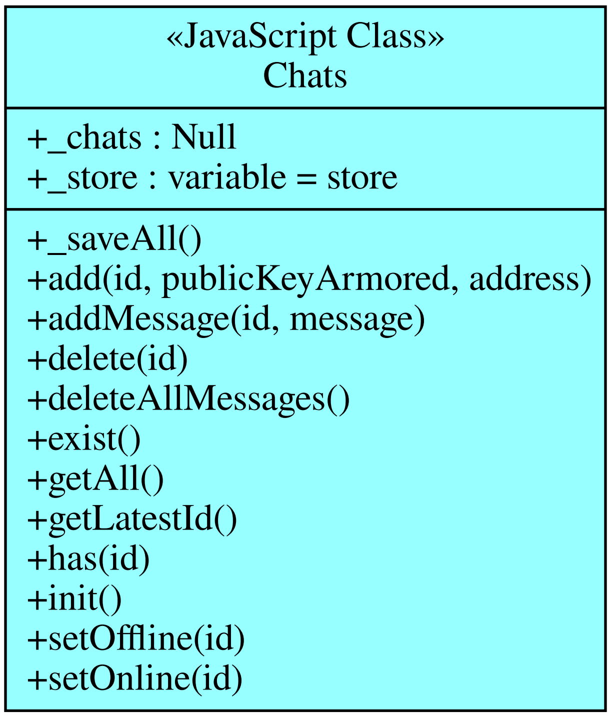
[18] Ben Wolford. What is PGP encryption and how does it work? *ProtonMail Blog*, Aug 2019.

Appendix A

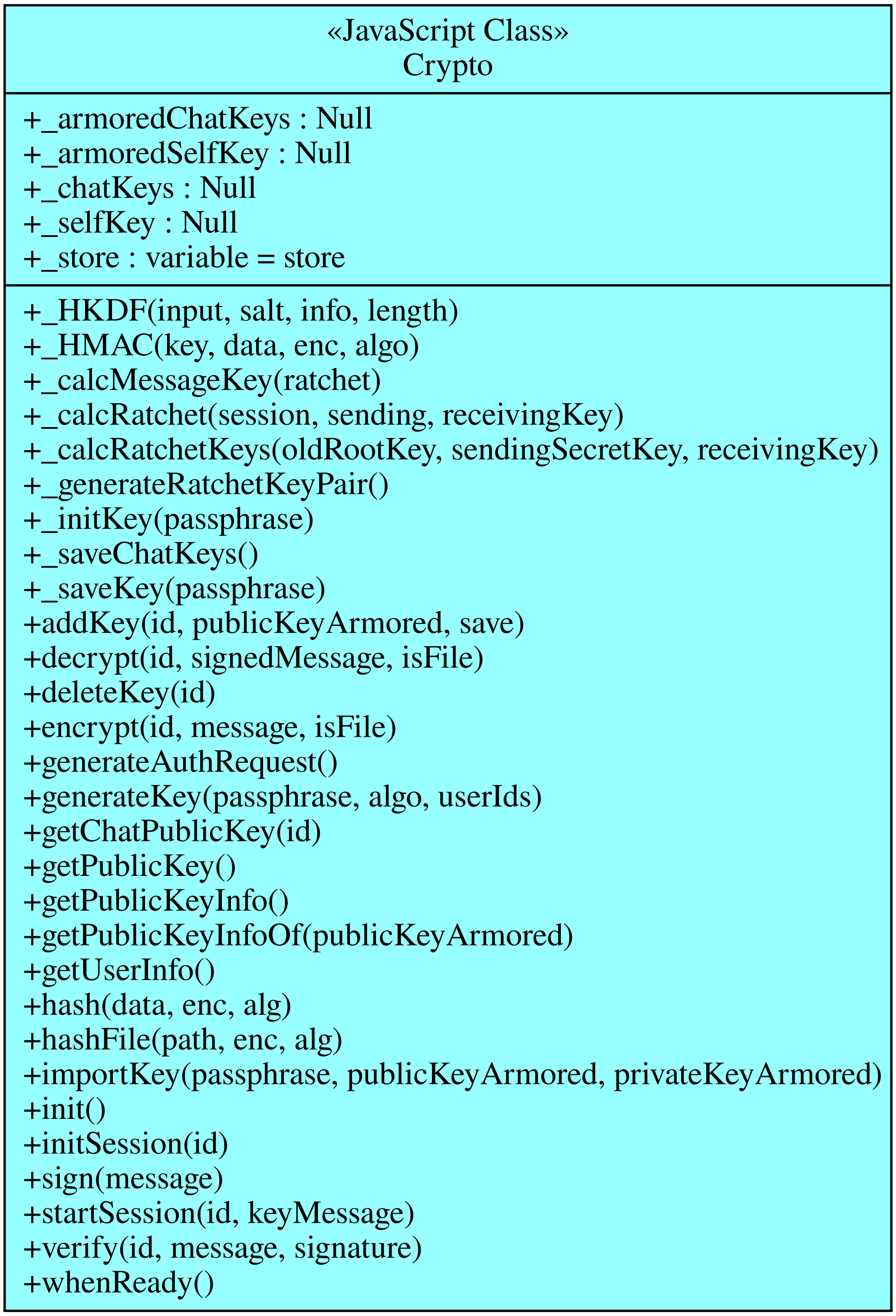
Extra Information

A.1 Class diagrams

A.1.1 Chats Class



A.1.2 Crypto Class

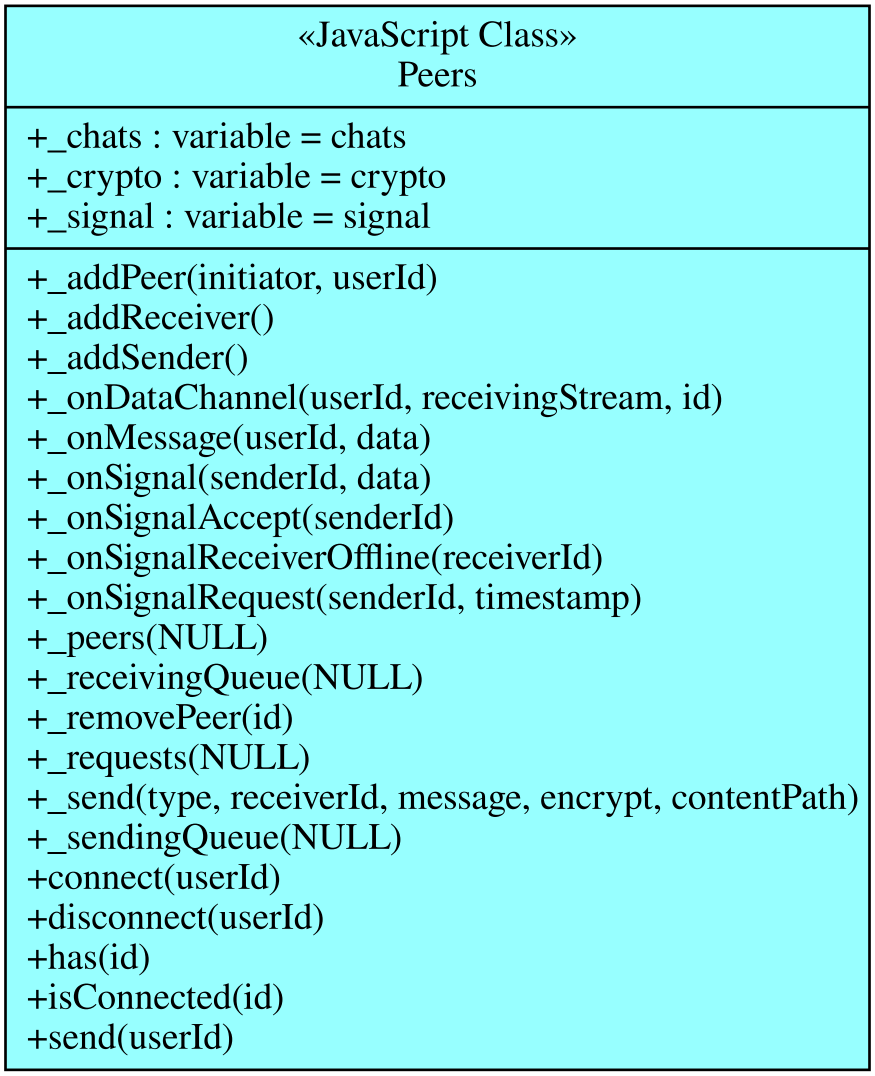


«JavaScript Class»
Crypto

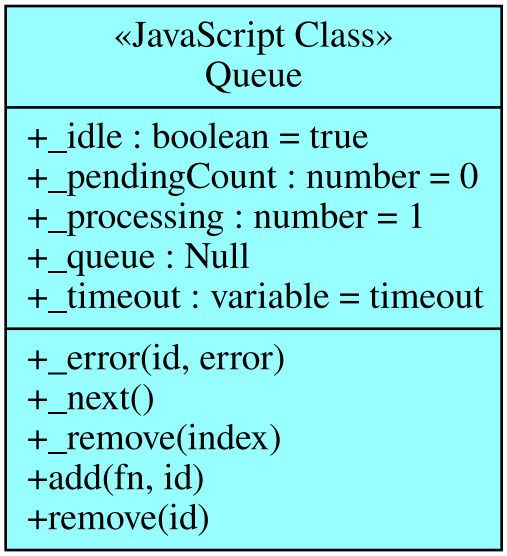
+_armoredChatKeys : Null
+_armoredSelfKey : Null
+_chatKeys : Null
+_selfKey : Null
+_store : variable = store

