

# Modern End-to-End Encrypted Messaging for the Desktop

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering and Internet Computing**

eingereicht von

**Richard Bayerle**

Matrikelnummer 1025259

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdozent Dipl.Ing. Mag. Dr. Edgar Weippl

Mitwirkung: Dr. Martin Schmiedecker

Wien, 2. Oktober 2017

---

Richard Bayerle

---

Edgar Weippl



# Modern End-to-End Encrypted Messaging for the Desktop

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**Richard Bayerle**

Registration Number 1025259

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdozent Dipl.Ing. Mag. Dr. Edgar Weippl

Assistance: Dr. Martin Schmiedecker

Vienna, 2<sup>nd</sup> October, 2017

---

Richard Bayerle

---

Edgar Weippl



# Erklärung zur Verfassung der Arbeit

Richard Bayerle  
Seestraße 67  
78315 Radolfzell am Bodensee  
Deutschland

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Oktober 2017

---

Richard Bayerle



# Acknowledgements

Thank you, Katherine.





# Kurzfassung

Das Ziel dieser Arbeit ist, Benutzer von Desktop-Betriebssystemen an den Fortschritten im Bereich des Instant Messaging teilhaben zu lassen, die überwiegend auf mobilen Geräten gemacht wurden.

Um dies zu erreichen, wurden aktuelle Technologien identifiziert und bewertet, sowohl für zwei als auch für mehr Konversationspartner. Das Resultat ist ein direkter Vergleich zwischen *OpenPGP*, *OTR*, und dem *Signal Protocol* zum einen, und eine Gegenüberstellung üblicher Mechanismen für Gruppenkonversationen mit spezielleren Protokollen wie *np1sec* zum anderen. Weiterhin wurden die Anforderungen für ‘modernes’ Messaging spezifiziert, und ihnen entsprechende Technologien ausgewählt.

Das Ergebnis ist ein erfolgreiches Plugin für die *libpurple*-Messaging-Library, das das *OMEMO-XMPP*-Erweiterungsprotokoll implementiert, und somit auch dem weit verbreiteten *Pidgin*-Messenger zur Verfügung stellt. Um dies zu erreichen, wurden auch Libraries für das Arbeiten mit der *Signal Protocol*-Implementation in C sowie das *OMEMO*-Protokoll geschrieben. Weiterhin wurde auch ein *libpurple*-Plugin entwickelt, das es ermöglicht, mehrere Geräte gleichzeitig zu verwenden.



# Abstract

The aim of this thesis is to let users of desktop operating systems partake in the advances the field of instant messaging has made on mobile devices.

To this end, current technologies are first identified and evaluated, both for the two-party and multiparty case. The outcome is a direct comparison between *OpenPGP*, *OTR*, and the *Signal Protocol* for the former case, and a comparison between common schemes for secure multiparty communication and specifically developed protocols such as *np1sec*. Afterwards, the requirements for ‘modern’ messaging are laid down, and fitting technologies chosen.

Based on this groundwork, the main result is a successful *OMEMO XMPP* extension protocol plugin for the open-source *libpurple* messaging library whose graphical frontend *Pidgin* is a widely used messenger. Achieving this required writing libraries to handle the *Signal Protocol* library on a higher level, and for dealing with the *OMEMO* protocol itself. Additionally, a further *libpurple* plugin for enabling multi-device support is written.



# Contents

|  |             |
|--|-------------|
| <b>Kurzfassung</b>                                       | <b>ix</b>   |
| <b>Abstract</b>  | <b>xi</b>   |
| <b>Contents</b>  | <b>xiii</b> |
| <b>1 Introduction</b>                                    | <b>1</b>    |
| 1.1 Motivation . . . . .                                 | 1           |
| 1.2 Aim of the Work and Methodology . . . . .            | 2           |
| 1.3 Background . . . . .                                 | 2           |
| 1.3.1 Privacy Attributes . . . . .                       | 3           |
| 1.3.2 Technical Concepts . . . . .                       | 4           |
| 1.3.3 Common Attacks . . . . .                           | 5           |
| <b>2 State of the Art</b>                                | <b>7</b>    |
| 2.1 Overview of Two-Party Encryption Schemes . . . . .   | 7           |
| 2.1.1 (Open)PGP . . . . .                                | 7           |
| 2.1.2 OTR . . . . .                                      | 11          |
| 2.1.3 Signal Protocol . . . . .                          | 18          |
| 2.1.4 Other Protocols . . . . .                          | 24          |
| 2.2 Evaluation of Two-Party Encryption Schemes . . . . . | 24          |
| 2.2.1 PGP . . . . .                                      | 24          |
| 2.2.2 OTR . . . . .                                      | 27          |
| 2.2.3 Signal Protocol . . . . .                          | 30          |
| 2.2.4 Summary . . . . .                                  | 33          |
| 2.3 Overview of Multiparty Encryption Schemes . . . . .  | 36          |
| 2.3.1 Two-Party Scheme Reuse . . . . .                   | 36          |
| 2.3.2 Pairwise Sessions for Key Transport . . . . .      | 37          |
| 2.3.3 Use of Group Key Agreement . . . . .               | 38          |
| 2.3.4 Evaluation . . . . .                               | 42          |
| <b>3 Design</b>  | <b>45</b>   |
| 3.1 Approach . . . . .                                   | 45          |
| 3.2 Used Technologies . . . . .                          | 47          |
|  | xiii        |

|          |  |            |
|----------|--|------------|
| 3.2.1    | XMPP . . . . .                               | 47         |
| 3.2.2    | OMEMO . . . . .                              | 48         |
| 3.2.3    | Pidgin and libpurple . . . . .               | 51         |
| 3.2.4    | Programming Language . . . . .               | 52         |
| 3.2.5    | Additional Libraries . . . . .               | 52         |
| <b>4</b> | <b>Implementation</b>                        | <b>53</b>  |
| 4.1      | General Notes . . . . .                      | 53         |
| 4.2      | carbons . . . . .                            | 54         |
| 4.2.1    | XEP-0280: Message Carbons . . . . .          | 54         |
| 4.2.2    | Implementation Details . . . . .             | 56         |
| 4.3      | axc . . . . .                                | 57         |
| 4.3.1    | The libsignal-protocol-c Interface . . . . . | 57         |
| 4.3.2    | Implementation Details . . . . .             | 58         |
| 4.4      | libomemo . . . . .                           | 67         |
| 4.4.1    | XEP-0384: OMEMO Encryption . . . . .         | 67         |
| 4.4.2    | Implementation Details . . . . .             | 70         |
| 4.5      | lurch . . . . .                              | 75         |
| 4.5.1    | Design . . . . .                             | 75         |
| 4.5.2    | Implementation Details . . . . .             | 75         |
| 4.6      | Evaluation . . . . .                         | 87         |
| <b>5</b> | <b>Discussion</b>                            | <b>97</b>  |
| 5.1      | Conclusion . . . . .                         | 97         |
| 5.2      | Related Work . . . . .                       | 98         |
| 5.3      | Future Work . . . . .                        | 98         |
|          | <b>List of Figures</b>                       | <b>99</b>  |
|          | <b>List of Tables</b>                        | <b>101</b> |
|          | <b>Listings</b>                              | <b>101</b> |
|          | <b>Bibliography</b>                          | <b>103</b> |

# Introduction

## 1.1 Motivation

Most of the time, the motivation behind developing a new scheme or tool for end-to-end encryption will sound very similar to Philip Zimmerman’s article “Why I Wrote PGP” from 1991 [Zimb]. In short, it is the belief that privacy is a basic right. While this idea is not very controversial and part of many countries’ constitutions in one way or another, it seems to be the case that governments and companies simply cannot resist the temptation posed by the huge amounts of easily collectable and searchable personal data made available by the spread of information technology. Zimmerman updated his article in 1999 to feature a few examples from his home country, the United States, and more have emerged since then. A case which garnered a lot of attention was the revelation of the United States National Security Agency’s *PRISM* program, which enables it to directly access the data of many big providers of communication services used around the world, such as Microsoft, Apple, Google and Facebook [GM].

Although the disclosure of *PRISM* was shocking, the more unsettling side was the relatively lax restriction of access to private data already in place for law enforcement. Ignoring the political issues behind this, the central problem is having to trust such providers with sensitive information in the first place. Not taking legally binding orders into account, providers still often maintain invasive policies in regards to personal data. For instance, Google made it clear that users should expect their email will be “processed” [Rus], Microsoft adopted a similar approach after acquiring the popular messaging service *Skype* [Bee], and Yahoo agreed to scan incoming and outgoing emails in real time for law enforcement without question while hiding it from their own security team [Men].

Therefore, there is definitely a case to be made for not trusting service providers with access to private communications. Even a responsible company must comply with legal demands and government warrants, so it is better to avoid the question of trust altogether. This is more or less possible by employing end-to-end encryption, which limits the amount

of information providers can access. Naturally, a server needs to save some account information in order to function, but with growing public awareness regarding privacy, reducing the collected data has seemingly become a selling point. Informal evidence of this trend is the multitude of new messengers offering end-to-end encryption, examples of which can be found in later sections.

In today's technological landscape, it is not surprising these new developments usually focus on mobile devices. However, those with mobile devices often have and use desktop computers, which creates the demand for multiple device support and desktop clients. As a consequence, it is of interest to take a closer look at the progress introduced by mobile applications and find a way to adapt new schemes and concepts for the desktop setting.

### 1.2 Aim of the Work and Methodology

The goal of this work is to enable users of desktop operating systems to use a modern and effective end-to-end encryption scheme on a native client. *Modern* here means that additional popular use cases are covered, which are identified as support for connecting multiple devices to the same account, and group conversations.

In order to do this, the following questions need to be answered:

- Which new developments exist and how do they compare to the existing ones?
- Are adaptations or further improvements necessary?
- Can a two-party scheme be reasonably reused for the group setting, or are more specific protocols necessary?

The rest of this work is structured as follows:

First, a literature review is conducted to answer the posed questions. The results can be found in chapter 2. Second, suitable technologies need to be chosen and the general design laid out. This is done in chapter 3. Third, the implementation needs to be performed. It is described and evaluated in chapter 4. Lastly, the results are discussed in chapter 5.

### 1.3 Background

In this section, some key words and concepts of encrypted messaging will be explained in order to enable the understanding of the overview that follows in chapter 2. As some of the concepts, especially newer ones, do not have one exact definition in literature, it is useful to state how exactly even known terms are used in this document.

In accordance with general practice, conversation partners will be called *Alice* and *Bob*, a third conversation partner is *Carl*. An eavesdropper trying to learn the contents of a conversation is *Eve*.



### 1.3.1 Privacy Attributes

There are attributes which are generally used in literature to classify encrypted messaging protocols. An example can be found in [UDB<sup>+</sup>15], on which the following overview is loosely based.

Of course one can also find more, or present a more fine-grained view – both is done in [UDB<sup>+</sup>15] in addition to the selection found here –, but this is not necessary at this point and more details will be provided if necessary for the discussion.

**Confidentiality** This attribute describes the baseline of end-to-end encrypted messages: No one but the intended recipients should be able to read them.

Even though transport encryption like *SSL/TLS* is widely used nowadays – preventing eavesdropping on the network connection – data which is not additionally encrypted is still not confidential, as messaging usually relies on some kind of server for relaying the data between conversation partners, which then has access to the information.

Also note that in case of an authentication failure, the cryptographically intended recipient is not necessarily the recipient intended by the user.

**Integrity** It is possible to verify that an incoming message is exactly as it left the sender, i.e. has not been altered in transport.

**Authenticity** It is possible to verify the sender of a message.

**Deniability** A participant of a chat can deny having sent a certain message because from a cryptographic viewpoint, anyone could be its author.

**Forward Secrecy** Even if a user's long-term keys were obtained by a third party, past messages cannot be decrypted.

**Future Secrecy** Even if a user's ephemeral keys were obtained by a third party, future messages cannot be decrypted. Note that this excludes an active attacker who gains access to long-term keys.

When considering the group setting, the following additional attributes are usually also taken into consideration:

**Origin Authentication** It is not only possible to verify that a message was written by a group member, but also the exact sender.

**Transcript Consistency** It is possible to verify that all group members see the same transcript of the conversation, i.e. no messages were added or held back for single users.

**Participant Consistency** It is possible to verify that all group members see the same group participants. In practice this protects from hidden users who can still read the conversation.

### 1.3.2 Technical Concepts

Naturally, talking about encryption in some depth is not possible without knowing some concrete concepts and techniques. While most are commonly known, their effect on the introduced attributes might not be.

This section deals with some of those that are often found in applications for encrypted messaging, as they are either a problem that needs to be solved, or a common solution to one. These definitions are also going to be useful during comparison and evaluation.

**Types of Cryptography** For the sake of completeness, let it be said that there are two categories of cryptographic systems. *Symmetric-key* algorithms use the same key for encryption and decryption, while *public-key* algorithms make use of a pair of keys – the private key can decrypt what was encrypted with the public key, and vice versa. Often also just called *symmetric* and *asymmetric encryption*, both are used to achieve *confidentiality*, frequently even both at the same time.

**Digital Signature** Closely related to the concept of public-key algorithms, digital signature schemes use the signer’s private key and some data as input to produce a *tag* as proof of *authenticity* and *integrity* of said data.

This proof is then considered *cryptographically* non-repudiable, i.e. it could have only been produced using that specific private key.

**Message Authentication Code** Usually abbreviated to *MAC*, these are used similarly to digital signatures, that is to provide authenticity and integrity. However, instead of one person’s private key, a shared secret is used to produce the tag from the message. This has the effect of removing the non-repudiability property, as anyone who knows the secret can produce this type of tag. In turn, it adds a certain degree of *deniability*, as while Bob definitely knows an incoming message must have come from Alice, Eve cannot tell who a message originated from, and cannot prove it was either of them if she managed to get her hands on the key data, as she could have produced the message herself. This also means that if there are more than two participants, *MACs* fail to provide origin authentication.

A common implementation is *HMAC*<sup>1</sup>, in which any cryptographic hash function can be employed to calculate an authentication tag.

**Key Exchange** In order to make use of a shared secret as a key for symmetric encryption, it first has to be established. Before public-key cryptography can be used, public keys need to be exchanged between the conversation partners. There are a number of

---

<sup>1</sup><https://tools.ietf.org/html/rfc2104>

algorithms concerned with this problem.

Even after obtaining such a symmetric or asymmetric key, one needs confirm its authenticity, which is a major design task for each scheme.

**Key Derivation Function** Often shortened to *KDF*, it is used to derive an appropriate cryptographic key from some form of input data. ‘Appropriate’ here means adequately random, but also of the necessary size for the algorithm it is going to be used with. Again, a common variant employs cryptographic hash functions to achieve this goal, which is then called *HKDF*<sup>2</sup>. In fact, it does not use the hash algorithm itself, but a *HMAC* function which utilizes it, together with some optional input like a salt, info string, or both.

**Cryptographic Ratchet** A ratchet is “a mechanism that [...] [allows] effective motion in one direction only”<sup>3</sup>. This word was first applied to cryptography by Adam Langley in his description of his program *Pond* [Ada], and can be understood in terms of the key material moving forward in a non-predictable way, which achieves forward secrecy as past messages cannot be decrypted.

### 1.3.3 Common Attacks

Another category of problems contains the ones caused by malicious attackers. Again, these issues often go along with encrypted messaging, so it makes sense to explain them once and then look at how they are solved, given they exist in a specific scheme.

**Man-in-the-Middle** Often abbreviated to *MitM*. A situation where Eve relays messages from Alice to Bob, and from Bob to Alice. While both of them think they are talking to each other, they are talking to Eve, who thus has access to the messages sent. In the context of encrypted communication, this is usually only relevant during session establishment, as afterwards Eve can only receive the ciphertext, learning nothing. Accordingly, it is one of the main points to consider for a key exchange scheme.

**Unknown Key-Share** When Eve requests Alice’s key data and afterwards – *unknownst* to Alice – *shares* it to Bob as her own, she performed the *UKS* attack. Sometimes it is also called *identity misbinding attack*.

The effect is that Eve now can forward messages she received from Bob to Alice, to whom it looks like she is the original and intended recipient, as Bob used her key data.

**Replay Attack** If this kind of attack is possible, Eve can save and reuse intercepted ciphertext messages without having to decrypt them (given she can somewhat guess their content, e.g. because the protocol is publicly documented and the first steps are always the same). Such a vulnerability can happen when messages do not contain any hints

---

<sup>2</sup><https://tools.ietf.org/html/rfc5869>

<sup>3</sup><http://www.merriam-webster.com/dictionary/ratchet>

## 1. INTRODUCTION

---

about their ‘freshness’, such as an indication which message is replied to, or simply some sort of ID.

# State of the Art

## 2.1 Overview of Two-Party Encryption Schemes

### 2.1.1 (Open)PGP

#### Background

*PGP* stands for PRETTY GOOD PRIVACY and is an encryption scheme created by Philip Zimmerman in 1991 [ASZ96]. His motivation was the belief in the right to a private conversation, something that was not possible in electronic form at that time because strong cryptography was not available to the public [Zimb, Zima].

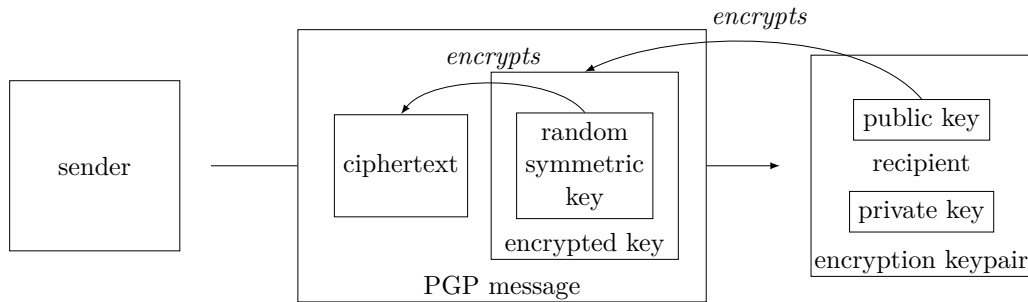
Back then, electronic communication was usually done via email, bulletin board systems or newsgroups. In many of these, *PGP* is still widely used today in the form of its successor format, *OpenPGP*. Additionally, it is still often used in new developments, such as *OX* [Flo], which utilizes *OpenPGP* to enable encrypted chats using *XMPP* (see 3.2.1) as transport protocol.

In this document, the name *PGP* generally refers to this newest standard, and it should be noted that the basic principle described below did not change.

#### Technical Details

As per the RFCs [ASZ96, CDFT98, CDF<sup>+</sup>07], programs which implement *PGP* have to offer four main services that are required to produce the packets in the format also described therein. These services are confidentiality, digital signature, compression, and Radix-64 conversion. Another service not explicitly mentioned in the RFC which is often also found in programs implementing *PGP* deals with key management.

These services will be briefly explained here.

Figure 2.1: Structure of a *PGP* encrypted message.

While there are at least one algorithm per category that has to be implemented to ensure interoperability, it is not necessary to implement all of them. The algorithm used can then be specified in a certain field of the wire format using the IDs defined in the RFC. As an example, *AES-128* has the ID 7 in the category of symmetric-key algorithms.

**Confidentiality** Confidentiality is achieved by employing a combination of symmetric and asymmetric cryptography. For each “object”, i.e. a message or a file, a one-time random symmetric *session key* is generated which is then used for encryption.

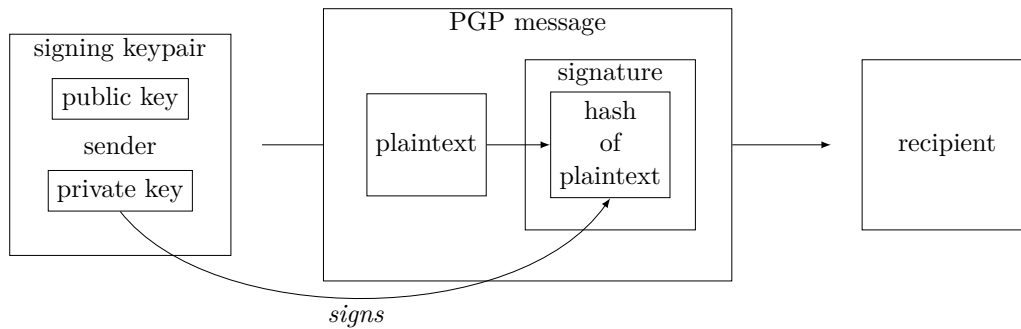
This session key is then encrypted itself using the recipient’s public key and sent with the message. The recipient can then decrypt the session key using his private key, and use it to decrypt the object. A depiction of a *PGP* message can be found in figure 2.1. In the case of multiple recipients, the session key can simply be encrypted with each of their public keys separately. As the fixed-length key is likely shorter than the plaintext, this method is probable to be faster than encrypting all of the data for every recipient. The must-implement algorithms are *Elgamal* [ElG85] as public-key algorithm, and *TripleDES*, *AES-128* [Nat01] and *CAST5*<sup>1</sup> as symmetric-key algorithms.

**Digital Signature** As noted in section 1.3.2, the digital signature service may be used to ensure authenticity of a message. In *PGP*, it is implemented by creating a hash of the message, and then signing it using the private key of the sender and a digital signature algorithm.

On the receiving end, the signature can be verified by using the sender’s public key. Since it is assumed only the owner of the private key could have used it to create the signature, this ensures the identity of the sender. The now verified hash can then be compared to the locally calculated hash of the message. If the received and calculated hashes are the same, the message has not been altered. Figure 2.2 shows this procedure. Often, encryption and signatures are used together – in this case, the signature is created over the plaintext as shown, and then the ciphertext is created over both the original message and the attached signature.

One should note that the public and private keys mentioned here are not the same keys

<sup>1</sup><https://tools.ietf.org/html/rfc2144>

Figure 2.2: Structure of a *PGP* signed message.

used for confidentiality. *PGP* ‘keys’ actually contain a set of keys – the master signing key, and a number of subkeys, e.g. the one used for encryption on the same device, or signing and encryption keys used on other devices. Thus, when creating a key, the user has the possibility to create a *DSA* or *RSA* sign-only keypair, or a pair for signing and encryption using *DSA/Elgamal* or *RSA/RSA*.

In the first versions, *MD5*<sup>2</sup> was used as hashing algorithm, but it is now deprecated as it is easy to create collisions (see e.g. [Ste06]). It was replaced by *SHA-1*<sup>3</sup>, but other algorithms such as *RIPEMD-160*<sup>4</sup> are also optionally supported.

As for digital signature algorithms, both *DSA* [Nat13] and *RSA*<sup>5</sup> must both be implemented.

**Compression** This step is optionally applied before encryption, but after the signature, if this service was used. It is not integral to the functioning and simply decreases the amount of data that is sent.

At first, only the *ZIP* algorithm<sup>6</sup> was supported, and its implementation is still recommended. But by now, other popular algorithms such as *ZLIB*<sup>7</sup> and *BZIP2*<sup>8</sup> can be specified in the official message format.

**Radix-64 Conversion** The internal representation of encrypted data, signatures, and keys is a stream of arbitrary octets, or 8-bit bytes. In order to make sure that this information can be stored on systems and transmitted through mediums that might not support raw binary data in this format (such as emails), *PGP* can represent it in printable ASCII characters.

<sup>2</sup><https://tools.ietf.org/html/rfc1321>

<sup>3</sup><https://tools.ietf.org/html/rfc3174>

<sup>4</sup><https://homes.esat.kuleuven.be/~bosselae/ripemd160.html>

<sup>5</sup><https://tools.ietf.org/html/rfc8017>

<sup>6</sup><https://tools.ietf.org/html/rfc1951>

<sup>7</sup><https://tools.ietf.org/html/rfc1950>

<sup>8</sup><http://www.bzip.org/>

This representation is called *Radix-64*, and the whole output including correct formatting and headers is called *ASCII Armor*.

*Radix-64* is based on the *MIME Base64* encoding<sup>9</sup> developed for emails (*MIME* are the MULTIPURPOSE INTERNET MAIL EXTENSIONS). This means the data is represented using the characters a-z, A-Z, 0-9, +, /, and = for padding.

More specifically, the *Radix-64* encoding is simply the *base64*-encoded data and an optional 24-bit *CRC* (CYCLIC REDUNDANCY CHECK) checksum represented in the same set of characters.

### The Web of Trust

As mentioned in the introduction, the specification does not define a method to actually exchange the keys. They can be often found as email signatures or on personal blogs, but there also exist keyservers which all hold the same data, synchronized by the *SKS* (SYNCHRONIZING KEYSERVERS) protocol [UHHC11]. Keys uploaded to keyservers can easily be searched: A key can be identified by its 8-octet key ID, the fingerprint, or the user ID of its owner, which is made up from a username and an email address. While none of these are guaranteed to be unique, one usually knows at least two.

Assuming Alice found Bob's key on one of the keyservers<sup>10</sup>, how does she know it is actually his? More precisely, how can she *trust* it is his?

Public-key infrastructures (*PKIs*) often rely on (a hierarchy of) *certificate authorities* (*CAs*) to provide this trust – a key signed by such an authority can be trusted to be valid, and the list of trusted authorities is usually distributed with the software. This happens e.g. for *TLS*: web browsers come with a list of trusted *CAs*. While *CAs* also exist in the *PGP* model, they are just a special category of users, as anyone can sign a key, attesting to its authenticity.

In both cases, there needs to be a trusted *chain* of these certificates, but while it is fixed in a hierarchical structure, such a chain first needs to be found in the *WoT* model.

This chain can be found because a *PGP* key actually contains all of the signatures it collected as certificates. Alice can search the keys she already knows and has added to her keychain, e.g. because they were exchanged in person, for a path to Bob's key. Luckily, her friend Carl, whose key she has added, signed Bob's key, so the chain is rather short. But before she can also *trust* it to be Bob's key, there's another important point to consider: How much does she trust Carl to verify the authenticity of Bob's key before signing it?

This is called *introducer trustworthiness*, and each implementation can and does use its own metrics to derive the total trustworthiness of the found path. For example, in *GnuPG* the certification path may not be longer than five keys, and they all must be fully trusted, or there must be three redundant paths.

---

<sup>9</sup><https://tools.ietf.org/html/rfc2045>

<sup>10</sup>e.g. <http://keys.gnupg.net>



Note that this trustworthiness decision is a local one and, unlike the certificates, is not propagated. Alice may trust Carl to verify keys before signing them, but since she just started talking to Bob, she does not trust him yet to do the same. On the other hand, Carl has known Bob for a long time and trusts him well. As a result, the *WoT* differs for each user, and in effect mirrors social relations in the real world, i.e. it will be hard for Alice to find a trustworthy certificate chain for a person who does not somewhat belong to her circle of friends already [UHC11].

More in-depth analyses can be found in [Mau96] and [Car00].

### 2.1.2 OTR

#### Background

*OTR* stands for OFF-THE-RECORD and thus carries in its name what the designers Nikita Borisov, Ian Goldberg, and Eric Brewer wanted to achieve: private electronic communication [BGB04]. The paper it was first described in is pretty straightforward concerning its intentions – the second part of the title is “Why Not To Use PGP”.

Their criticism of *PGP* will be discussed in more detail in section 2.2. Generally said, the authors work with an updated definition of *privacy*, and an updated model of electronic communication. This updated definition includes forward secrecy and deniability, and it will be explained below how these are achieved. Instead of considering email and BBS posts, the chosen type of electronic communication is instant messaging, facilitating the attainment of said attributes.

As can be seen in the next section, the cost of “better” attributes is higher complexity. The result is that unlike in *PGP* the steps cannot realistically be done manually any longer.

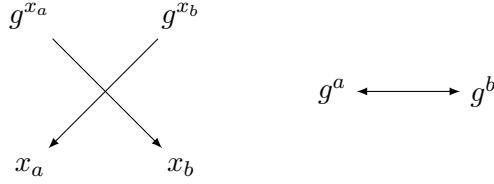
The first version of the suggested protocol was not without errors, and it was updated accordingly. As these mistakes provide an interesting insight in points to consider when designing such an encryption scheme, the evolution of this protocol will also be inspected instead of just considering the newest version as a whole.

#### Technical Details

**Session Establishment** To communicate using *OTR*, a session needs to be established first. In the first version, this is done by performing the *Diffie-Hellman Key Exchange*, also often abbreviated to *DH*. Whitfield Diffie and Martin Hellman developed it in 1976 based on the concepts of Ralph Merkle in order to enable two parties to create a secure connection over a public channel [DH76], so exactly what is needed here. In following versions, a different algorithm is used for the initial key exchange to achieve additional properties, but as it will become clear, this approach still plays an important part in the whole scheme.

It works as follows: Beforehand, Alice and Bob agree on a value  $p$ , which is a prime,

Figure 2.3: Establishing the *DH* shared secret  $g^{ab}$  between Alice and Bob. The short notation on the right will be useful later.



and a value  $g$ , which is a primitive element of a finite field with  $p$  number of elements. Having established this, Alice now picks a secret value  $x_a$ , and Bob a secret value  $x_b$ , with  $1 \leq x_a, x_b \leq p-1$ . This  $x$  is the *exponent* they will use, so afterwards, they can compute the value  $g^{x_a} \bmod p$  and  $g^{x_b} \bmod p$  respectively, and send each other these values. This  $x$ , i.e. the chosen exponent, is also called the short-term or ephemeral *DH private* key, and the whole expression  $g^x \bmod p$  the short-term or ephemeral *DH public* key (as opposed to long-term keypairs as e.g. used for digital signatures). Both Alice and Bob can now compute the same secret value  $s$  by exponentiating the received ephemeral public key with their own ephemeral private key, because:

$$\begin{aligned}
& (g^{x_b})^{x_a} && \bmod p \\
= & g^{x_b x_a} && \bmod p \\
= & g^{x_a x_b} && \bmod p \\
= & (g^{x_a})^{x_b} && \bmod p
\end{aligned}$$

Figure 2.3 also displays the procedure schematically.

For brevity, the  $\bmod p$  suffix will not explicitly be stated in the rest of the thesis, and only the exponent's suffix is used to identify it. This means that e.g. the private key  $x_a$  is shortened to  $a$ , and the corresponding public key from  $g^{x_a}$  to  $g^a$ .

In *OTR*, the first step of agreeing on values  $p$  and  $g$  is omitted, as  $g$  is simply fixed to 2, and  $p$  to the 1536-bit number defined by:

$$2^{1536} - 2^{1472} - 1 + 2^{64} * ((2^{1406} * \pi) + 741804)$$

The chosen  $x$  has to be at least 320 bits long. From the shared secret  $s$ , several values are derived by hashing it in various ways using *SHA256*. These are for instance *AES-128* encryption keys and *SHA256-HMAC* keys [OTR].

**OTR Ratchet** One of the main improvements the authors had in mind was *forward secrecy*. As a *OTR* views instant messaging as synchronous (i.e. users sit in front of their computer and pay attention to the messenger), a cryptographic session to them is about the same as a messaging session, that is, it is established when the conversation begins, and torn down when the client exits [BGB04].

In theory, this model alone is already enough to achieve forward secrecy, as the authors

note themselves. However, to decrease the amount of decryptable messages in case of a compromise, a *DH* exchange is (ideally) performed with every message, instead of at the beginning of a session. While this scheme was not called a “ratchet” because the term did not exist at that time, this name was applied to the algorithm when the concept got popular, e.g. in [Moxa]. Even though this goal was not explicitly stated, this approach additionally provides *future secrecy*.

Similarly to the first key derived at session establishment, the following message keys  $k_{ij}$  are then also obtained from a shared secret  $g^{ab}$ . In order to streamline the establishment of these following short-lived sessions, the ephemeral public keys  $g^{a_i|b_i}$  are simply sent alongside the regular messages. So the first message from Alice to Bob will not only contain the ciphertext encrypted by a message key  $k_{00}$  derived from the shared secret  $g^{a_0b_0}$ , but also a value  $g^{a_1}$ . When replying, Bob’s message will now be encrypted with a new key  $k_{10}$  derived from  $g^{a_1b_0}$ , and comes with a new value  $g^{b_1}$ . Alice can then encrypt her answer using the key  $k_{11}$  obtained from hashing  $g^{a_1b_1}$ , and so on.

In this ideal case, the message key is different with each message. If the conversation partners do not take turns sending messages, e.g. because Alice is telling a long story, the same key keeps being used for every message, and the same ephemeral public key will be announced alongside it. Until Bob replies, and therefore confirms the announced ephemeral public key by using it for the encryption key himself, Alice cannot use her new key. Also note that a shared secret is never used to derive a message key for both encryption and decryption by the same person.

An example of this is illustrated in figure 2.4.

**Authentication** As it stands, both Alice and Bob only know that they established a shared secret with someone else. This someone could also be e.g. Eve the malicious server administrator performing a *MitM* attack.

*OTR* initially solved this problem by employing digital signatures. Aside from the ephemeral *DH* keypair generated for session establishment, each client also generates a longstanding *DSA* keypair. This is used to sign the transmitted *DH* public key during the initial exchange, which is therefore called *authenticated key exchange*, or *AKE*. The public *DSA* key is also appended to these key exchange messages, as can be seen in figure 2.5.

While this does not protect from a *MitM* attack per se, it can be detected by comparing the sent public *DSA* key to a previously received one. Unless the attack was performed every single time with the same key, a differing key should raise some suspicion. This approach of course is still not a guarantee unless the keys are known to be valid by verifying them out-of-band, e.g. by comparing fingerprints in a personal conversation, which the paper suggests doing.

In order to nonetheless achieve deniability for the actual messages, *MACs* are then used instead of digital signatures for the rest of the conversation. As already noted, the *SHA256-HMAC* key data is derived from the *DH* secret for this purpose.

When *OTR* was scrutinized later, the used *AKE* was found to have several weaknesses [DRGK05]. For instance, a replay attack can be performed because the public *DH* key

Figure 2.4: An *OTR* ratchet example.