+_HKDF(input, salt, info, length)
+_HMAC(key, data, enc, algo)
+_calcMessageKey(ratchet)
+_calcRatchet(session, sending, receivingKey)
+_calcRatchetKeys(oldRootKey, sendingSecretKey, receivingKey)
+_generateRatchetKeyPair()
+_initKey(passphrase)
+_saveChatKeys()
+_saveKey(passphrase)
+addKey(id, publicKeyArmored, save)
+decrypt(id, signedMessage, isFile)
+deleteKey(id)
+encrypt(id, message, isFile)
+generateAuthRequest()
+generateKey(passphrase, algo, userIds)
+getChatPublicKey(id)
+getPublicKey()
+getPublicKeyInfo()
+getPublicKeyInfoOf(publicKeyArmored)
+getUserInfo()
+hash(data, enc, alg)
+hashFile(path, enc, alg)
+importKey(passphrase, publicKeyArmored, privateKeyArmored)
+init()
+initSession(id)
+sign(message)
+startSession(id, keyMessage)
+verify(id, message, signature)
+whenReady()

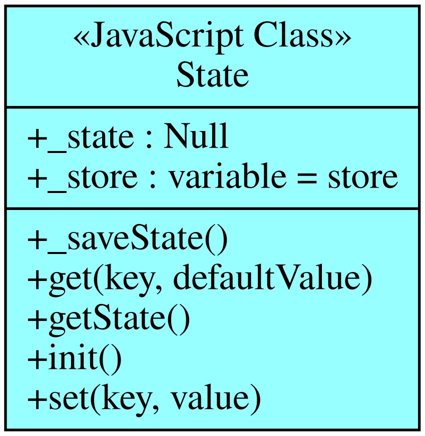
A.1.3 Peers Class


«JavaScript Class»
Peers
+_chats : variable = chats
+_crypto : variable = crypto
+_signal : variable = signal
+_addPeer(initiator, userId)
+_addReceiver()
+_addSender()
+_onDataChannel(userId, receivingStream, id)
+_onMessage(userId, data)
+_onSignal(senderId, data)
+_onSignalAccept(senderId)
+_onSignalReceiverOffline(receiverId)
+_onSignalRequest(senderId, timestamp)
+_peers(NULL)
+_receivingQueue(NULL)
+_removePeer(id)
+_requests(NULL)
+_send(type, receiverId, message, encrypt, contentPath)
+_sendingQueue(NULL)
+connect(userId)
+disconnect(userId)
+has(id)
+isConnected(id)
+send(userId)

A.1.4 Queue Class



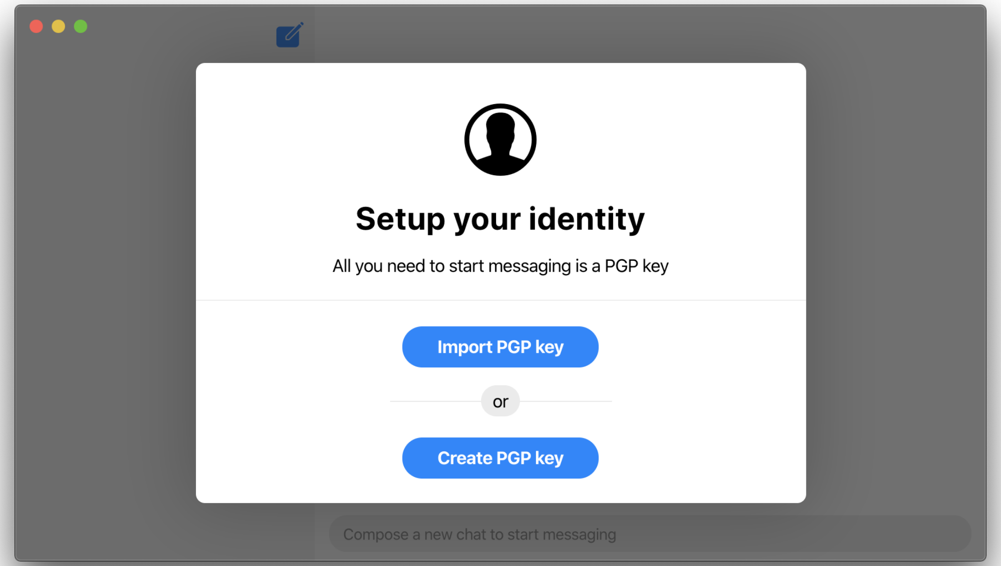
A.1.5 State Class



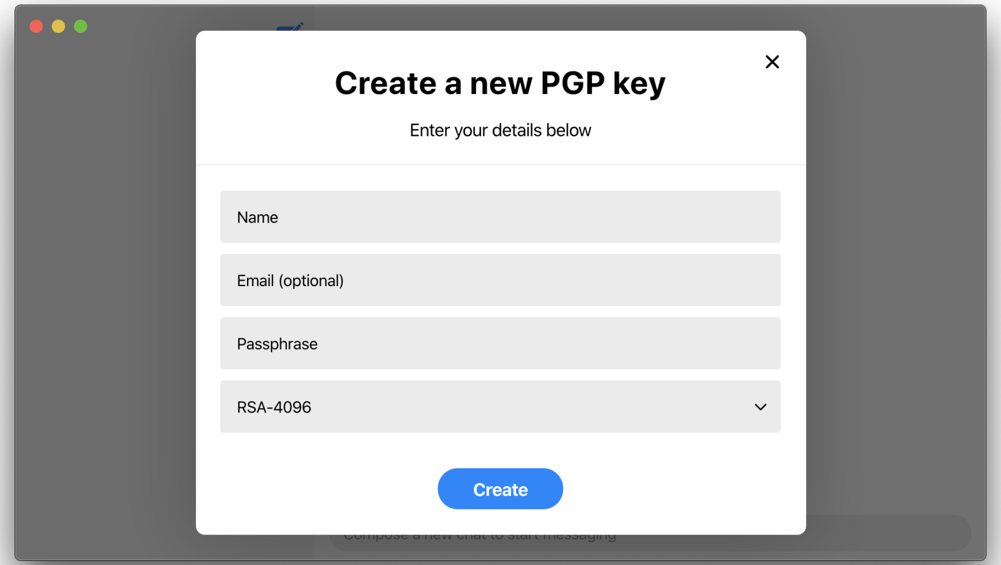
A.2 Graphical User Interface

A.2.1 Standard

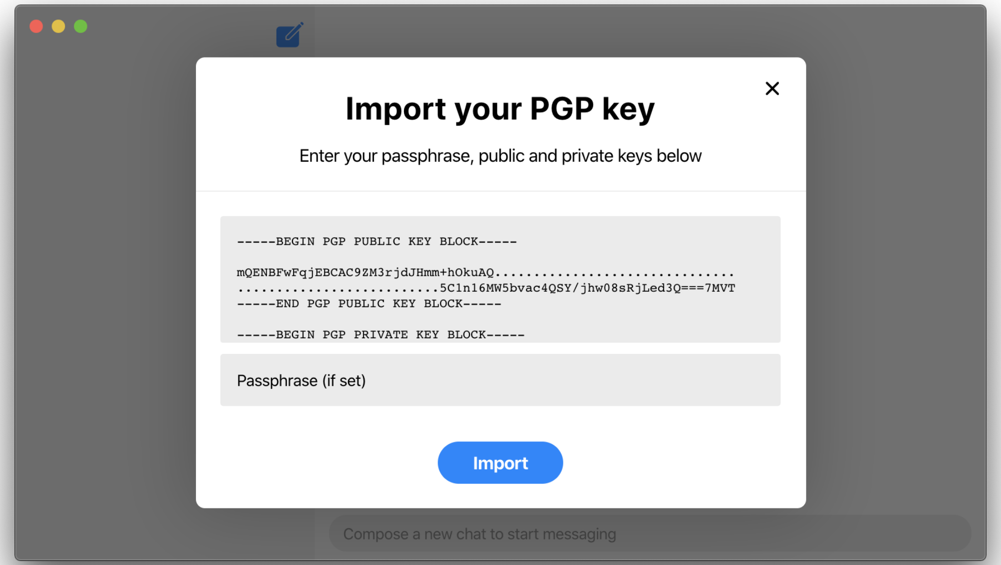
Setup Identity


Setup your identity
All you need to start messaging is a PGP key
Import PGP key
or
Create PGP key
Compose a new chat to start messaging

Create Identity screen



Import Identity screen


Import your PGP key

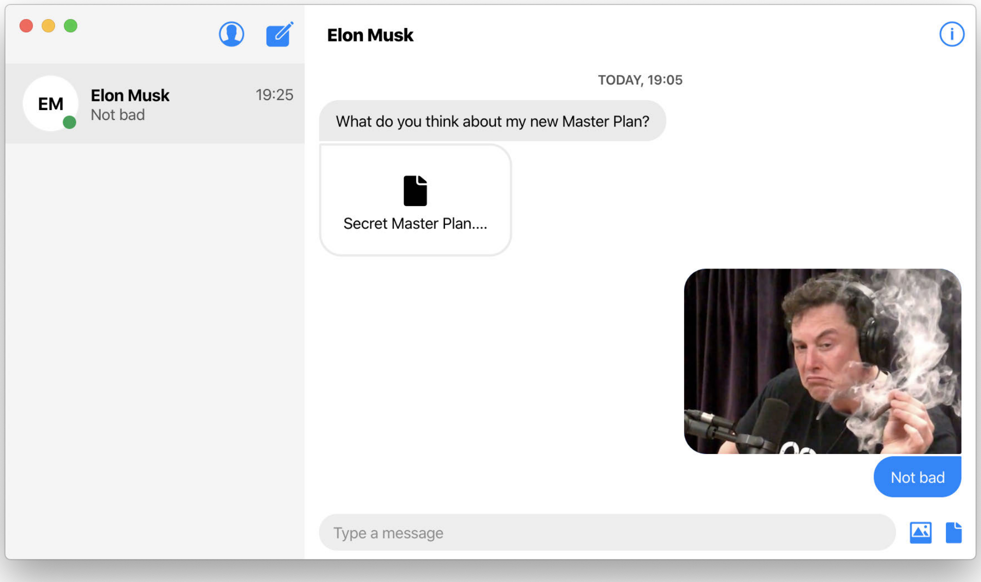
Enter your passphrase, public and private keys below