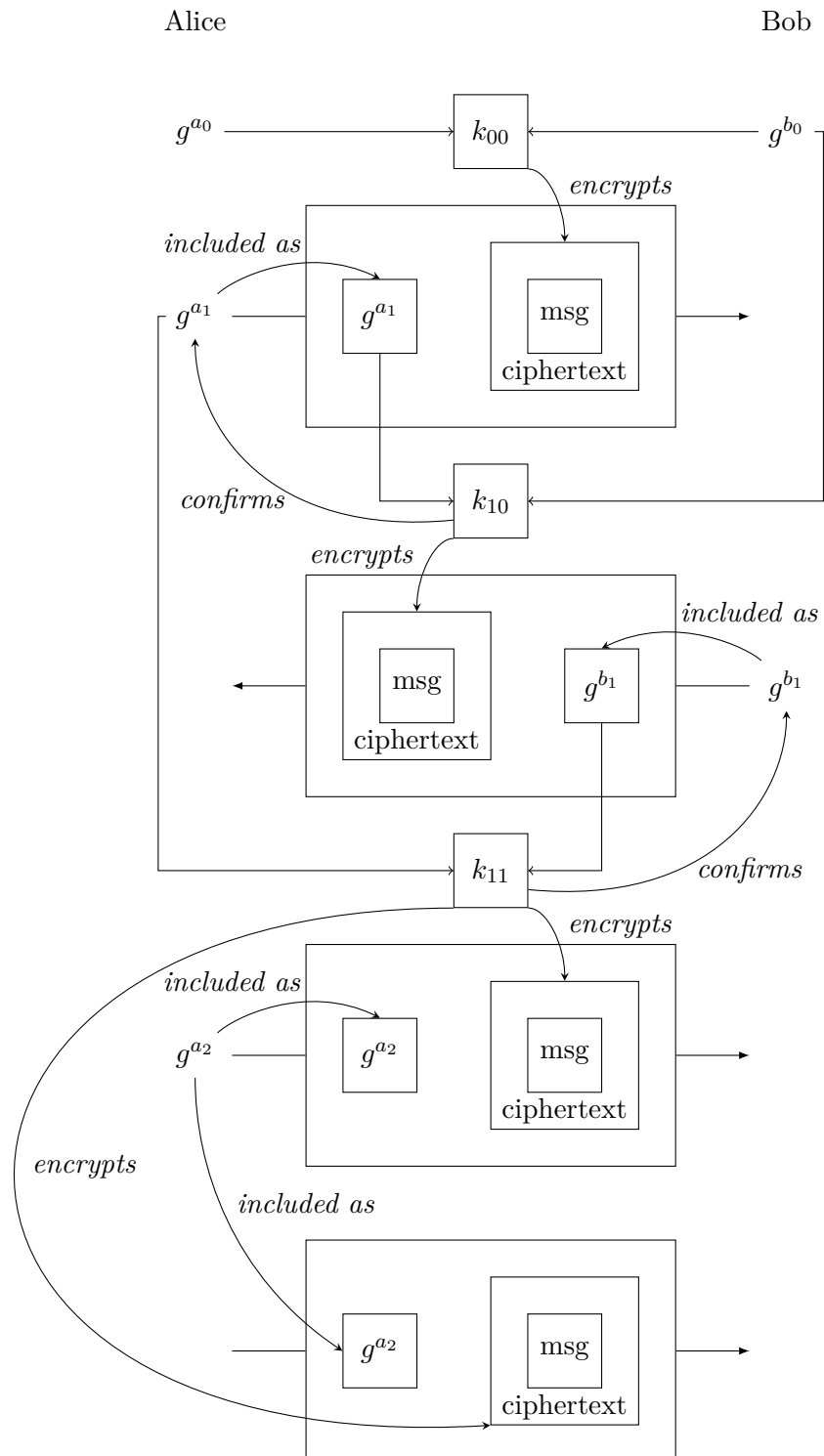
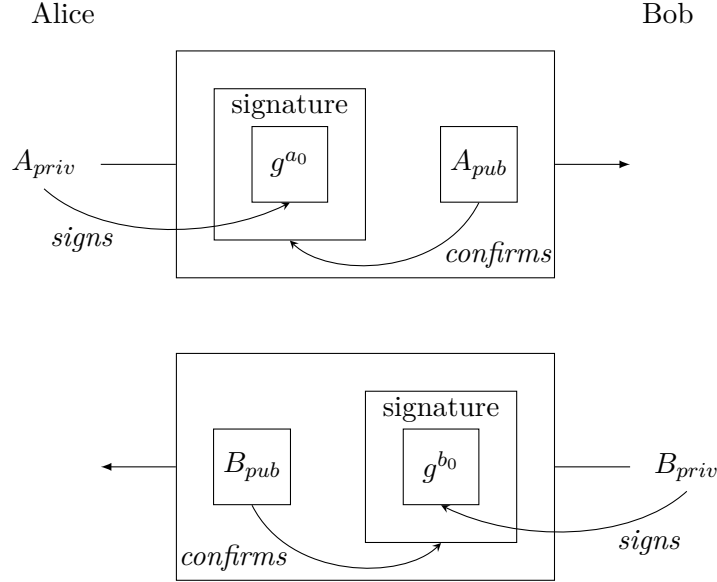


Figure 2.5: *OTR*'s initial ‘authenticated DH’.  $A$  and  $B$  are Alice’s and Bob’s *DSA* signing keypairs respectively.



is sent in the clear together with a signature. As long as the key behind the signature does not change (e.g. because the client is reinstalled), this message containing both the ephemeral key and a signature on it can be used to impersonate the sender. The other vulnerability, a possible *UKS* attack, is based on the same design mistake.

For version 2, *OTR* followed the advice presented in the same paper and improved their *AKE* by employing the *SIGMA* protocol (“SIGN-AND-MAC”) in an own adaptation based on the *SIGMA-R* variety as described in [Kra03]. This adaptation is described in [AG07], but as it was only done to work around message size limitations and adds complexity, the original scheme is described here instead.

Initially, the *DH* ephemeral public keys  $g^{a_0}$  and  $g^{b_0}$  are exchanged without any authentication, i.e. like in simple *DH* seen in section 2.1.2. Note though that those keys are not the only information being sent - a session identifier and a nonce are usually also needed for keeping track of messages and helping the processing as well as preventing replay attacks. After the *DH* public values were exchanged by Alice and Bob, the *DH* shared secret  $g^{a_0b_0}$  is used for derivation of an encryption key  $k_e$  for the remaining two handshake messages, and a *MAC* key  $k_m$  for computing a *MAC* over the long-term public signing key. Since  $k_e$  and  $k_m$  can be derived from the shared secret by both Alice and Bob, they can be used on the other end for confirming the *MAC* and therefore also the authenticity, and decryption of the whole message. Aside from the public signing keys  $A_{pub}$  and  $B_{pub}$  and their *MAC*s, these two last messages encrypted with  $k_e$  also contain a summary of the data exchanged so far, and a signature created over this information using the private counterpart of the authenticated public signing keys,  $A_{priv}$  and  $B_{priv}$

respectively. This can be observed in the illustration found in figure 2.6.

Improving the authenticated handshake protocol still does not change the fact that a *MitM* attack is possible. In order to detect it, users still have to manually confirm that the long-term public signing keys look the same to both parties. To make this comparison a bit simpler, usually not the complete key data but its identifying *fingerprint* is displayed to the user, which in case of *OTR* is its *SHA-1* hash. In the view of the *OTR* authors, this process had to be simplified even further in order to also protect users who do not have an understanding of public-key cryptography. This is why they added the *SMP*, or *SOCIALIST MILLIONAIRES' PROTOCOL*, to *OTR*.

In the *Socialist Millionaires' Problem* [JY96], two millionaires want to know if they are equally rich, but do not wish to disclose their wealth to each other, or anyone else. In the original problem [Yao82], the millionaires wanted to know who of them is richer, so in this case they are 'socialist' because they want to know if their secret is the same. (Not disclosing their wealth to an eavesdropper probably helps to protect them from being sent to jail by their socialist state's secret police too.)

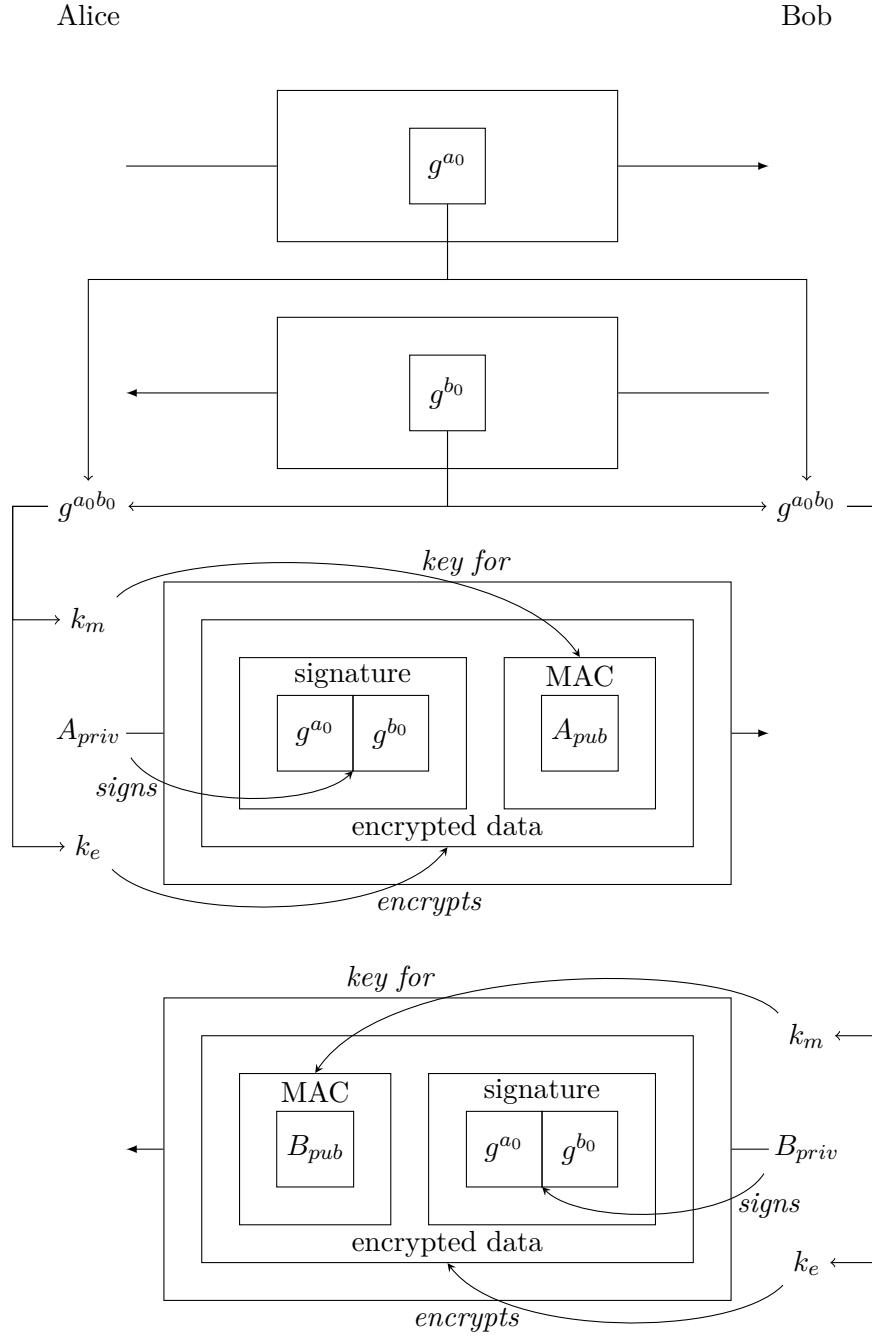
Based on the assumption that two people who wish to have a conversation with each other will always have some sort of secret an unrelated third party could not easily guess, *OTR* adapts the presented problem to let two users compare this real-world secret they share without revealing any information except for the fact that the secret matches (or not) to anyone. So when the *SMP* is initiated, both users are prompted to enter a string, which is then compared in this way. To prevent a third party from simply forwarding these messages, the actual secret which is compared consists not only of the entered word or expression, but also of the *SHA-256* hash of the session ID and both Alice's and Bob's key fingerprints [AG07].

The specifics are mathematically too involved to be relevant in this thesis, but can be found in the paper [AG07]. There is an interesting fact to *SMP*, though: Just like the *DH* handshake, it is based on the assumption that given  $g^x$  and  $g^y$ , it is infeasible to find  $g^{xy}$ . This is also called the *COMPUTATIONAL DIFFIE-HELLMAN ASSUMPTION*, or *CDH assumption*. In fact, two *DH* handshakes take place in the beginning to agree on two other generator values  $g_2$  and  $g_3$  which will be needed to perform the algorithm ( $g_1$  is 2, as before).

**Forgeability** In addition to providing cryptographic deniability, *OTR* goes one step further and tries to make messages forgeable. The intended effect is that in theory anyone observing the communication can just as likely have created any message in the first place. To achieve this, the authors use two measures.

On the one hand, *AES* is used in *counter mode* (*CTR*), which works (for any block cipher algorithm) by encrypting some counter that is increased for every block, and then applying the *XOR* operation on the resulting data and the plaintext block. This turns the block cipher into a stream cipher, the consequence of which is that the ciphertext could still decrypt to something meaningful if a bit is changed, as it has exactly one corresponding bit in the plaintext. Other modes chain the input, therefore the result is more likely to be uninterpretable garbage data when the ciphertext is modified.

Figure 2.6: The basis for *OTR's* improved handshake, the *SIGMA-R* authenticated key exchange protocol.



The authors reason that Eve could change the ciphertext to decrypt to anything she wants given she can guess the original plaintext, increasing the forgeability of messages. Meanwhile, Alice and Bob can still use the *MACs* to verify the integrity of the messages they receive.

On the other hand, *MACs* are a central part for their main forgeability measure. Once Alice knows that all messages using a certain *MAC* key have been received, she publishes that *MAC* key with her next message. She knows this because Bob confirmed the ephemeral public *DH* key the *MAC* key is derived from by sending a message encrypted with an *AES* key derived from the same secret.

While the authenticity and integrity of messages can be verified when they are received, this is not possible after the key was published, as it can now be used on any message. The authors say this can be seen as “forward secrecy for authentication”.

### 2.1.3 Signal Protocol

#### Background

In 2010, the company Whisper Systems released the beta of their new instant messaging application called *TextSecure* [Whi]. They chose Android as the platform, and this is because initially the idea was to make SMS more secure; an idea that was eventually abandoned because of a multitude of reasons. For instance, SMS as a transport always leaks metadata and therefore can never be really private. Also, SMS can not really be used for seamless encrypted conversations as there e.g. is no way to detect an uninstall for a conversation partner [Moxh].

After using *OTR* for encryption in the beginning, an own scheme was developed which is better suited for mobile platforms, and solves some issues which will be discussed further below in the evaluation chapters. The result had several parts: a new key exchange scheme, a ratchet algorithm, and the actual protocol incorporating these. All of it combined, but also just the ratchet, were both named *Axolotl* at first. Later, when *TextSecure* was merged with the *RedPhone* application for encrypted voicecalls, it was renamed to *Signal* [Moxd, Moxi]. With this, the ratchet was renamed to *Double Ratchet*, and the protocol to *Signal Protocol*.

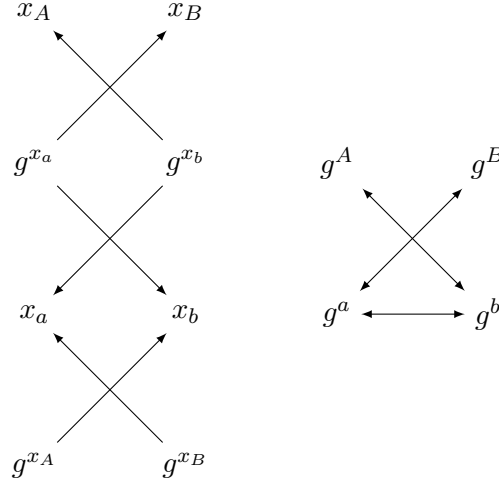
In this section, the *Signal Protocol* will be inspected to determine the characteristics which are supposed to make it an improvement over *OTR*, and how exactly these were implemented.

#### Technical Details

**Session Establishment** For the *Signal Protocol*, the developers adapted the *3DH* or *TRIPLE DIFFIE-HELLMAN* handshake from [KP05], which in turn is based on a protocol presented in [BWJM97]. Instead of just performing one *DH* exchange using both users’ ephemeral public keys  $g^{x_a|b}$ , there are (unsurprisingly) three exchanges, as each user also has a long-term *DH* keypair and additionally sends this long-term public key  $g^{x_{A|B}}$  for



Figure 2.7: Establishing the three *DH* shared secrets  $g^{ab}$ ,  $g^{aB}$ , and  $g^{Ab}$ . Short notation on the right.



combination with the other person's ephemeral public key.

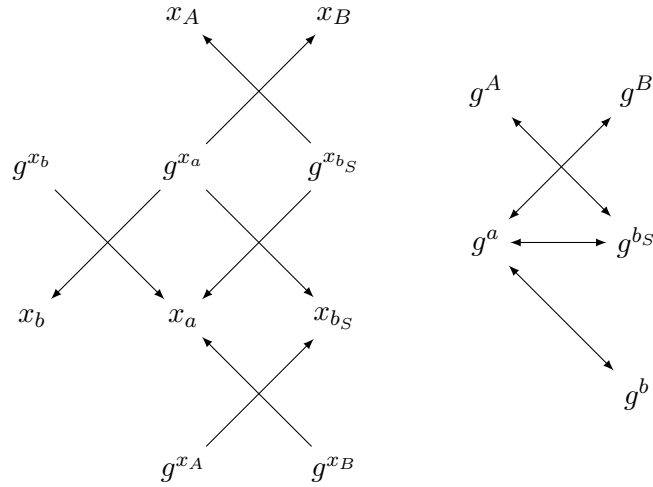
The result is three shared secrets  $g^{x_a x_b}$ ,  $g^{x_a x_B}$ , and  $g^{x_A x_b}$  (or shorter:  $g^{ab}$ ,  $g^{aB}$ , and  $g^{Ab}$ ) which can be used for further calculations such as deriving a single shared secret via some sort of *KDF* [Moxj]. This is shown in figure 2.7. Note that even though there are now three secrets, the exchange is still completed by one message from each participant as both long-term and short-term keys can be transported at the same time.

Using this algorithm enabled the developers to allow asynchronous session establishment with the help of an additional trick: Both the long-term *DH* public key (the *identity key*) and a number of pre-computer short-term *DH* public keys (the *pre-keys*) are uploaded to a server which is assumed to always be online, unlike Bob. If Alice wants to message her new contact Bob, she can ask this server for a *bundle* consisting of Bob's identity key and one of his pre-keys. After receiving this information, Alice only needs to generate the own ephemeral *DH* keypair, following which she can compute the shared secret to derive a session. The first message to Bob can then already be encrypted using this session, Alice just has to add both of her public *DH* keys  $g^a$  and  $g^A$  as well as the ID of Bob's pre-key she used in plaintext, making it a *pre-key message*. All Bob has to do is look up the private counterpart of the used pre-key by the received ID, and derive the encryption key from the calculated secrets.

The described procedure is part of the *X3DH* (EXTENDED 3DH) specification which is implemented in the current version of the *Signal Protocol* [Mox16]. In fact, the possibility to perform a synchronous handshake in absence of a server, i.e. the regular *3DH* exchange, was removed from the official library <sup>11</sup>.

<sup>11</sup><https://github.com/WhisperSystems/libsignal-protocol-c/commit/d83a61a328d4e36bcccf9066c925b63bb75bf968>

Figure 2.8: In *X3DH*, Bob’s signed pre-key replaces his regular pre-key, which can then optionally be used for a fourth secret computed with Alice’s ephemeral key.



In this extended version, which is based on the idea of adding signatures from [CF11], the uploaded bundle also contains a single pre-key signed with the identity key and the corresponding signature. This signed key actually takes the place of the one-time pre-key as described before, but can be used more than once. Optionally, but recommended for better security properties, a one-time pre-key can additionally be used like before to calculate a fourth shared *DH* secret with the initiator’s ephemeral key. Note how in figure 2.8 the signed pre-key  $g^{bs}$  replaces  $g^b$ , which is pushed to the side.

Instead of ‘regular’ *DH*, the protocol uses *ECDH* (ELLIPTIC CURVE DIFFIE HELLMAN) over either the *Curve25519* or *Curve448* [Mox16], so the actual calculations differ and e.g. use multiplication instead of exponentiation, but the  $g^x$  format was kept for consistency in order to ease understanding. The *Signal Protocol* implementation uses *Curve25519*. Now, one detail is still missing: Since the identity key is a *Curve25519 DH* key, how can it be used to create a *EdDSA* signature, which requires a *Ed25519* key? The answer is *XEdDSA*, which was also developed in the context of this protocol and provides a way to convert a *Curve25519 DH* key to a *Ed25519 EdDSA* key [Tre16a]. This is simply done on-the-fly when a signature is needed.

Finally, to provide some mitigation against *UKS* attacks, some authenticated data is also part of the final message format. This data at least contains both used identity keys, but can also contain more identifying data, sacrificing deniability.

**KDF Chain** Before getting to the ratchet algorithms, it is important to introduce the notion of the so-called *KDF chain*, as the final scheme contains two of those per user. It is called a ‘chain’ because it is iteratively applied, i.e. every *KDF* key but the very first one is both the output from a previous *KDF* computation and part of the input for the next key, the other part being some additional data such as a constant.

An important detail is that the *KDF* key is not the only result of the *KDF* functions defined in the *Signal Protocol*. Their output is a pair, but as this pair is computed in different ways, the exact explanation will be delivered further below.

After a first shared secret of 32 bytes has been established through some sort of handshake, which in the *Signal Protocol* is of course *X3DH*, it can be used as the first *root key* of the chain, from which a *sending chain key* and *receiving chain key* are derived. This *sending chain key*,  $CK_s$  for brevity, is then used to compute a key for the symmetric-key cipher which is used for encrypting the plaintext messages. The other party's *receiving chain key*  $CK_r$  corresponds to the same key, and vice versa (or else the correct decryption key could not be obtained).

*AES-256* in *CBC* mode is recommended as the symmetric-key algorithm, and used in the reference implementation. How these keys are moved forward in their 'chain' by the ratchet and how exactly parameters for symmetric encryption are derived is discussed next.

**Double Ratchet** As noted in the introduction to this section, Open Whisper Systems devised an own cryptographic ratchet for use in their *Signal Protocol*, which they called *Double Ratchet*. This is because it combines two known concepts, the details of which are discussed below based on the official specification [Tre16b].

The fundamentals of the first ratchet should already be known, as it is based on the ideas of the *OTR* ratchet described in section 2.1.2. After performing some improvements to turn the 'three-step ratchet' into a 'two-step ratchet' [Moxa], it was included as *Diffie-Hellman Ratchet* into this scheme.

Like the name implies, shared secrets established through a *DH* exchange play a major role in the algorithm. Each user has a current *DH* key pair per session, called *ratchet key pair*. At session initialization, Alice has to generate this pair, but uses the *DH* key already used in the handshake as Bob's public ratchet key. When combined with *X3DH* handshake like in the *Signal Protocol*, the already used key is the signed pre-key. In order to forward the ratchet, Alice calculates the shared *DH* secret from the ratchet key she just generated, and Bob's public ratchet key. When she then sends a message to Bob, she simply includes her generated public ratchet key, and he can derive the same shared *DH* secret. In figure 2.9, the newly generated ratchet key is marked with a circle.

The next time Bob writes a message he generates a new ratchet key pair, uses it to calculate a shared secret with the public ratchet key Alice sent him before, and includes his new public ratchet key with the ciphertext. Alice repeats this procedure for her next message to Bob, and so on. In short, the sender's current public ratchet key is appended to every message. It is also appended when it was not just freshly generated but e.g. a second or third message is sent in a row, but more on that later. Note that unlike in the *OTR Ratchet*, where a *DH* key has to be 'confirmed' before it can be used, the *DH* key sent in a *Double Ratchet* message was already used to encrypt the containing ciphertext. Instead of operating on this shared *DH* secret  $s_{ij}$  directly, it is used in combination with the previously established root key. Both are given as input to the root key's derivation function  $KDF_{RK}()$  to produce the updated root key, and a chain key. In more detail,

$KDF_{RK}()$  is a *HKDF* which uses *SHA-256* as the hash algorithm, the current root key as salt, the *DH* secret as the input key material, and the info string *WhisperRatchet* to derive 64 bytes of data. The first 32 bytes are then set as the new root key, and the last 32 bytes are the chain key.

The term ‘chain key’ was used in a non-specific way as the sending and receiving chain keys are not updated at the same time, but one after the other. This is because one party’s sending chain key must be the other party’s receiving chain key, and vice versa. Therefore, the initiating party’s first chain key will be the sending chain key, and the receiving party’s first chain key will be the receiving chain key. It can be seen in figure 2.9, where the chain key derived from the root key with an odd index is Alice’s sending chain key and Bob’s receiving chain key, and after forwarding the root key once to receive an even index, it is Bob’s sending chain key, and Alice’s receiving chain key. As mentioned before, and also visible in figure 2.9, these chain keys are not directly used for deriving symmetric encryption keys, as there is another layer of indirection. This other layer is the other ratchet.

The concept of the second ratchet is described to be based on the SILENT CIRCLE INSTANT MESSAGING PROTOCOL (*SCIMP*) in a blog post [Moxa], but the specification points out this approach has been known for some time and is e.g. mentioned in [AB00] as re-keying based on an unbalanced tree.

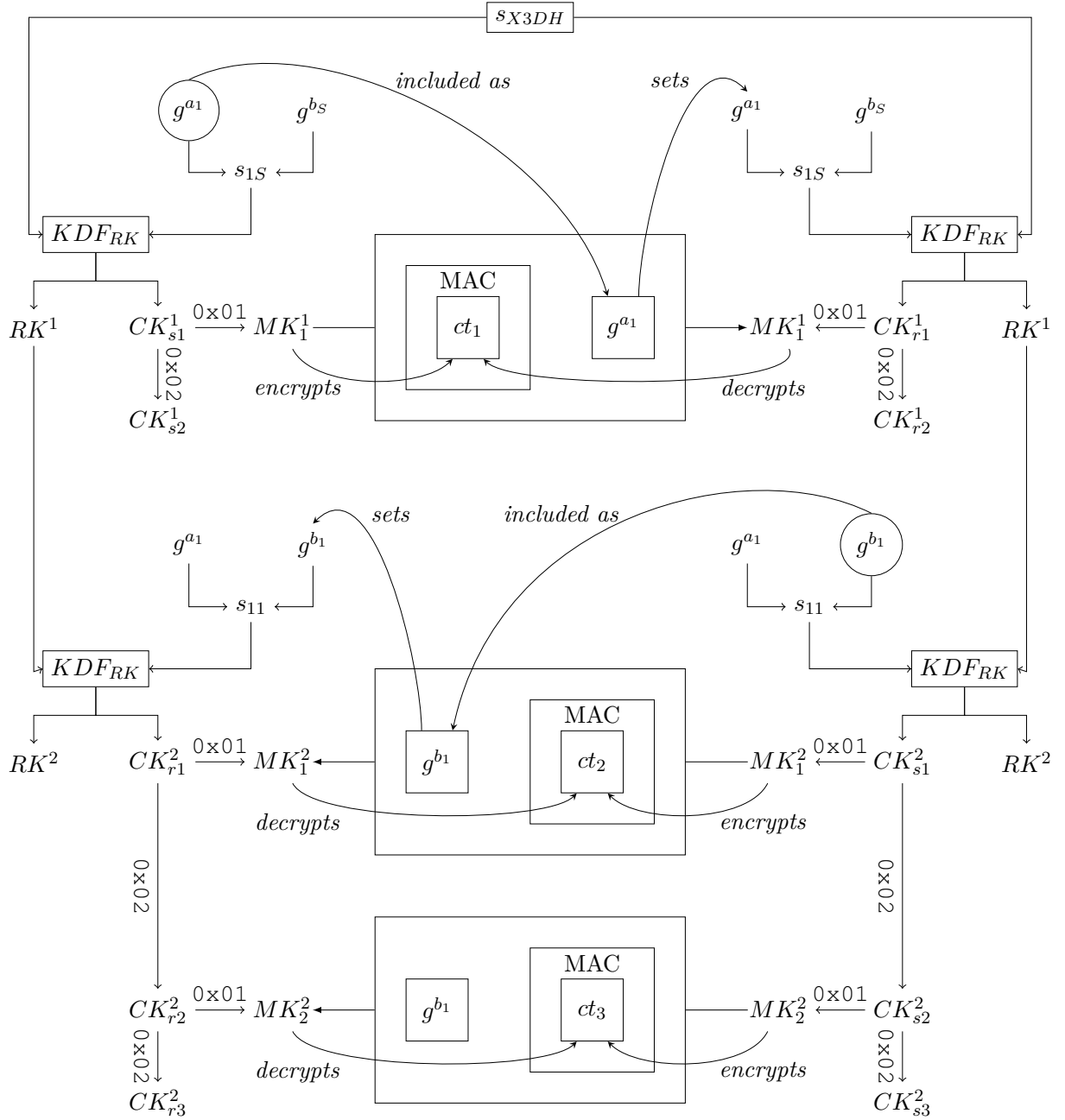
This ratchet is called *symmetric-key ratchet* and it basically just describes moving the current chain keys forward locally using a *KDF*. The *KDF* used here is called  $KDF_{CK}()$  and its output pair is calculated using a *HMAC* function. The first output of this *KDF* is the actual message key to be used for encryption. It is computed by using the current chain key as the *HMAC-SHA-256* key and the constant byte  $0 \times 01$  as input. The second output of the *KDF* is the next chain key, which is the result of the same current chain key as used for the message key, and the constant byte  $0 \times 02$ .

For every single message that is sent consecutively, i.e. without having received the conversation partner’s new public ratchet key to move the *DH ratchet* forward, the sending chain key is advanced in this *symmetric-key ratchet*. Naturally, the receiving side does the same to the receiving chain key in order to be able to derive the same message key. As a message key is not derived from an earlier one, it can be held on to without posing a security risk. In fact, this is what happens in order to deal with out-of-order messages. To facilitate this, the current message keys’s number in the chain is sent with the message, along with the length of the last chain which helps in case the *DH ratchet* was advanced by one of the missing messages.

Finally, in order to prepare the message for sending, another *HKDF* is utilized. Using *SHA-256*, a salt of 80 zero bytes, the info string *WhisperMessageKeys*, and the previously derived message key as the input key material, an 80-byte output is obtained, which is then divided into a 32-byte encryption key, a 32-byte authentication key, and a 16-byte initialization vector. The plaintext can then be encrypted with *AES-256* in *CBC* mode, initialized with the encryption key and IV.

By prepending to the ciphertext the associated data already used in the handshake, i.e.

Figure 2.9: The Double Ratchet.



a concatenation of both parties' public identity keys, the data over which the *MAC* is to be calculated with help of the authentication key is acquired. For the *MAC*, the specification suggests using a *HMAC* which reuses the same hash algorithm utilized for the *HKDF*, which in case of the *Signal Protocol* is *SHA-256*. Afterwards, the received tag is appended to the chained data, so that the final message consists of those three parts. These details are not contained in figure 2.9 for space reasons.

### 2.1.4 Other Protocols

It should be noted at this point that there are no other protocols which can seriously be considered 'state of the art'.

The *SCIMP* protocol whose ratchet previously mentioned as one of the inspirations for the *Double Ratchet* was discontinued and its specification can only be found in an archive [Vin12]. Silent Circle switched to the *Double Ratchet* in their *SilentPhone* app [Sil12]. *SafeSlinger* [FLK<sup>+</sup>13], *Threema* [Thr17] and *surespot* [sur] all use a *PGP*-like scheme for their end-to-end encryption, with varying ideas on how to replace the *WoT* for authentication.

*Telegram* offers so-called secret chats, using a standard unauthenticated *DH* exchange for establishing a shared message key, and re-negotiates it per 100 messages or every week for forward secrecy [Tel]. Their *MTPProto* protocol makes weird choices in regard to cryptographic primitives, leading to some theoretical vulnerabilities which make it fail the definition of *authenticated encryption* [JO15]. It is doubtful whether this protocol will find application outside of *Telegram*, and it is in fact not even implemented in all official clients.

The most interesting protocol is *Tox*, which tries to avoid issues related to using central servers by building a peer-to-peer network between the users. As for the end-to-end encryption, it uses the cryptographic primitives included in the NaCl library<sup>12</sup> [Toxb]. Even though reference implementations of the protocol and clients exist, the whole project is still in alpha stage and the documentation fragmented and incomplete [toxc]. Also, it is still missing features regarded as necessary in the context of this thesis, such as support for multiple devices [Toxa]. Therefore it currently cannot be considered for use, but this can change if it is further improved, as it looks very promising.

Some older academic publications which were never widely adopted can be found in [UDB<sup>+</sup>15].

## 2.2 Evaluation of Two-Party Encryption Schemes

### 2.2.1 PGP

As *PGP* was one of the first publicly available encryption tools, it did not have the possibility to have learned from others' mistakes. Considering it is still in use today, over 25 years later, it seems to be doing quite well. This is because it basically just

---

<sup>12</sup><https://nacl.cr.yp.to/>

describes a relatively simple format for applying cryptographic algorithms. During its evolution, it deprecated those that were discovered to be unfit, and added new developments, such as elliptic curve cryptography, in new or updated RFCs<sup>13</sup>. When weaknesses were found in the format of the specified data packets, such as in [KR02], they could also be corrected without breaking the underlying scheme. Being designed to be able to provide confidentiality, authenticity, or both, for a single message, it fulfills these goals.