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBFwFqjEBCAC9ZM3rjdJHmm+hOkuAQ................................
..........................5C1n16MW5bvac4QSY/jhw08sRjLed3Q===7MVT
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP PRIVATE KEY BLOCK-----
```

Messenger screen


Elon Musk

EM Elon Musk 19:25
Not bad

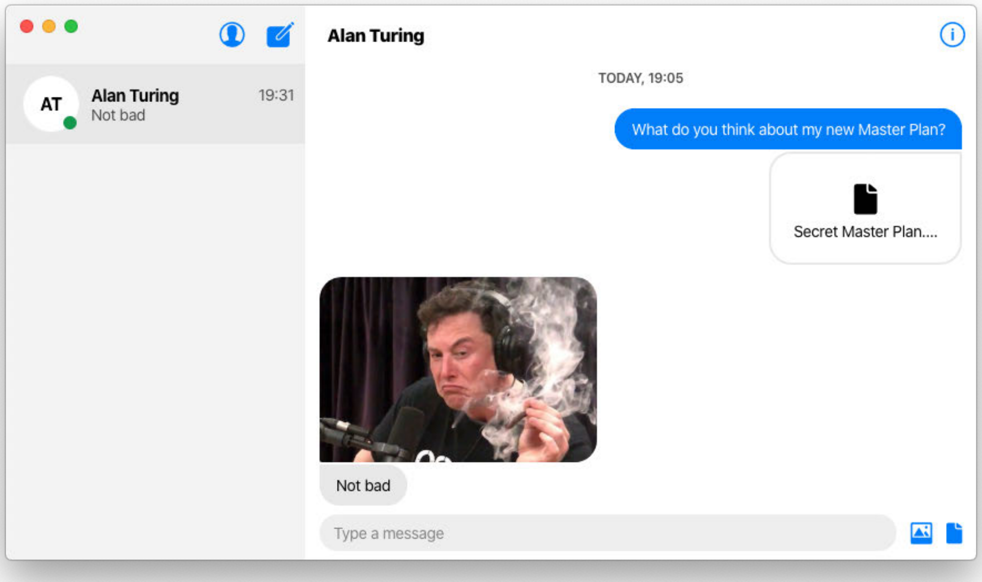
TODAY, 19:05

What do you think about my new Master Plan?

Secret Master Plan....

Not bad

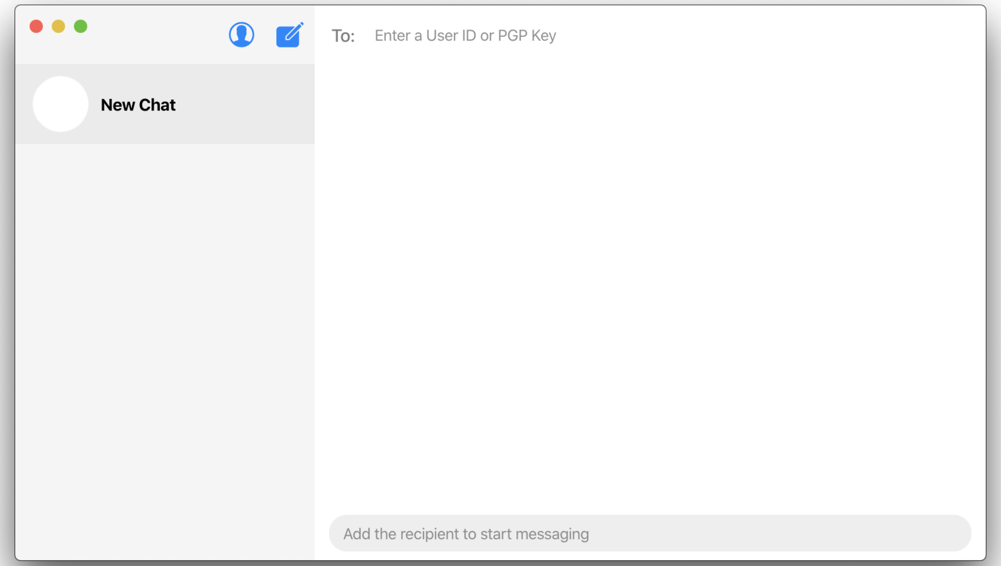
Messenger screen (receiving end)



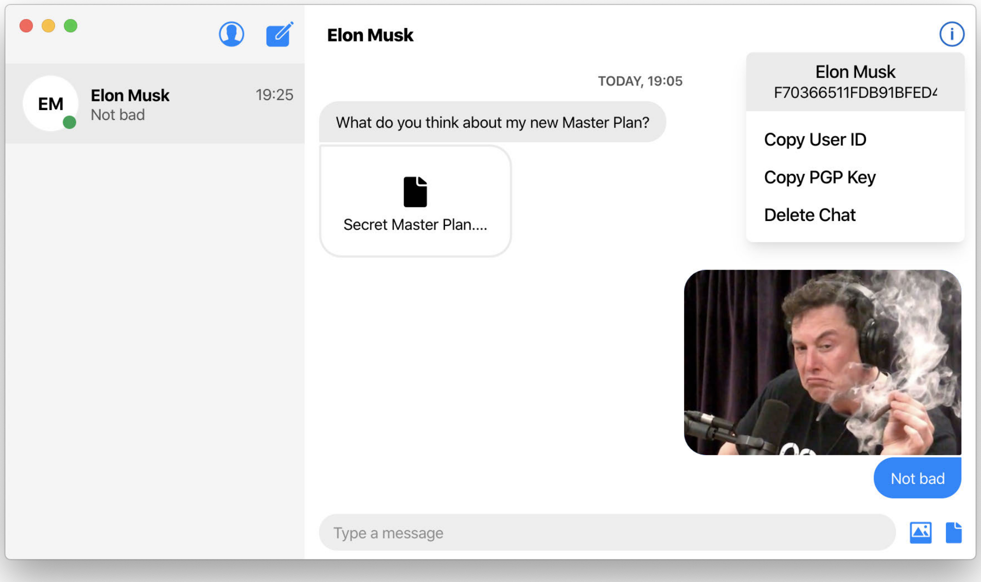
What do you think about my new Master Plan?

Not bad

Compose chat screen


New Chat
To:
Enter a User ID or PGP Key
Add the recipient to start messaging

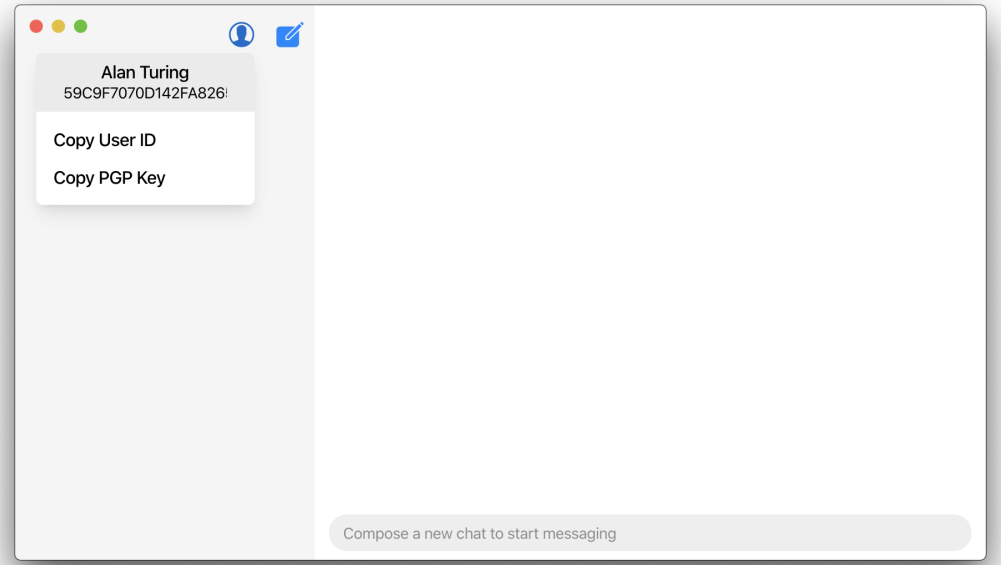
Chat info screen



What do you think about my new Master Plan?

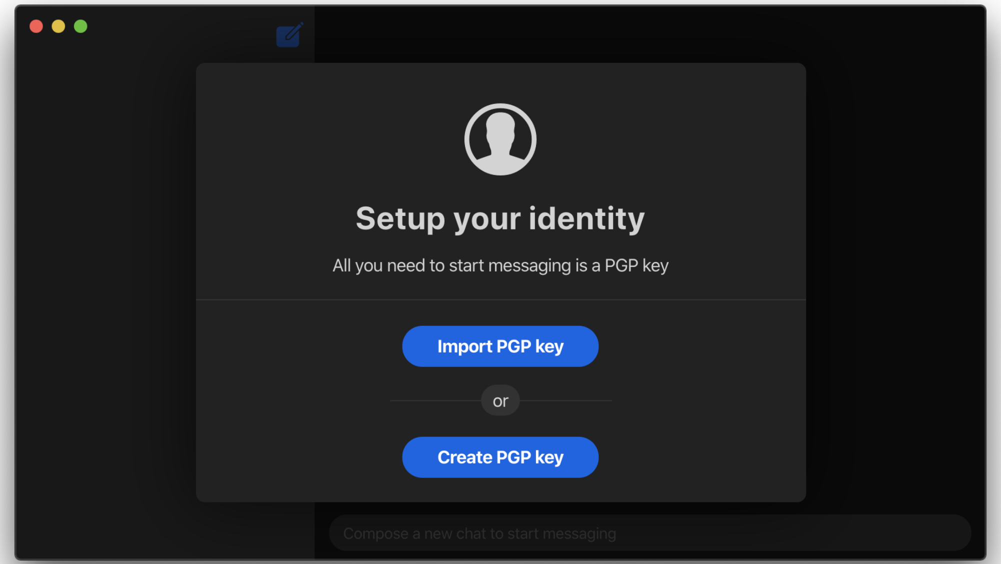
Not bad

User info screen

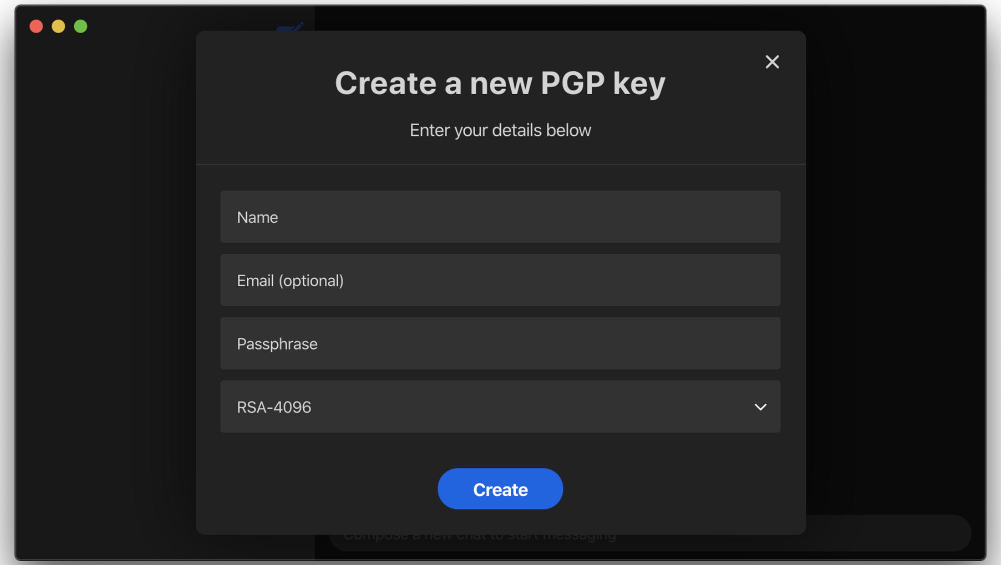


A.2.2 Dark Mode

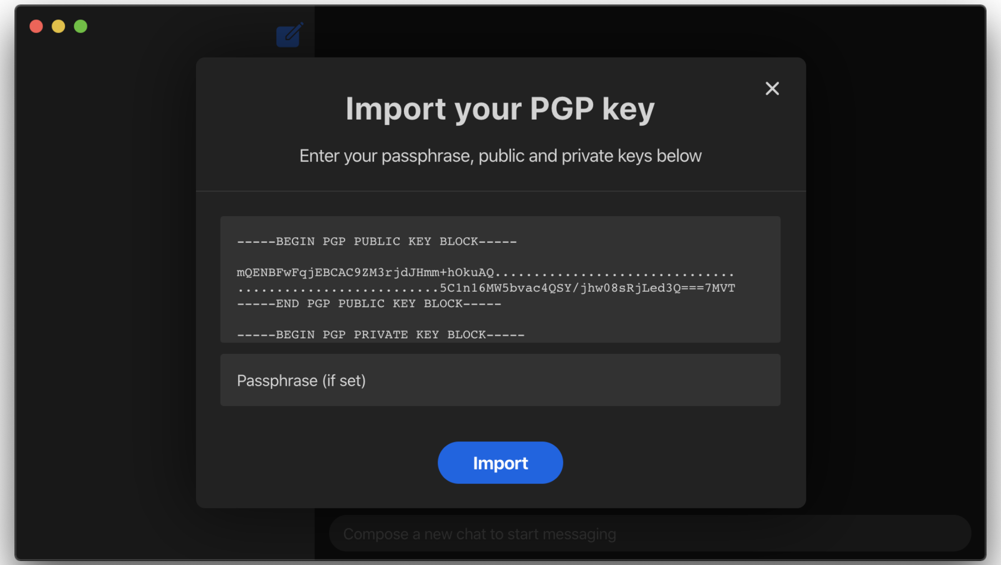
Setup Identity


Setup your identity
All you need to start messaging is a PGP key
Import PGP key
or
Create PGP key
Compose a new chat to start messaging

Create Identity screen



Import Identity screen



Import your PGP key

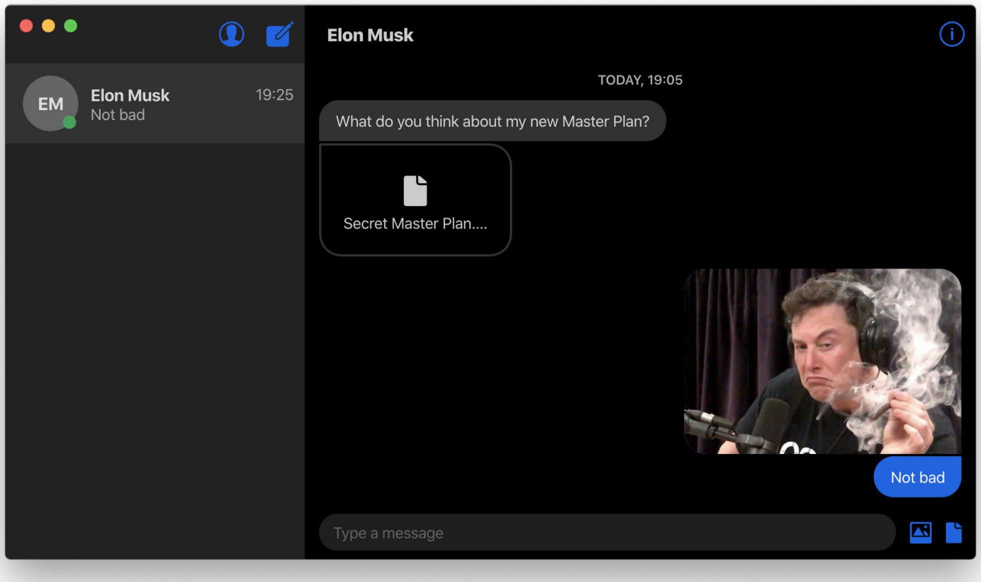
Enter your passphrase, public and private keys below

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBFwFqjEBCAC9ZM3rjdJHmm+hOkuAQ................................
..........................5C1n16MW5bvac4QSY/jhw08sRjLed3Q===7MVT
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP PRIVATE KEY BLOCK-----
```

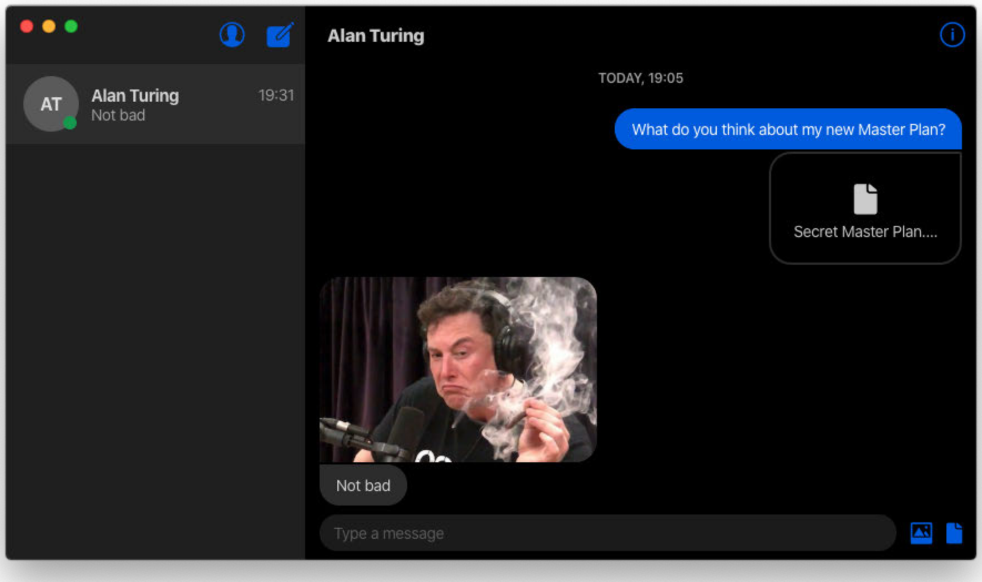
Messenger screen



What do you think about my new Master Plan?

Not bad

Messenger screen (receiving end)

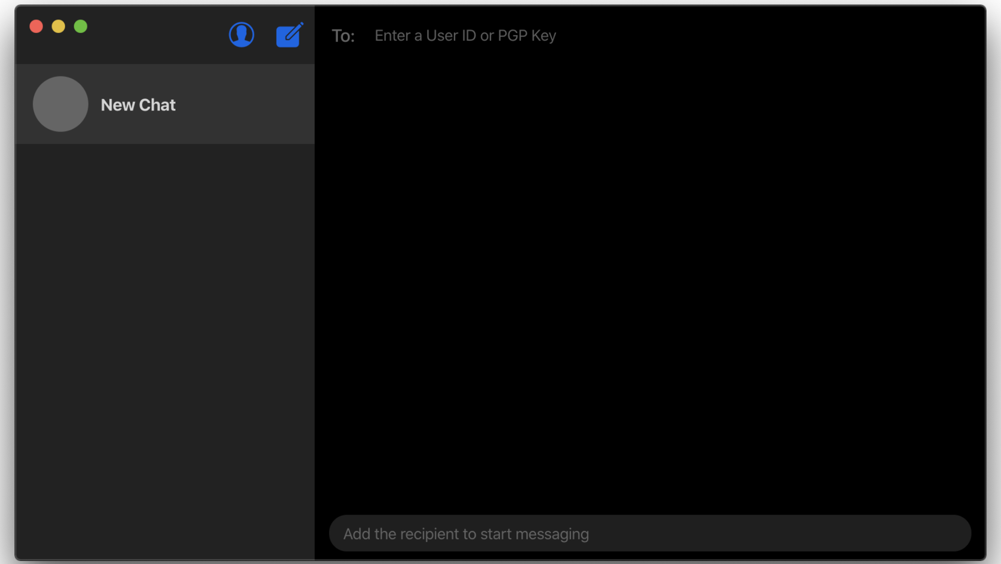


What do you think about my new Master Plan?