However, the simple format is also the reason why it is questionable if these and other implied attributes of *PGP* fulfill the requirements of modern electronic communication. For instance, there is no built-in protection against replay attacks. As pointed out in section 2.1.2, some researchers believed *PGP* to be unfit so strongly they titled their paper on a suggested successor “Why to not use PGP”. Their main arguments are as follows [BGB04]:

Because new encryption keys are not automatically propagated to contacts, users rarely exchange them, as it is hard to notify other users of this change. In the worst case, a third party who acquired the keys of a user can read all incoming *PGP* encrypted messages. Considering the US NSA is officially allowed to retain any encrypted communication [Gle13], this scenario is generally not too unlikely, and can be seen to break the goals outlined by Zimmerman for his own country: protecting the right to a private conversation against the government [Zimb]. While the used cryptographic primitives are sound, government actors can be considered to have alternative means of acquiring keys and their passwords [Dec09, Chr09]. The suggested solution to this problem is adding forward secrecy as a necessary attribute, which the authors did for their own work, *OTR*. As the authenticity of messages is cryptographically proven by digital signatures, there exists cryptographic proof of authorship of a message. According to the authors, this attribute of cryptographic non-repudiability contradicts the goal of privacy. While this sounds correct when viewed superficially, in reality a cryptographic proof is rarely accepted as proof in real life, and definitely not in court, as a digital signature only provides a binding to a private key. In order to additionally bind this key to a person, further evidence is needed, which is considered to be very hard if not impossible to obtain [Mas12], and is therefore definitely not produced by *PGP* which does not contain this design goal. This argument against *PGP* can therefore be considered invalid.

A different point of criticism is usability. Admittedly, this is not unique to *PGP*, as can be seen throughout this thesis. However, while other schemes can and do try to improve this aspect by adding some degree of abstraction from terms like *keys* or *fingerprints*, *PGP* cannot be used without some understanding of fundamental concepts of public-key cryptography – something an average computer user does not have.

In the 1999 paper “Why Johnny Can’t Encrypt” [WT99], only 4 out of 12 users were able to complete a task consisting of encrypting and signing the same message for multiple members of an imaginary team in a timespan of 90 minutes. Similar papers followed, e.g. [SBKH06] (“Why Johnny Still Can’t Encrypt”), suggesting improvements. Still, 16 years

---

<sup>13</sup><https://tools.ietf.org/html/rfc6637>

later, in a paper named “Why Johnny Still, Still Can’t Encrypt” [RAZS15], the findings remain very similar: Out of 10 pairs of participants, only 1 managed to send encrypted emails among themselves.

While the papers may not be directly comparable as different software was used (PGP 5.0<sup>14</sup> and the Mailvelope browser plugin<sup>15</sup>) and the sample size was very small, the identified problems seem to be the same, and it does not seem probable results would differ much with more participants. As claimed in the beginning of the paragraph, these problems are the missing understanding of public-key cryptography, leading to a failure to understand the necessary procedure and resulting in mistakes such as generating keys for the recipient, or using the own public key for encrypting a message to someone else. This leads to another slight disadvantage of *PGP* when compared to the other presented schemes: As those include automatic session establishment, they are safe against passive adversaries, even though key fingerprints need to be compared for detection of active attacks. The manual session establishment of *PGP* can easily fail through ignorance, which results in plaintext being sent. An application’s user interface can mitigate this, however.

Although it is not part of the specification, an evaluation of *PGP* is likely incomplete without also inspecting the *Web of Trust*. It is considered to be the reason why *PGP* prevailed over the PRIVACY ENHANCED MAIL (*PEM*)<sup>16</sup> a standard which was never widely used, presumably as it required a hierarchical public-key infrastructure including certificate authorities to work [Roe10]). Nevertheless, more recent evaluations such as in [UHHC11] paint a sobering picture. The snapshot used in the paper was retrieved from the keyservers in December of 2009 and contained about 2.7 million keys. Out of these, only about 325.000 have or have made at least one signature. These remaining keys were represented as a directed graph, where an edge marks a signature, allowing for some interesting conclusions. For instance, the authors looked at the *strongly connected components* (*SCCs*), i.e. subgraphs which only contain members that have a path to each other.

Ideally, the *WoT* would be one large *SCC*. Instead, there are about 240.000, with over 100.000 of those consisting of just one node, further 10.000 of just a node pair, and the remaining ones having a size of between 10 and 100. The largest *SCC*, i.e. the set of users who can actually benefit from the *WoT*, consists of only about 45.000 keys.

A possible point of compromise is also noted: A not insignificant number of these well-connected keys uses *RSA* with key lengths that have been shown to be factorizable (768, about 500 keys) or are suspected to be factorizable by now or in the near future (1024, about 4000 keys) [KAF<sup>+</sup>10].

Looking at more up-to-date data, there are now about 4.5 million keys uploaded to the keyservers<sup>17</sup>, with the largest *SCC* consisting of about 60.000 keys<sup>18</sup>, which is still not

---

<sup>14</sup><http://www.pgpi.org/>

<sup>15</sup><https://www.mailvelope.com/de>

<sup>16</sup><https://tools.ietf.org/html/rfc1421> through 1424

<sup>17</sup>[https://sks-keyservers.net/status/key\\_development.php](https://sks-keyservers.net/status/key_development.php), accessed 2017-02-20

<sup>18</sup><http://pgp.cs.uu.nl/plot/>, with data from 2017-02-06



much better. Generally said, a new user will have to manually verify key fingerprints to have some trust in the key of another person who is not part of these 60.000 (e.g. as *MitM* protection), and it is hard to find a reason why this new user should not simply use a scheme which also requires manual verification and out-of-band authentication, but is a lot more comfortable to use and provides additional useful properties, as any of the other protocols presented in this thesis do.

After the rather critical evaluation, it should be noted that *PGP* does have properties other protocols do not have. The session establishment can reasonably be done by a human and does not require interaction by the recipient. Also, anything can easily serve as transport. As a result, it can easily be used in systems that were not made with confidentiality in mind or cannot provide it, such as websites. A good example are darknet markets – they provided a step-by-step tutorial for their users on how to use *PGP* to send confidential messages between buyers and vendors using their internal messaging system, e.g. for the disclosure of the delivery address [BB15].

Thus it can be still useful in the right setting, to which email messages unfortunately still belong, but attempts to employ it for instant messaging, such as the aforementioned *OX* for the *XMPP* protocol, seem misguided, and it is doubtful whether it will ever be actually implemented.

### 2.2.2 OTR

The first version of *OTR* had some shortcomings in its authenticated key exchange. These were already described in section 2.1.2, as they were applaudably fixed in the next version and the improved *AKE* has been part of the protocol since. Amusingly, the initial key exchange had worse properties than the “badly authenticated *DH*” used as a counterexample in the *SIGMA* protocol paper, as none of the received data is signed for confirmation [Kra03].

It seems though that the updated key exchange partly invalidated an initial design goal: adding deniability through forgeability. At first, the long-term public keys were sent without a signature, which made the attacks that were found possible, but also meant that the previously used *MAC* keys published in following messages could be used by anyone to fake a message. Now, these signed keys are only in possession of users who already established a session with their owners, severely limiting the group of possible forgers [Moxj].

In general, the *SIGMA-R* protocol was shown in [DRGK06] to be *partially deniable*, i.e. while because of the employed digital signature participation cannot be denied, the identity of the communication partner as well as the content of all following messages can, which is completely adequate. As discussed in section 2.2.1, it is generally questionable whether deniability is an important attribute for privacy, since cryptographic proof needs to fulfill many more requirements to be usable in real-world scenarios. (‘Deniability’ as used here means *deniable authentication* and is not to be confused with *deniable encryption*.) In addition, as noted in [DRGK05], it is enough for the key exchange protocol to be deniable, therefore using *SIGMA* made the publishing of *MAC* keys obsolete anyway.

Considering all this, the added protocol complexity and message size of publishing used *MAC* keys seems unnecessary.

Another vulnerability of *OTRv1* found in [DRGK05] has not been discussed yet, as it brought only a minor (yet important) change. Unlike previously described, the *MAC* key in *OTRv1* was simply a hash of the message key. On the one hand, this unnecessarily weakened the message key, as the *MAC* keys are published in the clear. On the other hand, and more importantly, this allowed an attacker to hijack a session if an ephemeral key is compromised, as it allows the attacker to also authenticate the injected messages. This broke the future secrecy the *DH* ratchet can provide until the next session is established, but was luckily fixed in *OTRv2*.

A further big change in the second version of *OTR* was the addition of the *SMP*. This is notable as it tackles the issue of both authentication and trust in a way that is still unique to it. *MitM* attacks on *OTR* were relatively easy to perform, as evidenced by a module for a popular *XMPP* server which automatically executes them<sup>19</sup>. In fact, this module was cited as part of the motivation to make authentication easier [AG07]. Manual comparison of key fingerprints, as it was possible from the beginning, can detect this type of attack. However, this again introduces the need to have some understanding of cryptography, and it naturally cannot serve as identity verification unless conversation partners have another verified channel, such as meeting in real life. The *SMP* not only abstracts away the cryptographic concept of long-term keys, highly increasing usability, but by employing a conventional shared secret also somewhat manages to tackle the problem of identity verification at the same time.

Unfortunately, in practice users would often agree on a “secret” in-band even after skimming instructions, disclosing the information to a possible attacker and rendering the method useless for both purposes [SYG08]. Still, it is an interesting approach and a step in the right direction, and as mentioned in the previous section, *OTR* in any case provides protection against passive adversaries, unlike *PGP*. From a cryptographic point of view, the specific solution to the *SMP* which is used in *OTR* was proven to keep its promises, i.e. no information except if the secret matches is revealed [BST01].

Following the addition of the discussed improvements to the second version of the protocol, *OTR* was again scrutinized, and more possible attacks were found in [BM]. However, the paper assumes an attacker to have control over the whole network, which is why the vulnerabilities are of a rather theoretical nature. Since the proposed additions cannot be found in *OTRv3* [OTR], which was released much later, it can be assumed the *OTR* developers agree with this statement.

The exception is the possibility of the ‘version rollback’ attack mentioned in the same paper, forcing users to downgrade to version 1 and therefore use a weak key exchange. While *OTRv1* was not explicitly deprecated, it was removed from the features an *OTRv3* client has to implement [OTR].

---

<sup>19</sup>[https://www.ejabberd.im/mod\\_otr](https://www.ejabberd.im/mod_otr)

Now, the underlying models and assumptions will be inspected. Unlike *PGP*, which was made with emails in mind [Zimb], *OTR* was explicitly designed for instant messaging. Its authors realized from the start that their synchronous model of communication is incompatible with the asynchronous medium of electronic mail [BGB04], so it is likely their call to “not use *PGP*” should be rather seen as an appeal to re-evaluate its definition of privacy, especially as attempts of embedding it in instant messaging were being made. Examples of such experiments from around that time can be found in the *gaim-e* plugin<sup>20</sup> for the *AOL*, *MSN* and *Yahoo* protocols, or an extension for the *XMPP* protocol<sup>21</sup>. The original paper suggests automatically terminating *OTR* sessions at every client exit and after some period of inactivity, as well as implementing special *NAK* (NEGATIVE ACKNOWLEDGEMENT) messages which are automatically sent to renegotiate a session in case the session was terminated only on one side and an unreadable message was received. Additionally, the last sent message should be saved as plaintext so it could be resent in the new session. This complexity could have been an indication that modeling instant messaging conversations as synchronous is not optimal, and the only reasoning for establishing a new cryptographic session for every period of activity in a messaging session can be found in possible *MitM* detection by comparing the sent long-term public keys. However, as already mentioned, this automatic type of detection only works if the *MitM* was not performed every time, which however is not unlikely in case of a malicious server administrator. Furthermore, as (ideally) every message is encrypted with a new key through the employed ratchet, frequently re-establishing sessions does not improve forward secrecy. In the end, *OTR*’s conversation model led to degraded usability and a need for workarounds a few years down the road, when messaging applications on mobile platforms started implementing *OTR*. A few examples can be found in a blog post by Open Whisper Systems, explaining why they felt it was necessary to move away from *OTR* for *TextSecure* (on iOS) [Moxc]: Before sending an encrypted message, a session has to be established synchronously, i.e. both users’ clients have to be able to react to it. What is a given on desktop operating systems, need not be on mobile ones. For reasons such as battery and data traffic conservation, programs are ‘paused’ when sent to the background by a different application, limiting their capacity to react to events such as incoming messages. This is especially true when the screen is turned off, as then only ‘push notifications’ are received, which means that the device is only notified of a message and has to fetch the actual message contents when it is reactivated. So when a user of *TextSecure* with *OTR* received a handshake message while not actively using the phone, it was actually just a push message and the client had to wait until the phone was unlocked. Another example given is *ChatSecure* (also on iOS), which at the point of writing of the blog article silently ended the encryption session after the program has been in the background for two minutes, and equally silently discarded any further messages. In *OTRv3*, which was released at the end of 2012, sessions of indefinite length were also allowed [OTR], likely fixing the worst of these problems as the session only had to be established once. However, that was not enough.

---

<sup>20</sup><https://sourceforge.net/projects/gaim-e/>

<sup>21</sup><https://xmpp.org/extensions/xep-0027.html>

The *OTR* developers also did not consider the possibility that a user owns multiple devices and wants all of them to receive all messages. To be fair, this feature was not widely supported by popular IM protocols at the time *OTR* was conceived. Still, this requirement became more significant with the widespread use of mobile devices, as cited as motivation for this work in the introduction. This fact was partly acknowledged by the developers, as *instance tags* which permanently identify a client were also added in *OTRv3* [OTR]. With this, the sender at least has the possibility to choose the recipient device of an encrypted message, while before the result was implementation-dependent. The result still cannot be called ‘multi-device support’, though, as this does not help with multiple *own* devices, which in the best case end up with all incoming but none of the outgoing messages. Actually, the *OTRv3* specification does consider protocols that deliver messages to multiple devices at the same time, but only insofar as to suggest using the previously mentioned indefinite sessions for interleaving incoming messages from multiple sources.

To summarize, the *OTR* developers reacted to the emerging use of instant messaging with the right ideas, as can be seen informally proven by the long list of programs which make use of it to this day. Taking away most of the responsibility of key management from the user was also certainly a good design decision and a step in the right direction. This improved usability and thus actually enabled users to have private conversations, in contrast to *PGP*, whose perceived complexity seems to have hindered its adoption by less adept users. However, the developers missed out on modernizing their protocol by adopting it to newly surfacing requirements. The rather minimal fix which resulted in version 3 could have been further improved on, but to this day was not.

### 2.2.3 Signal Protocol

As already pointed out in the introduction to section 2.1.3, the whole reason for the *Signal Protocol* to exist is because its developers wanted to overcome the shortcomings of *OTR*, which is similar to the relationship between *OTR* and *PGP*. Therefore this evaluation will start by looking at what they think was missing, as well as how the improvements were achieved with this protocol. A good basis for this are the explanations given by them in a series of blog posts, some of which were already cited in the evaluation of *OTR* in the preceding section [Moxj, Moxa, Moxc].

One of the main reasons driving Open Whisper Systems away from *OTR* is its synchronous conversation model. Messaging on mobile devices came to replace *SMS*, which is an asynchronous medium, much like email. People just message away and do not expect and usually also do not receive an immediate response. Requiring the two conversation partners to be either active at the same time, or to wait several messages before any ciphertext can be exchanged can certainly be counted as bad usability. As said before, *OTR* was later updated to allow indefinitely long sessions, so the cumbersome session establishment only has to be done once. However, this is arguably still one time too many.

By employing the ‘cached one-round key exchange’ protocol  $(X)3DH$ , this major obstacle could successfully be overcome. This, of course, adds a dependency on a server, which does not matter to the creators of the *Signal Protocol*, who run their own messaging service. The stated design goal of *OTR* to work ‘on top of another protocol’ [BGB04] can be seen to imply the opposite. However, as it is trivial to convert an asynchronous scheme like this into a synchronous one, it should not necessarily have hindered *OTR*’s development, while allowing additional comfort on flexible protocols which allow working around this restriction.

Another argument brought up against *OTR* is the complexity of its message format. In order to perform a new  $DH$  calculation to receive a new shared secret, a client first announces the public key it is going to use, and then waits for it to be acknowledged by the conversation partner. This makes it necessary to include all key IDs in the messages: that of both currently used keys and the newly announced key. Not to forget the *MAC* key publishing scheme to allegedly add to the deniability by appending previously used keys to messages as plaintext. Both of these mechanisms increase the size of the message that is sent. While it might not be much of an issue nowadays, it is surely still nice to avoid unnecessary traffic, and as explained further below, the *Signal Protocol* certainly does make this unnecessary.

As a reminder, one of the main concerns of *OTR* is deniability, which is the whole reason for publishing the *MAC* keys, even though the usefulness of this procedure is questionable. In the *Signal Protocol*, this attribute was planned to be achieved from the start by the nature of the key exchange, as in fact the way the session is established is the most important part to achieving this attribute. This fact was briefly discussed in section 2.2.2, but the following citation from [DRGK06] reinforces it: “If the parties can deny having exchanged a key with the other party, then the rest of the communication can also be denied.”

According to [CGCD<sup>+</sup>16], the used  $X3DH$  protocol likely fulfills the *peer-deniability* attribute defined in [CF11], which is very similar to the *SIGMA* protocol’s *partial deniability*. As no ‘peer-provided element’ such as a nonce is signed in  $X3DH$ , it possibly also fulfills the slightly stronger *peer-and-time deniability* property also defined in [CF11], though no proof is given. This type of deniability not only allows denying the intended peer, but also the time window of the exchange. The counterexample for the latter given for *SIGMA* is a hash of a daily newspaper as the peer-provided element.