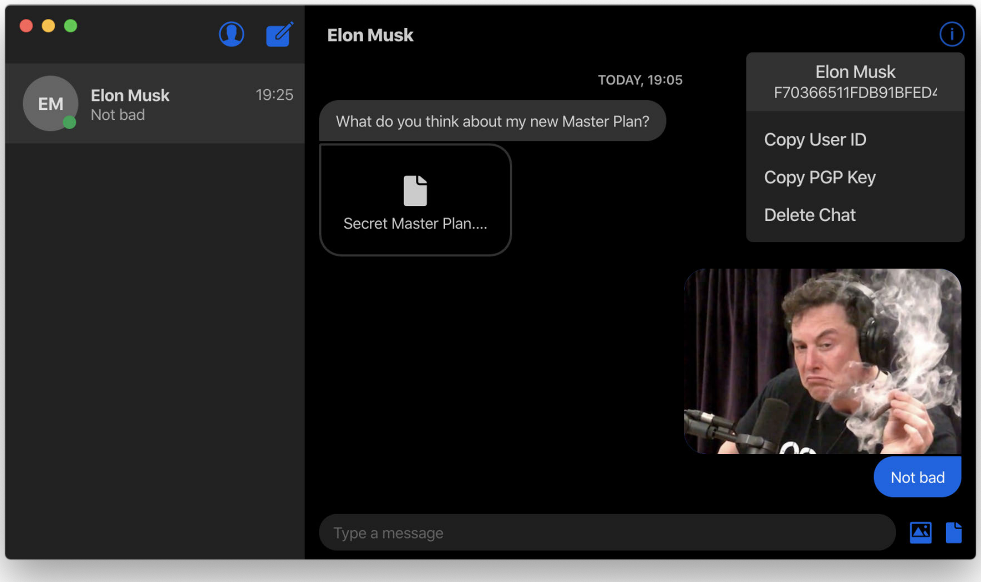
Secret Master Plan....

Not bad

Compose chat screen


New Chat
To:
Enter a User ID or PGP Key
Add the recipient to start messaging

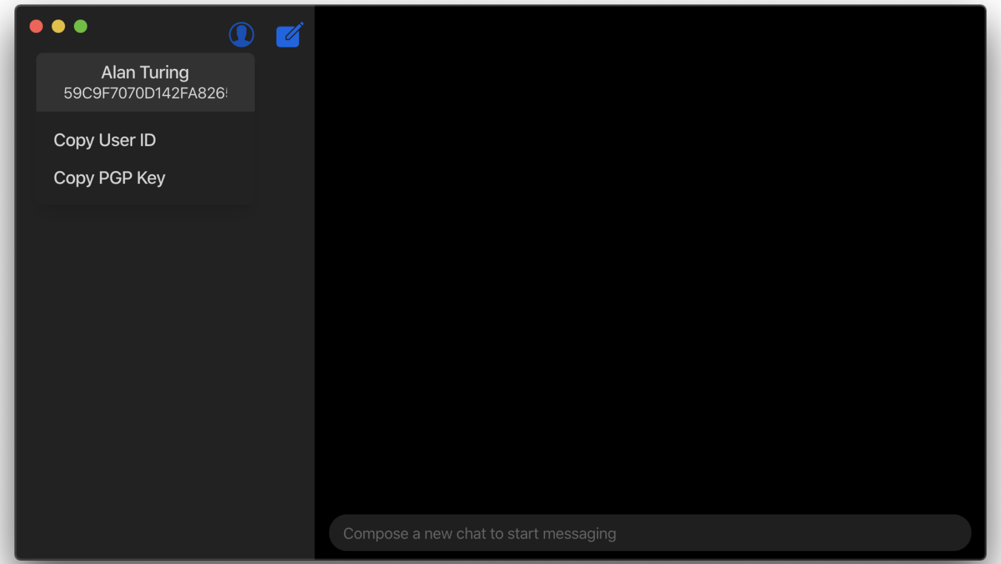
Chat info screen



What do you think about my new Master Plan?

Not bad

User info screen

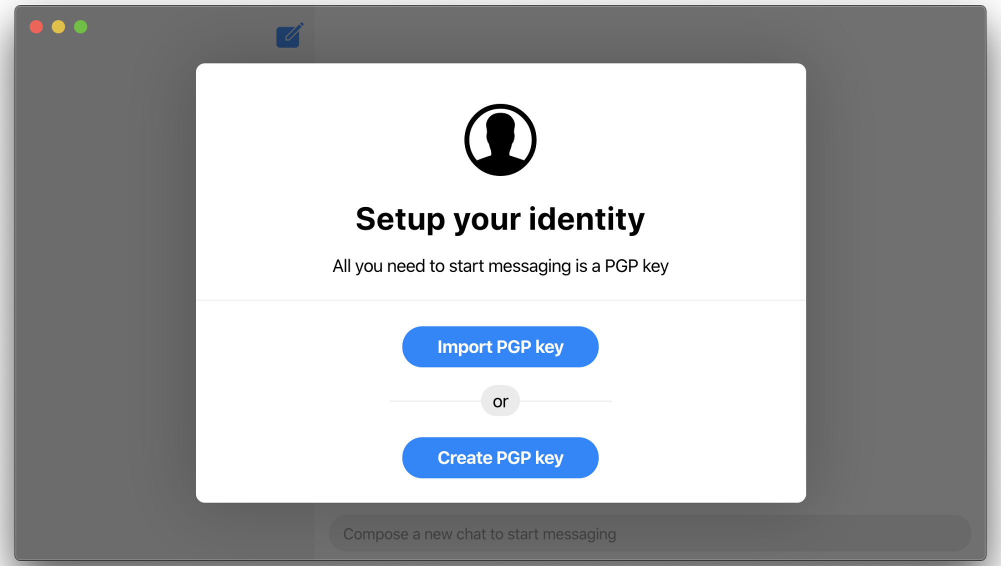


Appendix B

User Guide

B.1 Setting up your identity

When you first launch the application after installation, you will see the following screen and be asked to setup an identity which is represented by a PGP key in order to use the application.

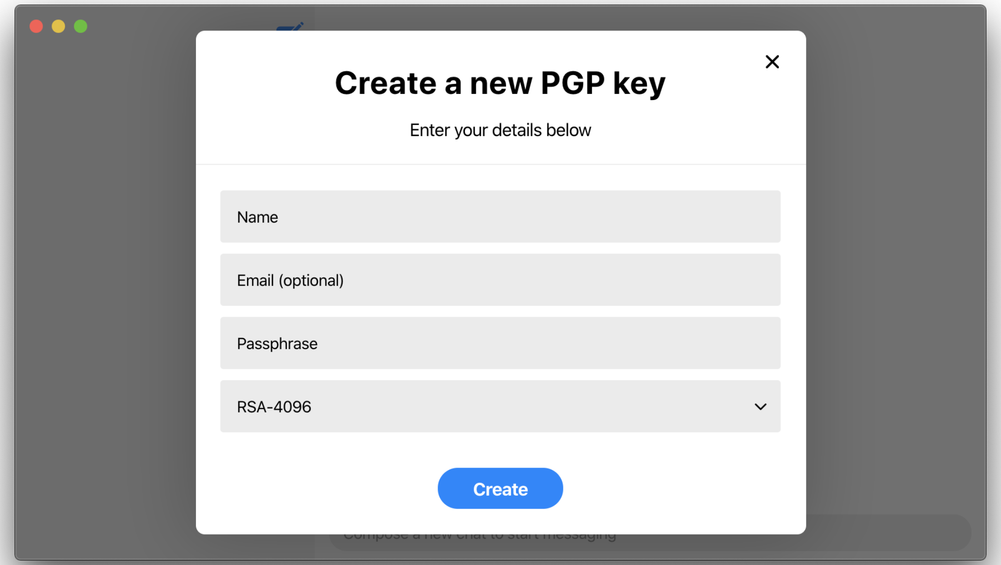

Setup your identity
All you need to start messaging is a PGP key
Import PGP key
or
Create PGP key
Compose a new chat to start messaging

If you already have a PGP key that you would like to use for the application then click the “Import PGP key” button and follow the instructions in §B.1.2.

Otherwise, click the “Create PGP key” button and follow the instructions in §B.1.1.

B.1.1 Creating your identity

Upon clicking the “Create PGP key” button in the setup identity screen, you will be presented with the following screen to create your identity.


Create a new PGP key

Enter your details below

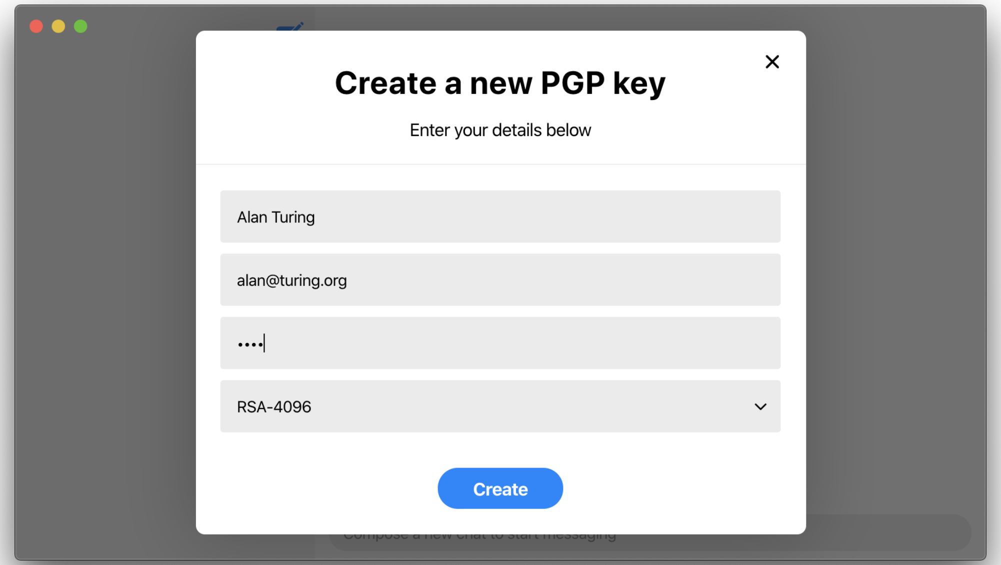
Name

Email (optional)

Passphrase

RSA-4096

Enter all your details like, for example, so:


Create a new PGP key

Enter your details below

Alan Turing

alan@turing.org

••••

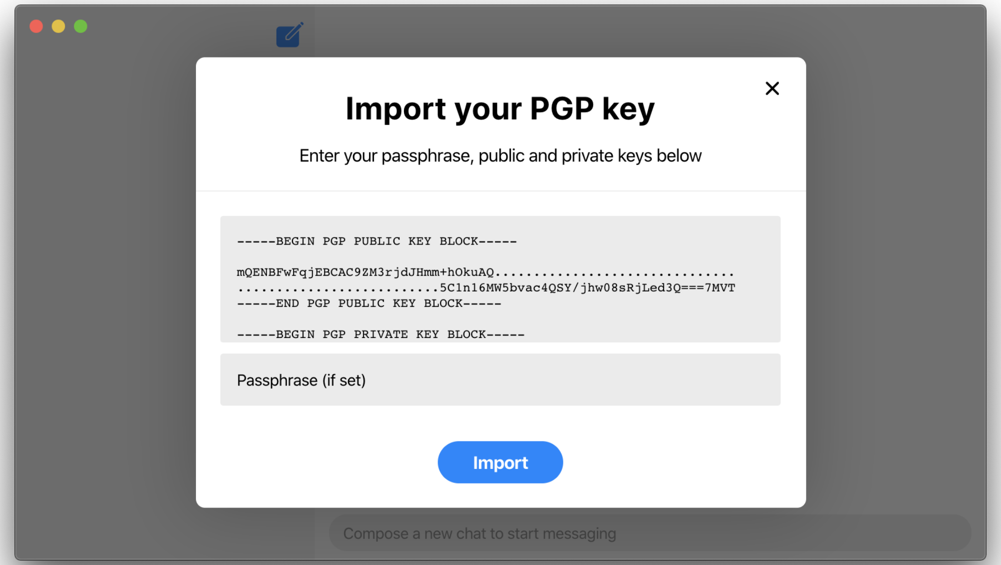
RSA-4096

If the given details are invalid, you should see an error message.

Make sure to select a strong *passphrase* as it acts as a password to protect your identity. Then click the “Create” button to finish the setup.

B.1.2 Importing your identity

Upon clicking the “Import PGP key” button in the setup identity screen, you will be presented with the following screen to import your identity.



Import your PGP key

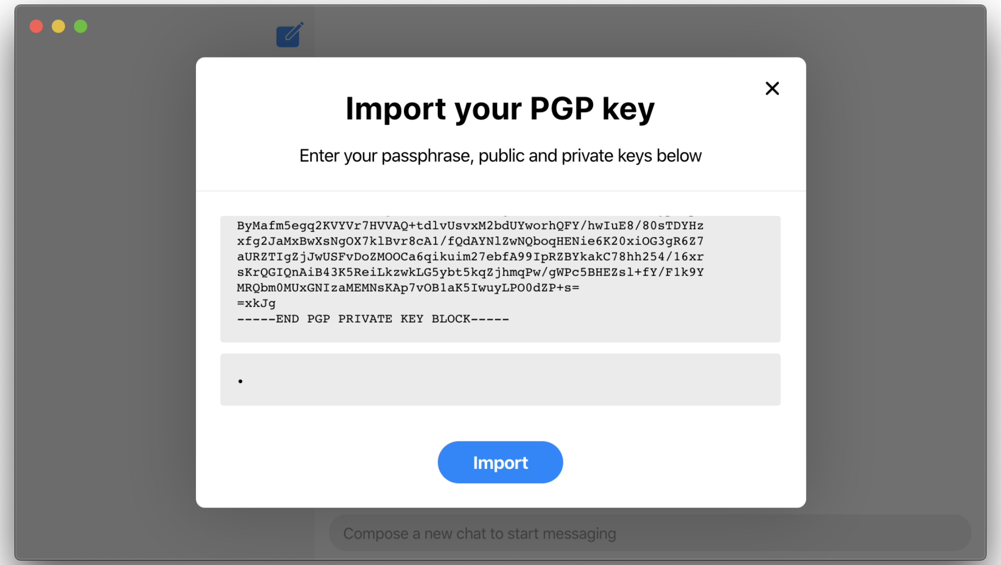
Enter your passphrase, public and private keys below

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQENBFwFqjEBCAC9ZM3rjdJHmm+hOkuAQ................................
..........................5C1n16MW5bvac4QSY/jhw08sRjLed3Q===7MVT
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP PRIVATE KEY BLOCK-----
```

Enter all your PGP key and its passphrase like, for example, so:


Import your PGP key
Enter your passphrase, public and private keys below
Import
Compose a new chat to start messaging