Curiously, the simple  $3DH$  as suggested in [KP05] is *fully deniable* in the sense of [DRGK06] [Moxj], as it did not employ digital signatures at all. Intuitively, this is because it is possible for Eve to fabricate a whole fake conversation by using Alice’s advertised long-term public key, but then replace the public ephemeral key with one she computed herself [Moxj, FMB<sup>+</sup>14].

However, as stated in [Kra05], this class of implicitly authenticated protocols cannot provide ‘full’ forward secrecy. This is because the following attack described in [CF11] is possible: If Eve replaces Alice’s ephemeral public keys with her own when they are requested by Bob (e.g. by substituting them on the server), she can decrypt messages

sent from Bob to Alice if she manages to learn Alice’s long-term private key at any later time. It should be noted that Alice cannot decrypt these messages which makes this attack very obvious and likely short-lived, however it should not be acceptable by design. Note the similarity to the previously described attack which was used to prove full deniability. As a remedy, the paper thus suggests signing the ephemeral public keys, which was incorporated into *X3DH*.

Then again, the inclusion of signatures re-enables the *key-compromise impersonation* attack for *X3DH* which empowers an attacker who compromised the signed pre-key’s private key to impersonate anyone to the victim, which is one of the reasons it is recommended to regularly replace it [Mox16]. This type of attack was excluded for the protocols underlying *3DH* [BWJM97], but also for the more interactive *SIGMA* protocol which requires the partner to generate a new signature and therefore provide proof of knowledge of the private key [BMP04]. These comparisons provide an interesting demonstration of the inherent tradeoffs between the key attributes authentication, forward secrecy, and deniability [DRGK06, CF11, BMP04].

An explicitly stated design goal of the *Signal Protocol* is future secrecy [Moxa], which it fulfills according to [UDB<sup>+</sup>15]. Comparing the attribute’s definition given therein (and used in this thesis) to the one given by the developer of the *Double Ratchet*, [CGCG16] reaches the conclusion that future secrecy is implied by forward secrecy anyway. However, both papers point out that future secrecy is not well defined.

[CGCG16] then introduces the stronger notion of *post-compromise security*. Unlike future secrecy, post-compromise security explicitly also covers the scenario of an attacker gaining access to long-term keys, but not all memory contents, which limits the attack window to a timeframe directly after the compromise, i.e. until “an honest communication exchange” occurs. One of the central mechanisms in *PCS* is a shared state, which is included in the *Signal Protocol* through use of the root key used by the *Double Ratchet*. However, while *PCS* seems to be achievable, the implementation in the *Signal* application is not, as a user’s long-term key is shared across multiple devices, but the *Double Ratchet* sessions are not, being established between devices individually. This allows an attacker who compromised the long-term key to impersonate a new device at any later time and thus gain access to conversations even if their state has been updated in the meantime. Since future secrecy is described as the main reason for including the symmetric ratchet in the *Signal Protocol* [Moxa], it seems important to point out what exactly was achieved by doing that. As the author notes in the cited blog post, *OTR* already achieves future secrecy through its *DH* ratchet. Yet, if a user writes multiple messages in a row, they are encrypted by the same message key, which means that in case of key compromise, they all can be decrypted. The counterexample used is the *SCIMP* ratchet, which hashes the key to receive a new key for every message (but of course does not provide future secrecy). So by incorporating the symmetric ratchet, the *Double Ratchet* minimizes the amount of messages whose confidentiality has been broken in case of ephemeral key compromise, and therefore has better future secrecy properties.

The specifications of the *Signal Protocol*'s several parts point out some attack vectors mostly related to key compromise as well as some security considerations [Tre16b, Mox16, Tre16a], however multiple papers examining the *Signal Protocol* generally attest its security in realistic scenarios [FMB<sup>+</sup>14, CGCD<sup>+</sup>16]. A flaw pointed out by [FMB<sup>+</sup>14] is the susceptibility to a *UKS* attack, but as previously discussed there is a tradeoff with deniability and since the specification itself warns about this attack [Mox16] this seems to be the result of a decision on this tradeoff. Furthermore, [CGCD<sup>+</sup>16] suggests slight improvements which would result in resilience against compromise of the random number generator. This is interesting as one of the reasons for dropping signatures for *3DH* was getting rid of the complexity that comes with *DSA* [Moxj], citing an example of a bad – i.e. static – *PRNG* leading to disclosure of keys [fai10], which of course generally extends to predictable values [Nat10, H D08].

As for the usability, purely on the protocol level it is the same as *OTRv1*, i.e. manual fingerprint comparison, and therefore highly depends on the implementation. That being said, the *Signal* app introduced so-called *safety numbers* to move away from the concept of ‘fingerprints’, as this metaphor does not seem to work well for the average user without any background in cryptography [Moxg]. These safety numbers consist of 60 digits (in 12 groups of 5) per conversation in order to decrease the needed comparisons to just one. Actually, this number still just contains a representation of both public keys, one being 30 digits long. In order to increase resistance against *UKS* attacks on the UI level, the phone number is included in the generation of these numbers as well.

Generally, using the phone number for identity establishment can be seen as user-friendly, as someone usually already knows the phone numbers of his or her friends. However, this decreases anonymity.

In 2016, the *Signal Protocol* was adopted by the *WhatsApp* messenger, the *Facebook* messenger, and Google's new *Allo* messenger with their many millions of users [Moxk, Moxb, Moxe]. It can therefore be considered the de-facto standard, and this in turn can be treated as an informal proof that it did many things right.

#### 2.2.4 Summary

This section features a more direct comparison between all three protocols as a summary of the preceding evaluation. Table 2.2 shows which of the initially mentioned attributes were achieved attributes in form of a rating of maximum three stars, while table 2.1 provides an overview of other attributes which cannot be quantified.

First, some commentary on table 2.1.

The first three rows are more or less related: Choice of main medium and communication model are directly associated with each other, and of course depended on what was current at the time. Electronic mail is naturally asynchronous, as well as mobile IM, which can be seen as a more flexible replacement of the equally asynchronous SMS. Nowadays, Desktop IM is not necessarily synchronous, but e.g. support for ‘offline messages’, i.e.

Table 2.1: General attributes of the evaluated protocols

| Protocol<br>Attribute    | <i>PGP</i>                       | <i>OTR</i>                       | <i>Signal Protocol</i>                               |
|--------------------------|----------------------------------|----------------------------------|--|
| Main Medium              | e-mail / BBS                     | desktop IM                       | mobile IM  |
| Communication Model      | asynchronous                     | synchronous                      | asynchronous   |
| Session Establishment    | non-interactive<br>(2-step)      | interactive<br>(4-step)          | non-interactive<br>(2-step)                          |
| Session Length           | single message                   | IM session<br>(both online)      | indefinite   |
| Multi-Device             | yes                              | no                               | yes  |
| Identity                 | long-term key                    | IM account                       | IM account<br>(phone #)                              |
| Trust Establishment      | WoT                              | FP verification<br>or <i>SMP</i> | FP verification<br>(safety numbers)                  |
| Cryptographic Primitives | selectable from<br>given options | fixed                            | chosen per imple-<br>mentation from<br>given options |

Table 2.2: Cryptographic attributes of the evaluated protocols

| Protocol<br>Attribute | <i>PGP</i> | <i>OTR</i> | <i>Signal Protocol</i> |
|-----------------------|------------|------------|------------------------|
| Confidentiality       | ★★★★       | ★★★★       | ★★★★                   |
| Integrity             | ★★★★       | ★★★★       | ★★★★                   |
| Authenticity          | ★★★★       | ★★★★       | ★★★★☆                  |
| Deniability           | ☆☆☆☆       | ★★★☆☆      | ★★★★☆                  |
| Forward Secrecy       | ☆☆☆☆       | ★★★★☆      | ★★★★                   |
| Future Secrecy        | ☆☆☆☆       | ★★★☆☆      | ★★★★                   |

store-and-forward, was only starting to be added to IM protocols around the time *OTR* was conceived. As an example, see the *Message Archiving*<sup>22</sup> *XMPP* extension, whose initial draft was published just the same year as *OTR*. This of course set the frame of what kind of session establishment is possible. Note that even for an interactive key exchange, 4 steps is still 1 step more than necessary, but the 4-step *SIGMA-R* protects the responder's identity against active attacks by delaying sending identifying information until the initiator's identity has been verified in the third step. 3-step variants which either protect the initiator's identity by adding encryption (*SIGMA-I*), or come with no protection at all (regular *SIGMA*) also exist [Kra03].

The session length however is not bounded by this and was set by the developers in

<sup>22</sup><https://xmpp.org/extensions/xep-0136.html>



accordance with their model of communication.

For multi-device support it is of course imaginable to simply establish pairwise sessions, but while *OTRv3* added per-client ‘instance tags’ for identifying recipient clients, the official *OTR* implementation only lets the user choose one of them to encrypt for. In addition, and most importantly, it does not account for multiple *own* devices on which the user might want to receive a copy as well for a consistent view of the conversation. The *Signal Protocol* reference implementation specifically includes per-device sessions per user, including the own. *PGP* trivially allows to encrypt the symmetric per-message session key for any number of recipients using each of their long-term public keys.

Initially, the idea behind *PGP* was that everyone generates one long-term key to be identified by, as the scheme is not constricted to a certain medium and can be used for many things. The developers of *OTR* found this to not fit in their definition of *privacy*, instead binding identity to IM accounts, which can be and usually are registered under a pseudonym. A similar approach is used in the *Signal Protocol*, though it should be noted that the *Signal* app uses phone numbers for registering accounts, subtracting from the provided anonymity.

In turn, this makes establishing trust somewhat easier, as a phone number is generally known by a person’s friends. Additionally, there’s the previously mentioned safety numbers for both *MitM* and *UKS* detection. The *Signal Protocol* by itself does not specify this and is left with simple fingerprint verification however. *OTR* also employs manual fingerprint verification, but added the *SMP* which can also make use of a shared real-life secret. *PGP* relies on the *Web of Trust*, which was lengthily discussed before.

Lastly, the protocols also differ in how they deal with the choice of cryptographic primitives. In its specification, *PGP* offers a choice of algorithms for each function, which implementations can decide to support. Naturally, an application striving for good interoperability should then implement most of these. In contrast, *OTR* fixed all primitives in their protocol specification. The *Signal Protocol* does something inbetween: The specification offers a choice of algorithms for specific parts, out of which a decision has to be made per implementation. Actually, there is no specification for the whole protocol, just for its parts which in theory do not necessarily have to be used together.

As for table 2.2, ratings like these are of course debatable, but the detailed reasoning can be found in the preceding sections. A short summary explaining the exact rating follows. The ratings are supposed to be based on an average realistic scenario, and not on either the worst or the best case.

Since all three protocols use solid and widely reviewed cryptographic algorithms for encrypting the payload, they get an accordingly high rating for confidentiality. However, as noted in the *PGP* evaluation, its wide range of choices unfortunately also allows long-term keypairs that can be considered compromisable, leading to a slight reduction. *OTR* and the *Signal Protocol* with their more rigid choice of algorithms do not suffer from this.

However, the latter loses half a star in its authenticity rating because the used *X3DH* key exchange algorithm is vulnerable to both *UKS* and *KCI* attacks, unlike *OTR*’s *SIGMA*

protocol. The specification points these out though and gives useful pointers on how to mitigate them on the application level, therefore the impact can be considered not severe. The half star the *Signal Protocol* loses against *OTR* in authenticity it gains again in deniability. While neither can reach full deniability while still providing authenticity, the *X3DH* handshake does not demand a user-supplied value and is therefore argued to still achieve a higher level of deniability. *PGP*, which unlike the other two protocols employs digital signatures instead of *MACs*, does not provide deniability at all.

Neither does it provide forward secrecy, as it does not force the long-term encryption keypair to change in any way. *OTR* specifically wanted to advance from this behaviour and its ratchet does a good job at regularly changing the encryption key. Nonetheless, the *Signal Protocol* managed to slightly improve on it and use a new symmetric key for every single message, even if the conversation partner does not reply, i.e. no new *DH* exchange can be completed.

Even though not explicitly stated as a design goal, the *OTR* ratchet also manages to achieve some form of future secrecy. The *Signal Protocol* did state this goal, but only in the definition that *OTR* already fulfills. As it turns out, it manages to achieve resilience against even more attack scenarios, and therefore receives the full rating. Since the reference implementation in the form of the *Signal* app was brought to comparison throughout the evaluation as well, it should be noted again that it does not achieve this *post-compromise security* through its handling of multiple devices, though possible in theory. With the same reason as before, *PGP* does not provide any future secrecy.

It seems like tables 2.1 and 2.2 manage to visualize the general feeling one got about the reasons for why one protocol superseded the other in the medium most used at the time: *OTR* obviously has properties that *PGP* does not have, but which can be counted as an integral part of ‘privacy’. The *Signal Protocol* on the other hand does not significantly improve in this regard. While it does seem to more or less optimize the ideas behind the conception of *OTR*, the reasons for its huge success seem to lie mainly in other areas, namely removing the restrictions of a synchronous model of communication and the limitation to one device per user.

Seeing how the *Signal Protocol* is the de-facto standard, it appears it a reasonable starting point for new developments in the area of instant messaging. Closer inspection revealed that it in fact has some best-case properties and fulfills other modern requirements, while other attempts at providing these appear to be absent.

## 2.3 Overview of Multiparty Encryption Schemes

### 2.3.1 Two-Party Scheme Reuse

After *OTR* gained some traction, a first attempt was made to apply it to the groupchat usage case in [BST07]. Called *Group OTR*, its concept is based on using one of the participants as a ‘virtual server’ which has an active session with every member of the group. All other participants only need a session with this virtual server, as it will

re-encrypt the message for everyone else after decrypting the received ciphertext.

In their implementation, the ‘virtual server’ role is filled by the initiator of the group chat. The *OTR* message format is extended to also contain the intended receiver, and original sender.

### 2.3.2 Pairwise Sessions for Key Transport

#### Per-Message Key

Assuming each participant of a group can establish a secure channel with every other participant, likely the most straightforward way to implement secure group communication is to use these individual sessions for every message. However, instead of encrypting the whole message for all recipients individually and then sending the ciphertext to each of them, the following technique is usually applied:

A symmetric-key algorithm is used to encrypt the plaintext with a random key, and the previously established session with every recipient is used to only encrypt this key. Since the key is likely shorter than the plaintext, this technique saves computation time, but also saves the sender from having to transmit the encrypted message to every user individually. Instead, one message which contains the encrypted key for each user is sent to the server, which delivers it to all users as usual. A receiving client then needs to identify which of the encrypted keys it can decrypt, and finally decrypt the payload, containing the message text.

This scheme should sound familiar from *PGP*, where it is for example used by *GnuPG*, but it is also used by other programs for their multi-party chat implementations, such as the *Signal Messenger* [The, Moxf].

#### Per-Session Key

Instead of using a pre-existing secure channel with each other group member for relaying single message keys, it can also be used to transmit other key material once. This can then be a session key which is the valid for the remainder of the group session, or until replaced).

One possible version of this mechanism is one of the participants generating a key to be used by the whole group, and transmitting it to everyone. Such a scheme is e.g. proposed in the draft of *flute* [Geob], where a ‘room captain’ is responsible for generation and distribution of a group key.

A similar technique is for every user to generate an encryption key and then announce it to every other user. This strategy is part of the *Signal Protocol* reference implementation, but as just mentioned, the *Signal Messenger* itself employs a different mechanism [Opea, Moxf]. However, it is used by e.g. *WhatsApp* group chats, and is also available as *Megolm* in the *Olm* implementation of the *Signal Protocol* [Wha, mat]. The following discussion is based on their documentation.

When a new member joins the group, the client needs to generate a new, random secret, as well as an ephemeral signing key pair. The first outgoing message needs to contain

this secret, which will be used to derive encryption keys for the following messages, and the public signing key used for verifying authenticity. This information is encrypted for each participant using the existing two-party session. Short-lived digital signatures are needed because in a multi-party setting like this, *MACs* computable by all participants obviously cannot authenticate a specific user.

For each subsequent message, the existing secret is used to derive both a new secret and an encryption key, similar to the *symmetric-key ratchet* described in the context of the *Signal Protocol*. As before, the secret can be advanced in the same way on the receiving end to derive the used message key.

### 2.3.3 Use of Group Key Agreement

#### *(n+1)sec*

A first attempt to create an explicit multi-party instant messaging protocol which can work on top of other protocols and does not require the IM server's assistance was made in [GUVGC09] and is called *mpOTR*, for MULTI-PARTY OTR. Despite its name, it does not employ *OTR*, nor is it technologically similar. Rather, both protocols share many design goals (and one inventor).

Unlike *OTR*, *mpOTR* neither came with a ready-to-use library and plugin, nor was the specification detailed enough for the creation of one. For this reason, several projects wishing to implement end-to-end encrypted group chats started working on *(n+1)sec* [eQua]. The basic mechanism of this protocol is based on *mpOTR*, however several improvements were made along the way, especially by choosing algorithms for those parts where none were defined, or some improvement was necessary. At the moment of writing, only incomplete drafts of the specification exist in several places [eQua, eQub], however there is enough information available for the following high-level overview.