```
ByMafm5egq2KVYVr7HVVAQ+tdlvUsvxM2bdUYworhQFY/hwIuE8/80sTDYHz
xfg2JaMxBwXsNgOX7klBvr8cA1/fQdAYNlZwNQboqHENie6K20xiOG3gR6Z7
aURZTIgZjJwUSFvDoZMOOCa6qikuim27ebfA99IpRZBYkakC78hh254/16xr
sKrQGIQnAiB43K5ReiLkzwkLG5ybt5kqZjhmqPw/gWPc5BHEZsl+fY/F1k9Y
MRQbm0MUxGNIzaMEMNsKAp7vOB1aK5IwuyLPO0dZP+s=
=xkJg
-----END PGP PRIVATE KEY BLOCK-----
```

•

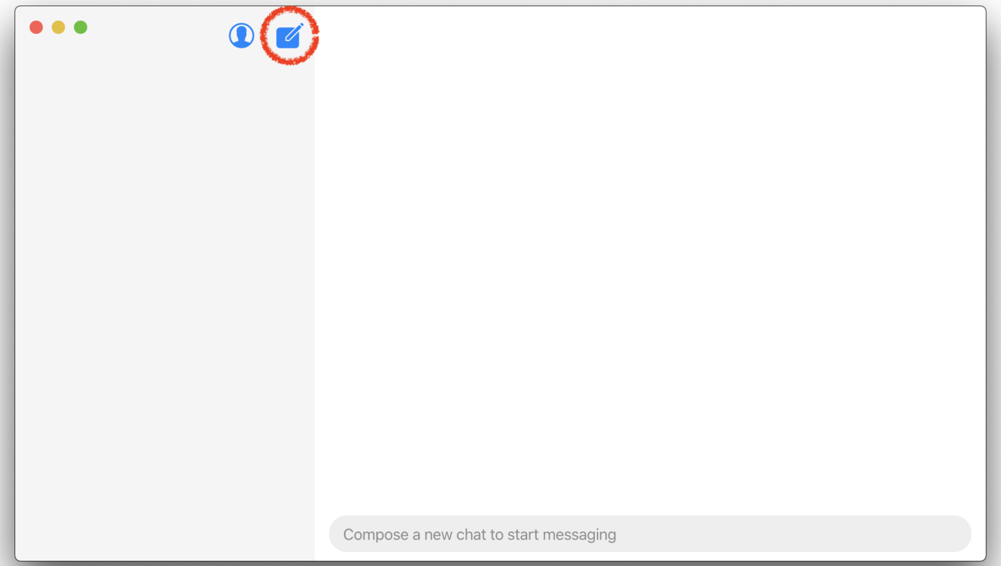
If the given details are incorrect, you should see an error message.

Then click the "Import" button to finish the setup.

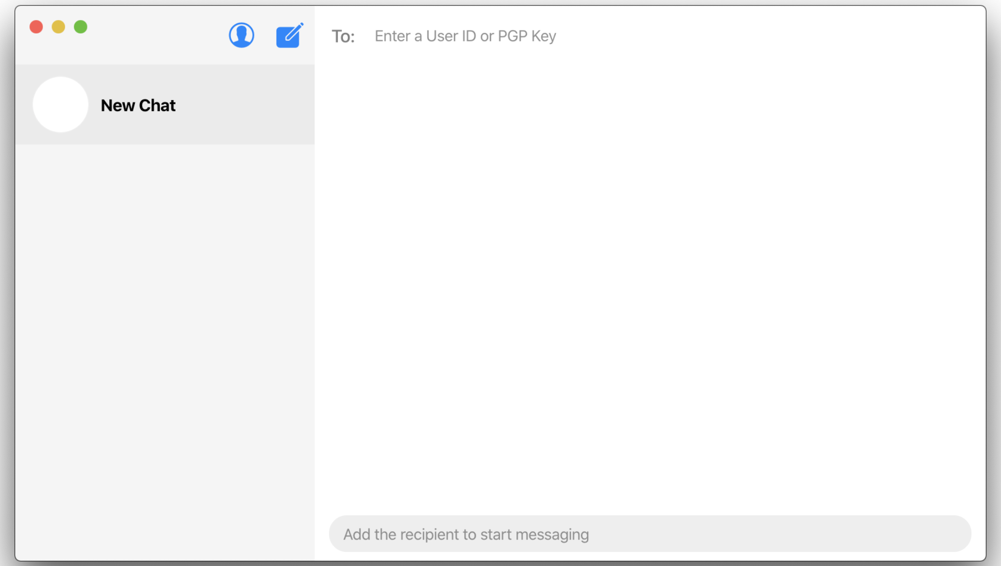
B.2 Composing a chat

In order to message someone, you need to compose a chat with them first.

You can do this by clicking the chat *compose icon* at the top right of the sidebar in the Messenger screen:



Once you do, you should see the compose chat screen with an input at the top:


To: Enter a User ID or PGP Key
New Chat
Add the recipient to start messaging

Enter or paste in the input the `userId` or PGP key of the user you want to chat with.

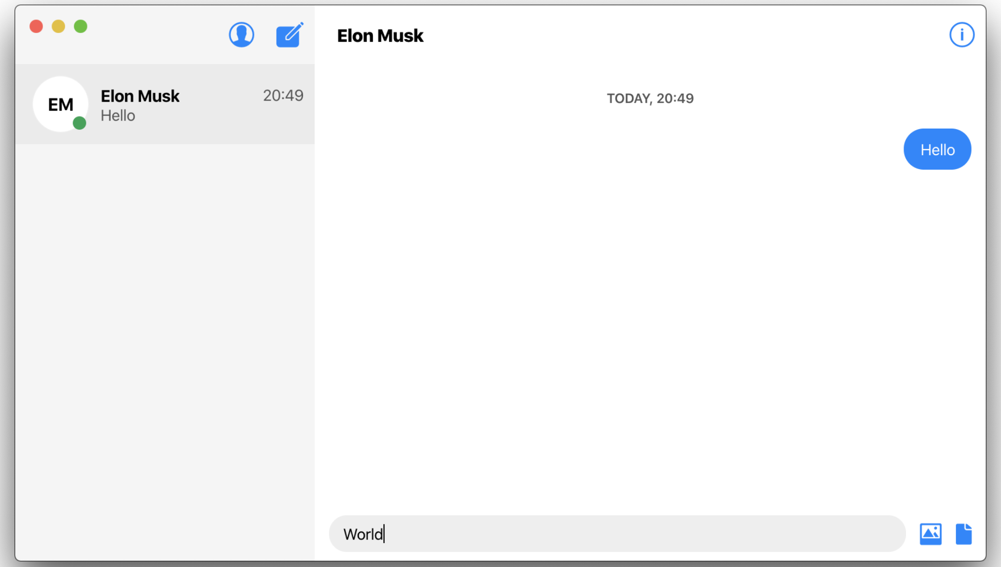
If the user does not exist or cannot be found you will se an error message.

Otherwise the chat will be created and you should be able to start messaging by selecting it.

B.3 Sending messages

B.3.1 Sending text messages

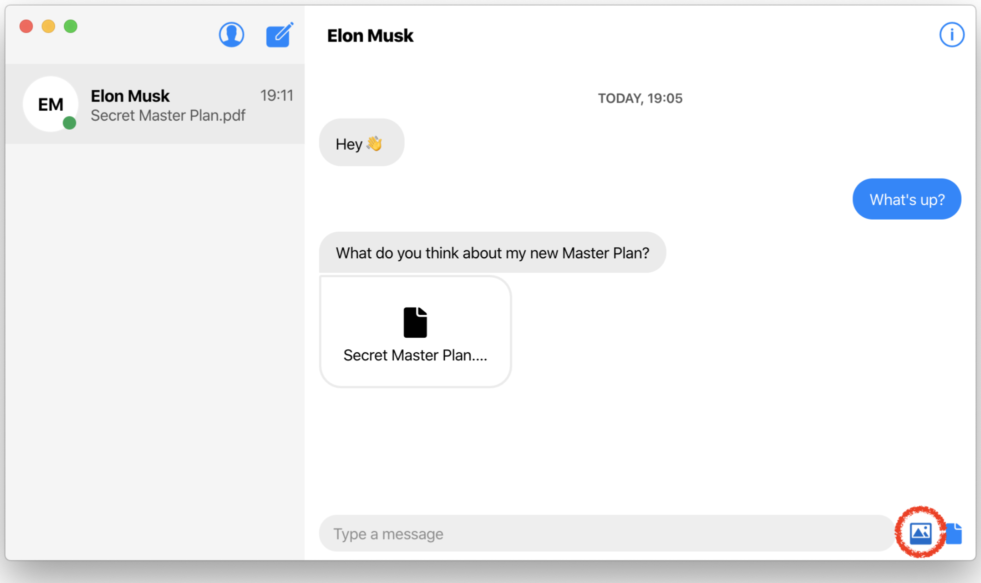
You can send text (including emoji) messages by simply selecting the desired chat and typing the message in the message bar at the bottom of the chat and pressing enter when you are done. As shown here:



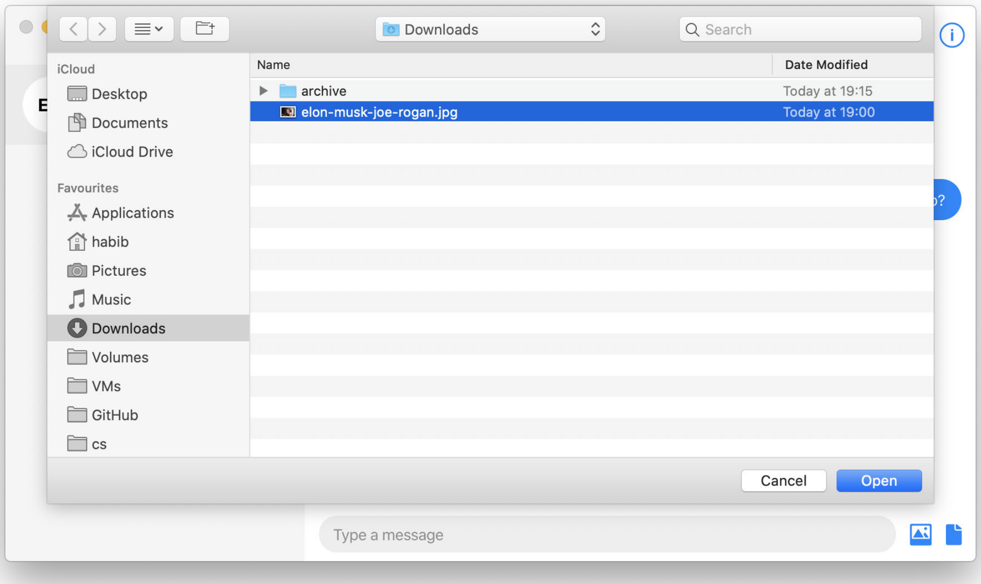
N.B. offline messaging is not currently supported so the recipient has to be online for the message to be successfully received.

B.3.2 Sending images

To send an image in a chat, first click on the *image icon* at the bottom right of the message bar:

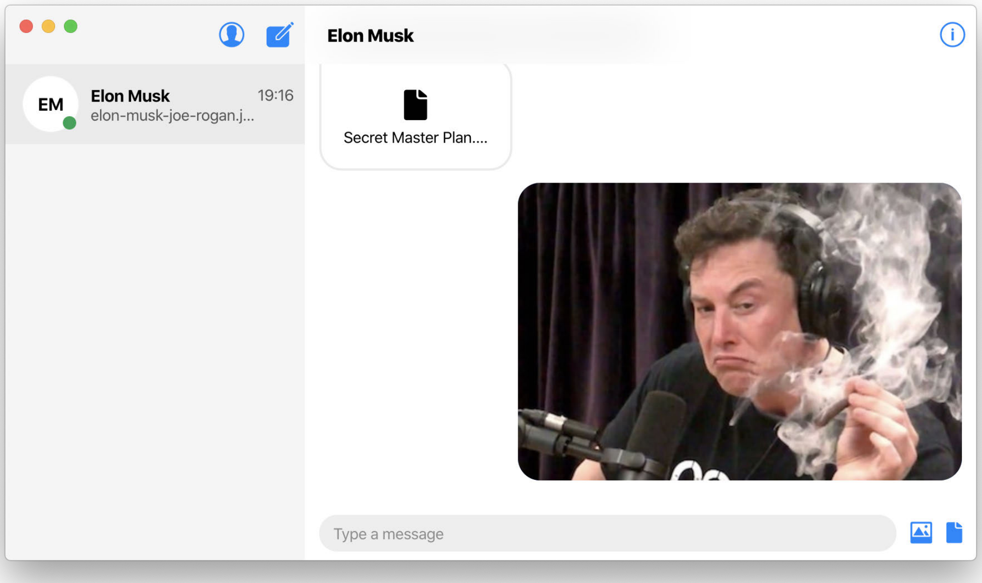


This will open a file dialog, select the image you would like to send like so:



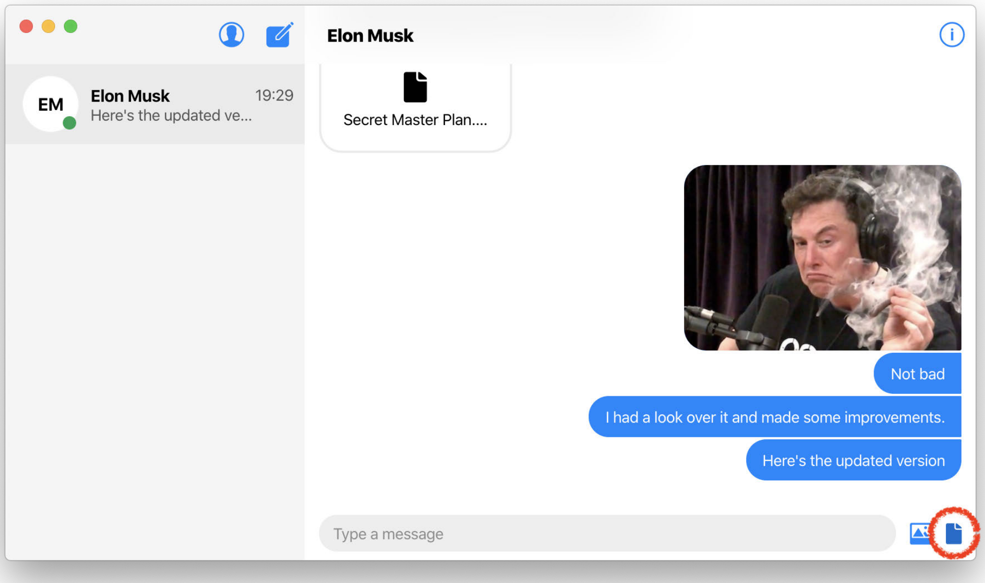
Name	Date Modified
archive	Today at 19:15
elon-musk-joe-rogan.jpg	Today at 19:00

And that's it! You should see the sent image in the chat like so:



B.3.3 Sending files

To send a file in a chat, first click on the *file icon* at the bottom right of the message bar:

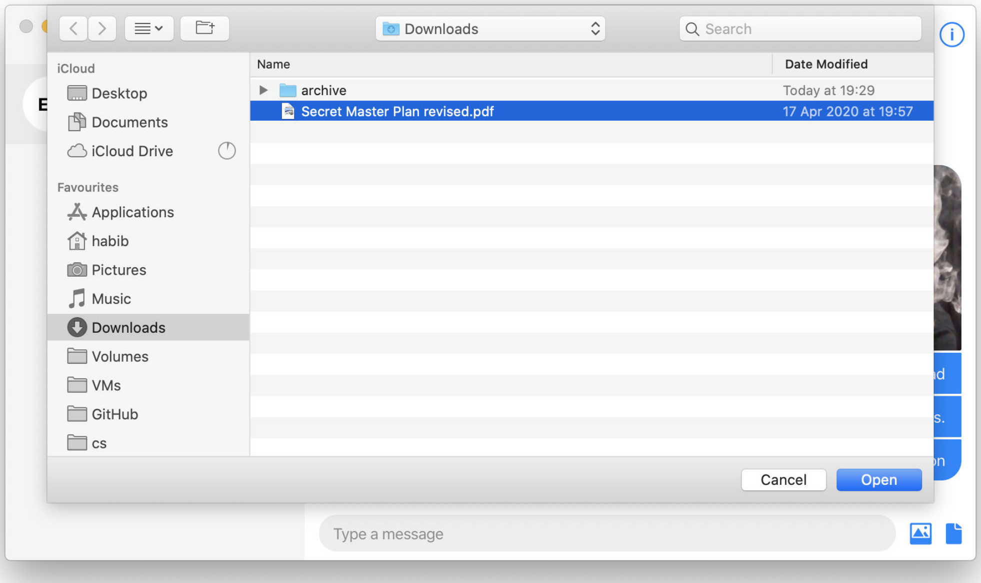


Not bad

I had a look over it and made some improvements.

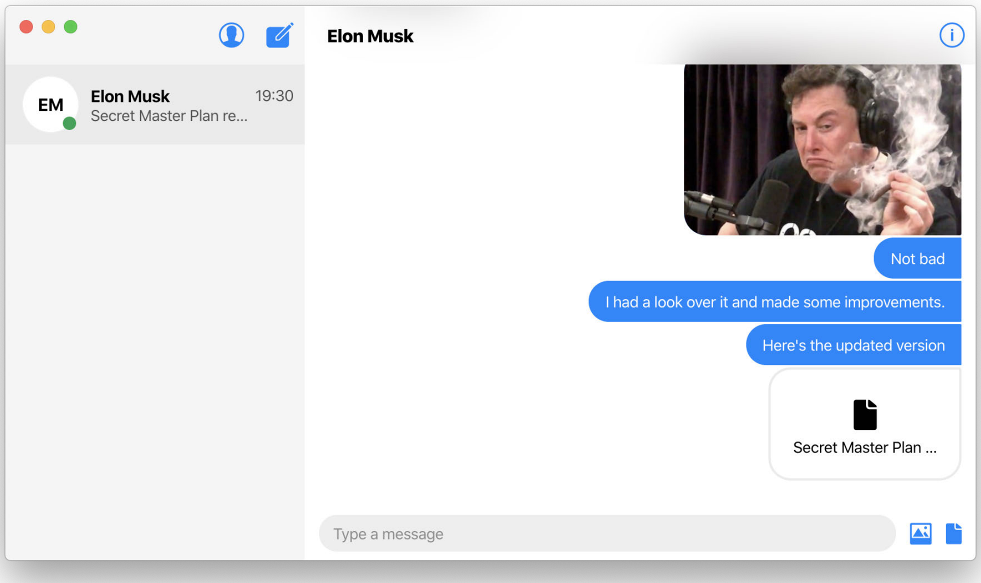
Here's the updated version

This will open a file dialog, select the file you would like to send like so:



Name	Date Modified
archive	Today at 19:29
Secret Master Plan revised.pdf	17 Apr 2020 at 19:57

And that's it! You should see the sent file in the chat like so:



Elon Musk

EM Elon Musk 19:30
Secret Master Plan re...

Not bad

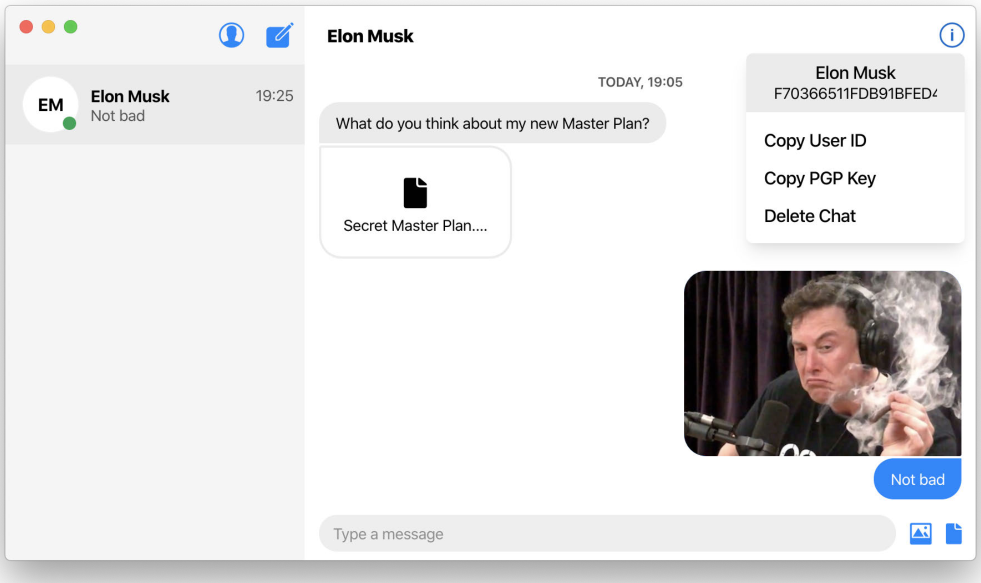
I had a look over it and made some improvements.

Here's the updated version

Secret Master Plan ...

B.4 Viewing information about a chat

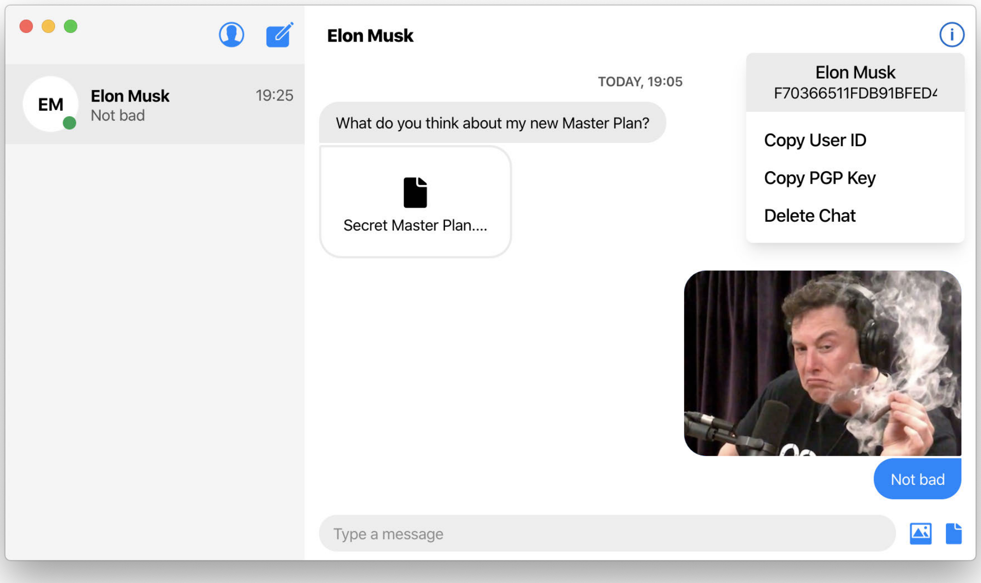
You can view the information including the full name, PGP key and userId of a user with whom you are chatting by simply selecting the chat and clicking on the *info icon* at the top right of the chat. This will open the chat info as a dropdown like so for example:



The full name and userId are shown at the top of the dropdown. You can also click on “Copy User ID” to copy and view it. You can click on “Copy PGP Key” to copy and view the PGP key of the chat user.

B.5 Deleting a chat

You can delete a chat by first selecting it, opening its chat info as instructed in §B.4 and then clicking on the “Delete Chat” option:


Elon Musk

EM

Elon Musk 19:25
Not bad

TODAY, 19:05

What do you think about my new Master Plan?

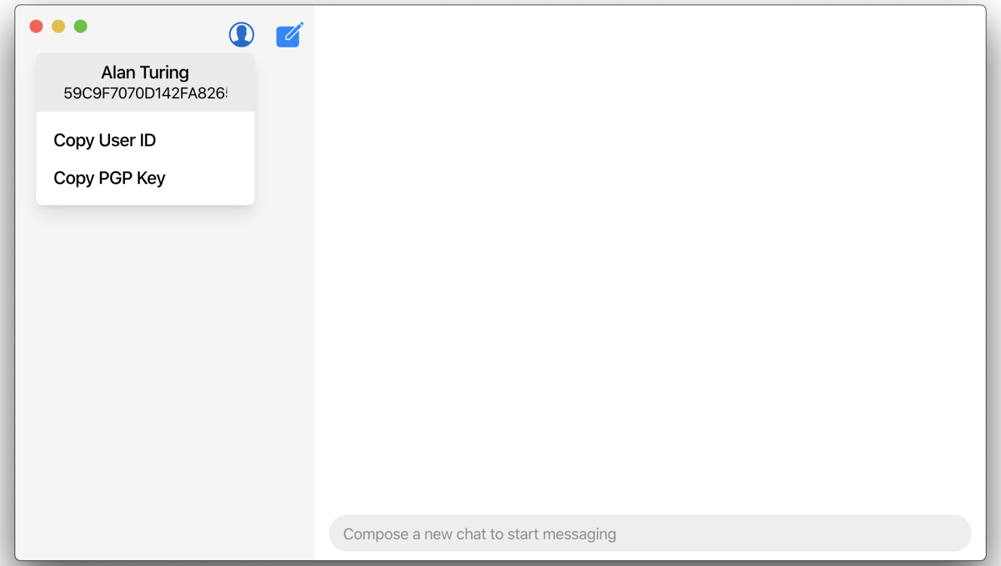
Secret Master Plan....

Elon Musk
F70366511FDB91BFED4

Not bad

B.6 Viewing your information

You can view your information including your full name, PGP key and userId by simply clicking on the *profile icon* at the top left of the sidebar. This will open your info as a dropdown like so for example:


Alan Turing
59C9F7070D142FA826