As a first step and before any group computations can be done, all users need to generate an ephemeral keypair and broadcast both its public key  $y_i = g^{x_i}$  and a long-term public key  $U_i = g^{x_I}$  to all other members. The ephemeral keypair will in fact be used for three distinct purposes: peer-to-peer handshake, group key agreement, and digital signatures for origin authentication.

Establishing the peer-to-peer authenticated channel is the next step. *(n+1)sec* follows the suggestion made in the *mpOTR* paper and uses an implicitly authenticated *DH* key exchange variant – the *Triple Diffie-Hellman* handshake described in section 2.1.3. It is ‘implicitly authenticated’ because the proof of authenticity happens through using the shared secret. In *mpOTR* it is suggested to use a key derived from the shared secret to encrypt the transmission of the ephemeral signing key, but as *(n+1)sec* only uses one ephemeral keypair in total, it uses an authentication challenge.

This challenge happens by sending a nonce  $N_{i,j}$ . A recipient Alice of such a challenge needs to compute the shared secret  $S$  using her own and the received keys, concatenate it with her own public key and the nonce, and return a hash of that data  $T = H(U_a|N_{b,a}|S_{a,b})$  to the sender Bob. If Bob gets the same result for this calculation, Alice has proven

knowledge of the secret  $S$  to him, and by extension also knowledge of the ephemeral private key  $x_a$ . All further messages, i.e. those needed for the protocol run as well as the ones containing user-written ciphertext, are then authenticated by a digital signature using this short-term key. How exactly the ephemeral signing keypair (*Ed25519*) is derived from the ephemeral *ECDH* keypair (*Curve25519*) is not specified.

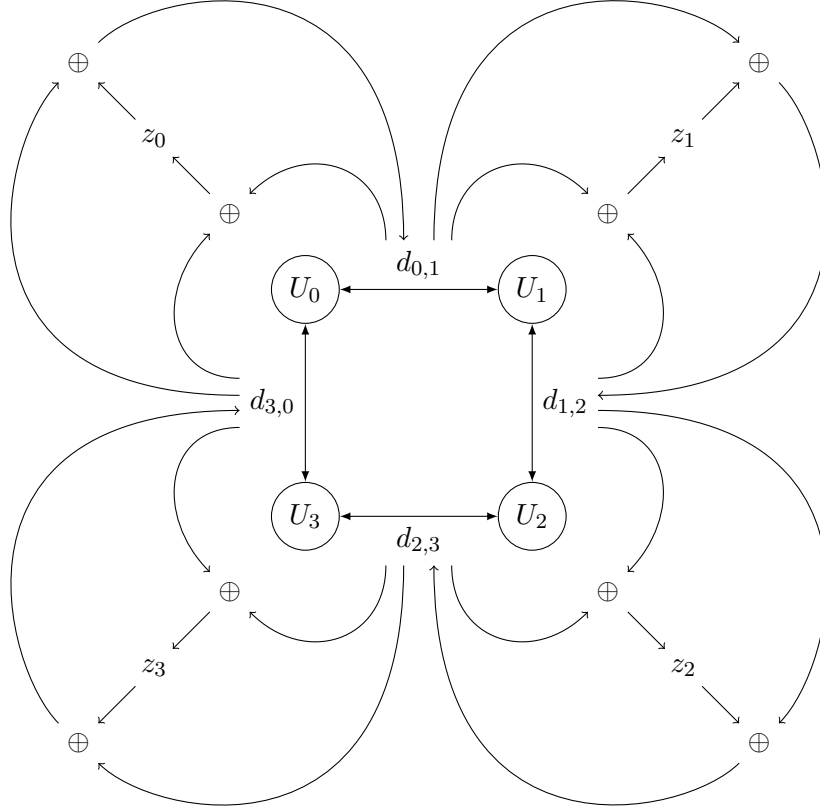
Once every group member confirmed the authenticity of every other member, the actual group key agreement can immediately begin, as the needed  $y_i$  and  $U_i$  have already been broadcasted. The basis for the key exchange scheme used in  $(n+1)\text{sec}$  is the *mBD+P* protocol proposed in [ACMP10], which in turn is a modification of the *Burmester-Desmedt* group key exchange protocol described in [BD95] – hence the name. Based on adjustments suggested in [Man09], it allows for reusing the public *DH* key  $y_i$  for both group and peer-to-peer secrets in order to decrease the number of communication rounds. Thus, the shared secret  $S$  resulting from the previously described *3DH* exchange is not  $s_{i,j} = H(g^{x_i x_j} | g^{x_i U_j} | g^{x_j U_i})$ , but a derived value  $S_{i,j} = H_p(s_{i,j}, U_i | y_i, U_j | y_j)$  in order to prevent collision attacks.

Before any cryptographic operations are performed, all  $n - 1$  user identities, represented by a concatenation of the username  $u_i$  and the long-term public key  $U_i$ , are again concatenated in order and then hashed to receive the **groupid**:  $groupid = H(u_0 | U_0 | \dots | u_{n-1} | U_{n-1})$ . The index stops at  $n - 1$  because the group members form a circle, i.e.  $u_0 = u_n$ .

Now, every user  $U_i$  can calculate the shared secrets  $d$  with both neighbors by calculating the *3DH* secret  $s$ , concatenating it with the **groupid**, and hashing the result to receive  $d_{i-1,i} = H(s_{i-1,i} | groupid)$  and  $d_{i,i+1} = H(s_{i,i+1} | groupid)$ . With these values, a user can calculate his share  $z$  of the group secret  $g$  by XORing them. Afterwards, this value  $z_i = d_{i-1,i} \oplus d_{i,i+1}$  is again broadcasted to all other users, together with the calculated **groupid**. Having received all  $z_{0..n-1}$  from the other group participants, a user  $U_i$  can calculate all other shared secrets. Starting from  $d_{i+1,i+2} = d_{i,i+1} \oplus z_{i+1}$  which can be derived from a shared secret  $U_i$  already possesses and a received value  $z_{i+1}$ , all values can be computed in a similar manner, i.e.  $d_{i+2,i+3} = d_{i+1,i+2} \oplus z_{i+2}$ , and so on. Hashing the concatenation of all  $d_{i,j}$  then yields the group secret  $g = H(d_{0,1} | \dots | d_{n-1,0})$ , from which an encryption key can be derived. This can be observed in figure 2.10.

In order to achieve forward secrecy, the initial draft at [eQua] suggested an *OTR*-like ratchet of users announcing new ephemeral keys with either their regular text messages or special heartbeat messages in order to periodically refresh the group secret. The newer specification draft does not contain such a scheme [eQub]. Instead, re-keying can be explicitly triggered by sending a **KEY\_RATCHET** message if the group key has been in use for too long. Generally, the group encryption key is renegotiated at every join or leave, which in any case provides at least per-session forward secrecy.

Furthermore,  $(n+1)\text{sec}$  tries to deal with denial-of-service attacks. Aside from timing out unresponsive parties, some adjustments were made to detect malicious behaviour so the offender can be excluded in a subsequent run of the protocol. For instance, a

Figure 2.10: *np1sec* group key agreement between four users.

malicious user  $U_m$  broadcasting a wrong **groupid** together with  $z_m$  will abort the protocol run, and cause a re-run excluding him. There is also another step after computing the group secret  $g$ : Broadcasting the value resulting from  $H(g|\text{groupid})$ . If all users sent the same value,  $g$  can be used. Otherwise, everyone has to publish their ephemeral secret key  $x_i$ , so that all calculations made can be reproduced, and the offending party be excluded in the re-run that follows.

### Group OTR

Unlike the scheme with the same name described in section 2.3.1, *Group OTR* as presented in [LVH13] does not try to reuse *OTR* for a group setting. Rather, it simply shares *OTR*'s design goals, just like *mpOTR*. Moreover, *GOTR* is meant to improve on *mpOTR*, which according to the authors does not provide the same level of repudiability as *OTR*. However, again very much like *mpOTR*, the *GOTR* proposal does not include a specification of cryptographic primitives, and merely a suggestion regarding a necessary

two-party key exchange protocol. As there is no protocol to inspect, this section only contains a high-level overview over the nonetheless interesting scheme.

*GOTR* is based on the *Burmester-Desmedt* group key exchange protocol, whose basics were already described in section 2.3.3. Since a non-modified version is used here, the mathematical operations differ from the description in that section, but this is not a concern for a general overview. It is still the case that a participant calculates his or her share of the *BD* group key by using the two neighbors' public keys, and these shares can be used by every participant together with their private key to compute the same group secret. Therefore, like previously described, each user needs to establish an authenticated channel to every other user first. After that is done, an ephemeral *DH* keypair is generated, and its public value broadcasted. However, *GOTR* does not treat one user as one participant – each user  $U_i$  acts as two virtual participant nodes  $D_{ij_0}$  and  $D_{ij_1}$  for every other user  $U_j$ , as can be seen in figure 2.11. Thus, each node  $D_{ij_0|1}$  has its own ephemeral *DH* keypair, resulting in two public values per every other user.

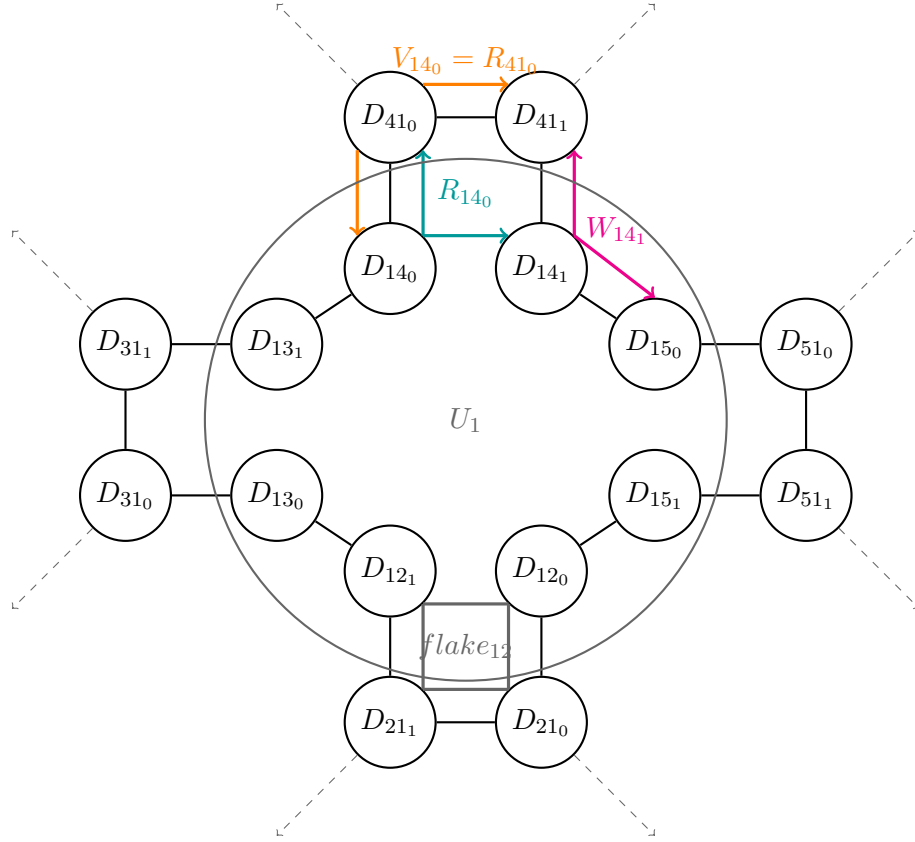
After receiving the other users' public *DH* keys, the first step for  $U_i$  is then to calculate the two own shares  $R_{ij} = (R_{ij_0}, R_{ij_1})$  of a *BD* group key between the four virtual nodes created between  $U_i$  and every other user  $U_j$ . The computed  $R_{ij}$  values are then broadcasted by every user, which means that the corresponding values  $V_{ij} = R_{ji}$  are also received. In theory, the *BD* group key of these four-node groups called *flake<sub>ij</sub>* could be calculated and used to secure further communication between  $U_i$  and  $U_j$  – this is not the main point of the algorithm however.

Instead,  $U_i$  also computes the *BD* shares  $W_{ij_0}$  and  $W_{ij_1}$  between  $(D_{ij_0}, D_{ji_0}, D_{i(j-1)_1})$  and  $(D_{ij_1}, D_{ji_1}, D_{i(j+1)_1})$  respectively, connecting the own virtual nodes created for different users with one other user's virtual node each. Now, a *BD* group key can be determined by using the received  $V_{ij}$  and the connecting  $W_{ij}$ . This group key is called  $U_i$ 's *circle key*  $K_i$ . Note that  $R_{ij}$  does not play a part in  $U_i$ 's computation, as  $D_{ij_0}$  and  $D_{ij_1}$  are not connected – only the 'outer' nodes form the circle.

Figure 2.11 shows the nodes used for a user  $U_1$ 's circle key connected by a black undirected edge. It also exemplifies how the  $V_{ij} = R_{ji}$  and  $W_{ij}$  values are used to construct the circle, and why the  $R_{ij}$  values are missing.

When writing a message, an encryption key is derived from  $K_i$ . However, the  $W_{ij}$  values necessary to compute it are not known to any user except  $U_i$  at this point. For this reason, every message sent by  $U_i$  contains all  $D_{ij}$  nodes' public keys and all shares of  $K_i$ , i.e. all  $V_{ij}$  and  $W_{ij}$  used to compute  $K_i$ . All  $U_j$  can therefore compute  $K_i$ , and derive the same encryption key.

In fact, a *MAC* key is also derived from  $K_i$  and used over the whole message, which also contains other information such as a session ID and a hash of all messages, including the sent one. But as any user can compute the *MAC* key, origin authentication cannot be provided by it. Instead, it is achieved by a peer-to-peer consistency check over the previously established authenticated channel: After receiving a ciphertext, all users compare the message log hash received as part of the same message. If the consistency

Figure 2.11: Calculation of a *GOTR* circle key.

check succeeded, the message is authenticated.

*GOTR* uses what the authors call a ‘hot-plug’ group key agreement, i.e. users can be added to and removed from the group without making it necessary to restart the key agreement protocol. The four virtual nodes representing the connection to another user, i.e. the flake, can be simply added to or removed from a circle key calculation. This not only simplifies changes to group membership, but also provides resilience against denial of service attacks – a user who fails to provide the necessary information does not disrupt the process and can be simply excluded.

### 2.3.4 Evaluation

As evidenced by the sections on  $(n+1)\text{sec}$  and *GOTR*, establishing a group secret in an authenticated way is not only mathematically complex, but also needs many communication rounds. Assuming a group chat is permanent and rather static, such as it is often the case in modern messaging applications, the cost of one join or leave can be considered



negligible even when high, but both schemes also suggest frequent re-keying in order to achieve forward secrecy. In *GOTR*'s test implementation, establishing a key between 16 chat participants takes a median time of about 6 seconds (with median transmission delay of less than 2 seconds) [LVH13], and in a test of the  $(n+1)\text{sec}$  reference implementation conducted by the developers the same took about 10 seconds [eQuc].

Even though the cryptography underlying both has been proven to fulfill its promises in [ACMP10] and [KY03], it is questionable whether the additional implementation effort and general complexity of a group key agreement protocol adds any value, especially in a case where a robust two-party scheme already exists and a key broadcasting scheme can be applied. Since the first step of a group key agreement is to establish a secret between each pair of users, simply stopping at this point and reusing a two-party scheme to securely transport key material will intuitively take less time. Schemes that can reuse pairwise sessions can speed this up even more.

On the other hand, a simple, naive reuse of a two-party scheme as e.g. presented in section 2.3.1 has more severe drawbacks and should merely act as a counter-example. Using one participant as a virtual server reintroduces properties which were tried to be avoided by employing *OTR* encryption in the first place – there are no guarantees of authenticity or integrity. The latter two properties cannot be achieved without at least initial pairwise authentication.

This leaves using pairwise sessions for transporting locally generated ephemeral key material. *flute*'s scheme as it is can be criticized for being susceptible to disruptions by a malicious ‘captain’, as the draft itself points out [Geob]. Its declared primary use case of establishing a short-term single-use chatroom might also provide a hint that *flute* will not improve its compatibility with more persistent chat models as used by modern messaging applications. For instance, there is currently no mention of a way to establish a new room captain should the first one disconnect – the session is then simply over.

Depending on the group size, encrypting a randomly generated message key for every user is definitely feasible. Since all properties transfer over from the underlying two-party scheme, this can be a simple extension of a well-studied protocol. Some properties which might require more attention in the group setting, such as transcript and speaker consistency, can be easily provided by e.g. also including hashes of the participant list and the chat log in the encrypted data.

If a sender's computation effort and message size are an issue at growing group sizes, they can be reduced by making the recipients save all other participants ephemeral keys, which are transmitted once at the beginning.

Previously made design choices played a big part in the choice of approach for group messaging. Considering a good two-party scheme was already chosen in the previous step, it is only reasonable to reuse it. After *XMPP* was chosen as transport protocol for the reasons laid out in 3.1, it became clear that there exists a protocol extension draft which makes use of the chosen *Signal Protocol*. *OMEMO*'s multi-device support however

## 2. STATE OF THE ART

---

is in essence a group chat between both participants' devices, which is why the same *per-message key* scheme was then also inofficially applied to the group setting.

### 3.1 Approach

#### Encryption

After surveying and evaluating the current technologies in chapter 2, the best suited encryption scheme was chosen. As initially suspected, this turned out to be the *Signal Protocol*. Now, naturally the question arises why there is a need for a new development which utilizes it, considering that, as pointed out before, it already has millions of users through not only through its own *Signal Messenger*, but also *WhatsApp*, *Facebook*, and *Google*. Some missing features, such as native clients on different systems, have already been pointed out. But in addition to that, it seems that the goal of gaining many users in the first case, and using encryption as a marketing strategy to keep them in all the other cases, brings about decisions which are detrimental to the initial goal of privacy. A more detailed discussion follows in the subsections below.

#### Anonymity

This attribute, which can surely be considered a part of privacy, is a great example for the trade-off just mentioned. Tying the identity of the IM user to his or her phone number allows for contact discovery and thus facilitates connecting to friends, who in the end are likely to be the most important factor in the decision whether to stay with the just installed client, or pick one of the many others available. However, privacy-minded users often consider this feature a flaw, preferring anonymity instead. More concretely, the *Signal Messenger* actively informs its users that someone whose number they have saved on their phone just joined – a situation the joining user might not want, but does not suspect from a messenger advertised as respecting privacy [Jam]. Similar points apply to the other messengers which adopted the *Signal Protocol*.

#### Use of An Already Existing Transport Protocol

The original *OTR* paper [BGB04] provides some good reasons why working on top of existing IM protocols is a good idea. These can be summed up as easier integration into existing structures, both technological and social. However, not many others seem to agree. For example, there is the *Wire Messenger*<sup>1</sup>, which uses an own implementation of the *Double Ratchet* called *Proteus*<sup>2</sup>. While *Wire* does allow signing up using just an email address (as opposed to requiring a phone number), the developers decided to create an own protocol. Naturally, there are valid reasons for this, one being that a specifically designed protocol will exactly fulfil the requirements. In this light, the *Cryptocat*<sup>3</sup> messenger's decision to use the open *XMPP* protocol but create an own user network by disallowing outside access seems less understandable. This however is a first pointer towards using *XMPP*, as it shows that end-to-end encrypted messaging can be built on top of it.

#### Native Client

While the chosen *Signal Protocol* does include support for multiple devices in its reference implementation, the applications which utilize it do not make good use of this functionality. Both *WhatsApp* and *Signal* offer the possibility to also use a browser to access their respective networks with the same account as used on the mobile app. However, this then unfortunately also means that both the web app in case of *WhatsApp* and the *Chrome* browser extension in case of *Signal* suffer from the downsides of being just another JavaScript program run in the browser, including e.g. possible XSS or phishing [Day]. Because the identity is tied to the phone number in both cases, native support on additional mobile devices such as tablets also does not exist.

The *Wire Messenger* actually does offer modern native clients on different platforms. But *XMPP*, which is already implemented by many clients on many platforms, an additional requirement can be fulfilled.

#### Federation

Another desirable property, this time not for privacy but for sustainability, is federation. This means that in theory anyone could set up a server, and users are not dependant on the whims or economical situation of a single provider. In reality, not many do this. But looking at the big picture, it is a necessary attribute for a scheme which is intended to last a long time.

The developers of *Signal* vehemently oppose federation [Jak], claiming it is hard to evolve a federated protocol as quickly as it is necessary nowadays. Another one of their main arguments is inconsistency of features, which makes for a bad user experience. While their success may prove them right in their approach in spreading end-to-end encrypted

---

<sup>1</sup><https://wire.com>

<sup>2</sup><https://github.com/wireapp/proteus>

<sup>3</sup><https://crypto.cat/>

communication, there can never be a guarantee that Open Whisper Systems will not for any reason stop supporting the *Signal Messenger*. Therefore it is important to create a sustainable solution employing good cryptography.

To be fair, both the *Signal Messenger* and *Wire Messenger* started releasing the source code of their server software [Opeb, Wir]. But as they do not currently allow federation, it would at best take some time to update the software to support it, and at worst will never happen, leading to a fractured userbase. Thus, *XMPP* appears to be the best candidate for this endeavour, and will be introduced in more detail in the next section.

## 3.2 Used Technologies

### 3.2.1 XMPP

#### Background

The EXTENSIBLE MESSAGING AND PRESENCE PROTOCOL, then called *Jabber*, started off as an open alternative to the closed instant messaging protocols available at the time, such as *MSN* or *ICQ* [xmp]. Later, it was extended and to support more use cases, renamed, and defined in several RFCs [SA11b, SA11c, SA11a].

As the name says, it is designed to be extensible and there exist many *XEPs*, or *XMPP* EXTENSION PROTOCOLS, which can be suggested by anyone. Over time, these added a lot of desirable functionality. For example, there is *XEP-0280: Message Carbons*, adding support for receiving messages on multiple connected devices [Joe]. The prerequisite for this is of course that multiple devices or clients can be logged in at the same time, which was supported by *XMPP* from the start, instead of e.g. disconnecting one device as soon as another login attempt is made.

Because of its openness and extensibility it is often chosen as basis for instant messaging developments. For instance, many companies made use of *XMPP* when building their own chat network, such as Facebook, Whatsapp, and Google [Ste, fip]. As will be explained shortly, these attributes were also very useful when trying to design an encryption solution that works as a part of *XMPP*, and not simply on top of it like *OTR*.

#### Technical Details

At its core, *XMPP* builds on the EXTENSIBLE MARKUP LANGUAGE *XML* [Tim]. This section will describe the basic workings of the protocol, which should be helpful with understanding some of the next sections.

*Jabber IDs* (often shortened to *JIDs*) are the account names used in *XMPP* and look a lot like email addresses: *localpart@domainpart*. The reason for this was previously mentioned: like email, *XMPP* is federated. If a user of one server wants to send a user of a different server a message, a server-to-server (*s2s*) connection is negotiated, and the message then transmitted. Afterwards, the receiving server treats the message the same as one sent among its own users.

Any client connected to an account is called a *resource*, and can be addressed individually like so: *localpart@domainpart/resource*. Otherwise, the *bare JID* without the resource suffix is used, e.g. if the server is supposed to decide which client receives a specific message.

On a lower level, the connection built between two machines and used to send *XML* data is called an *XML stream* in *XMPP*. It is opened by a `<stream>` tag, closed by the corresponding closing tag `</stream>`, and is the equivalent of a “document entity” in *XML* terms.

*XML* “fragments” [Pau] are a way to only deal with the parts of the whole document that are currently of interest, and are called *XML stanza* in *XMPP*. These stanzas are the elements contained in the stream and come in three flavours:

- `<message/>`: The method used to send messages, e.g. to other users.
- `<iq/>`: Stands for “Info/Query” and is a mechanism for structured queries.
- `<presence/>`: Used for announcing various status information.

All of these tags can and do have attributes, for example `from`, `to`, or `type`. Of course, these elements can have more specific sub-elements such as `<error/>` or the `<query/>` tag used inside the `<iq/>` stanza. A number of these is defined in the main specification, but any *XEP* can specify more as needed.

#### 3.2.2 OMEMO

##### Background

OMEMO MULTI-END MESSAGE AND OBJECT ENCRYPTION is an *XEP* which was called to life in order to replace *OTR* as the de-facto standard for forward-secret end-to-end encryption on *XMPP*. The authors’ reasoning sounds very similar to the drawbacks mentioned in the evaluation of *OTR* in section 2.1.2, and likely for similar reasons as pointed out in section 2.1.3, the *Signal Protocol* was chosen to power *OMEMO* in the background [And]. In fact, this *XEP* was developed in the context of *Conversations*<sup>4</sup>, an *XMPP* client for Android, in order to specifically deal with the shortcomings of *OTR* in the mobile environment. This program will be considered the reference implementation in the following sections.

When the work on this thesis began, *OMEMO* was merely a *Proto-XEP*, i.e. not officially accepted yet, and therefore also not widely implemented, which is why it seemed like a good candidate to achieve the goals laid out in the beginning. This is especially since again a promising new development was aimed at mobile devices. Luckily, this time it could be extended to desktop clients as well.

---

<sup>4</sup><https://conversations.im/>

## Technical Details

One very good design choice was to port the *Signal Protocol*, which depends on server-side logic, in a way that avoids this dependency. This is mostly for organizational reasons, as writing not only new client code but also new server extensions meant more work on the one hand, and on the other hand experience tells that server administrators are slow to install new software. It was done using *XEP-0163: Personal Eventing Protocol* (or *PEP* for short), which is already widely deployed on servers and available on clients. It is a subset of a more general publish-subscribe functionality described in *XEP-0060* [Peta] and allows *XMPP* users to create nodes, publish information on them, and have the subscribed users be automatically notified of changes [Petc]. In *PEP*, the subscribed users are usually the “friends” in the contact list, i.e. the users who have a presence subscription already. An example named in the specification is receiving updates about the song contacts are currently listening to [Petb], but as will be evident soon, it can also be employed for a lot more useful undertakings.

In *OMEMO*, *PEP* is used to distribute the information which would usually be held by the central server needed for asynchronicity features, i.e. all the *pre-keys*, the *signed pre-key*, and the public *identity key*. Like in the *Signal Protocol*, this collection of data is called a *bundle* and is published on such a *PEP* node. Unlike in *Signal*, the identity key is not shared by all devices of an account, but generated per device, which is why each client has to upload its own bundle. This makes it necessary to also have a per-account node containing a list of the *OMEMO*-supporting devices’ IDs, which is simply called *devicelist*. Setting or updating information on the nodes happens by simply sending an `<iq/>` stanza.

In order to start a conversation with another *OMEMO* user, his or her *devicelist* node is consulted, and then each device’s bundle is retrieved via the corresponding node so that a session can be built as usual. The own *devicelist* is also consulted, because each device has to build a session with every other device participating in the conversation, including the own.

The message body is not encrypted for every device, however. *OMEMO* uses a common scheme which was described in section 2.3.2: The payload is encrypted using a symmetric-key algorithm, in this case *AES-128* in *GCM*, using a randomly generated key and initialization vector. Then this 16-byte key, is then encrypted using the previously established long-term ratcheting session.

Even though *OMEMO* is designed to handle multiple devices, in this case meaning that a message is correctly encrypted for each of them as previously described, pure *XMPP* only delivers messages to one of the devices connected to the account, and which one is chosen is decided by implementation-dependent rules. This also means that none of the other own devices receive any copies of sent messages.

In order to improve this dated behaviour which treats each resource separately by default, the use of *XEP-0280: Message Carbons* [Joe] is recommended in the *OMEMO* specification, which is again supported by many *XMPP* servers and clients. With this extension, copies of sent messages are sent to all other clients of the same account, and incoming

messages are replicated for all of them as well (provided all clients support message carbons and have announced this to the server). The result is a consistent view across multiple devices which are online at the same time and prevents annoying situations which previously arose, such as the server forwarding messages only to the ‘wrong’ device, i.e. not the one expected by the user.

Of course it is not guaranteed that all devices a user owns are online at the same time, so for a limited kind of history synchronization another *XEP* needs to be employed. This extension is called *XEP-0313: Message Archive Management*, or *MAM* in short.

A server which supports *MAM* simply saves every message a user sends and receives, unless configured otherwise. When a client reconnects, it can query the archive for the activity that happened when it was offline, and reconstruct the message log. However, the `<message/>` stanzas sent with *OMEMO* do not have a `<body/>`, as the encrypted message is sent inside an `<encrypted/>` tag instead. Therefore it is necessary to give the ‘hint’ to still archive it, which is simply done by including a `<store/>` tag.

It should be noted that this history synchronization only works for devices which are already part of the conversation, but were simply offline at the time. More specifically, it does not mean that devices added at a later time can retrieve the history, as this is mutually exclusive with forward secrecy.

#### Evaluation

There exists an audit of this protocol which found some possible attacks that apply for uncautious users, though it is not clear how big of a threat they pose in reality [Seb]. In addition, there are some suggestions for good implementation practices.

A short summary follows.

One big problem pointed out is the possibility of a malicious device added by an attacker, as it can decrypt the symmetric key and thus also re-encrypt and re-authenticate any payload. Even though *GCM* automatically creates an authentication tag, it is simply appended to the ciphertext. The attack can even happen undetected if the attacker strips the message of references to the malicious device’s own ID.

The mitigation suggested in this document is including the authentication tag in the data that is encrypted by the *Double Ratchet* session, i.e. append it to the payload encryption key. This results in the actual ciphertext being authenticated, so that no malicious device can compromise its integrity. The downside of this suggestion is increased computational load, but this was deemed acceptable by the developers, seeing how this approach was adopted [Dan].

Another suggestion made in the audit is encrypting the list of recipients inside the *Double Ratchet* session as well, so that the client of a recipient could warn of unidentified devices (i.e. participant consistency). The actual problem here though is that adding to the device list of a user is not protected enough. A solution could be to require new devices to acquire a signature from an existing, trusted device, as it is less likely that a



user adds an unknown device to his own than questioning devices of his conversation partner. These signatures should then also have a limited lifetime and be revokable. However, the per-device trust decisions for both own and foreign devices seem to be enough of a mitigation, as the developers did not change anything in regard to this suggestion.

A different class of problems concerns devices that are either inactive, or never used to send a message, as the *Double Ratchet* root key is never moved forward, in effect disabling the *DH* ratchet.

It can be easily solved by a slight tweak, as *OMEMO* describes a `KeyTransportElement` `<message/>` which is specifically made for transporting key material and has no content. This type of message can be used as a ‘heartbeat’: Sent regularly, it will help evolving the root key, and can also be an indication of devices that are not used anymore.

This suggestion was also incorporated, but instead of using it directly, it was written down for a future, yet unpublished version of the *XEP* [str].

### 3.2.3 Pidgin and libpurple

*libpurple* is an open source instant messaging library which supports many protocols and is easily extendable by plugins. Its graphical frontend *Pidgin* is one of the more commonly used *XMPP* clients – for instance, it is often preinstalled on Linux distributions. These were probably the reasons for *OTR* to use it as a host for its reference implementation, and it seems they are equally good reasons in this case. Regrettably, the development has really slowed down since then, and many more modern *XEPs* are not implemented, which added some implementation effort.

In addition to providing a certain set of officially supported protocol plugins (*prpls*) such as the one for *XMPP*, *libpurple* exposes an interface for additional protocol implementations, which might be one of the causes for its success. At the moment, 40 additional messaging protocols can be added <sup>5</sup>.. For non-protocol plugins, *libpurple* also offers additional interface, making it possible to not only add internal, but also user-facing functionality which will then work in the frontends without writing specific code. As this is used in the implementation part, a short introduction follows.

There exists a simple per-account key-value store, making it possible to save simple data without DB access. For instance, a binary value such as if some option is turned on or off can be set by `purple_account_set_bool()`, and retrieved at a later point using `purple_account_get_bool()`. Similar functions for number and character types can be found as well. In order to enable plugins to react to events, there is a number of pre-defined signals, and more can be added. For instance, there is `account-signed-on` or `conversation-created`, and the *XMPP* protocol plugin adds several more specific signals such as `jabber-receiving-xmlnode`. A handler function for a signal can be registered using `purple_signal_connect()`. The previously mentioned possibility to add to

---

<sup>5</sup><https://developer.pidgin.im/wiki/ThirdPartyPlugins>

the user interface is based on ‘requests’ which can let the user input some value, ‘actions’ which are added as a menu item in the *Tools* menu, ‘commands’ which can be typed into the textbox, and finally an interface to simply add a plugin preferences dialog. As will be evident in section 4, even by only using commands a lot of useful functionality can be implemented.

### 3.2.4 Programming Language

After the host platform was chosen to be *libpurple*, the choice of the programming language was restricted to the same one it uses, namely *C*. Fortunately, an officially supported implementation of the *Signal Protocol* exists as *libsignal-protocol-c* [Opea].

### 3.2.5 Additional Libraries

**SQLite** In contrast to many other database systems, *SQL* or not, *SQLite* does not follow a client/server architecture, but rather uses a local file in a cross-platform data format. Still, it is a full-featured, transactional *SQL* implementation<sup>6</sup>.

As the official website points out, this is likely the reason it is one of the most widely deployed libraries, and a common choice for use as an application file format<sup>7</sup>. Naturally, it was therefore also a good choice for this application.

**cmocka** A simple unit testing framework<sup>8</sup>.

**libgcrypt** “*Libgcrypt* is a general purpose cryptographic library [...] [which] provides functions for all cryptographic [sic] building blocks”<sup>9</sup>. It is used to implement the cryptographic functions needed by *libsignal-protocol-c* for *axc*, and by *libomemo*.

**Mini-XML** A small library for handling *XML* data<sup>10</sup>. Used in *libomemo* to deal with *XMPP* messages, which as described before are based on *XML*.

**GLib** *GLib* provides an implementation of common data structures and many utility functions such as *base64* and is available on many platforms. It is written by (and initially extracted from) the GNOME project<sup>11</sup>.

---

<sup>6</sup><http://sqlite.com/features.html>

<sup>7</sup><http://sqlite.com/about.html>

<sup>8</sup><https://cmocka.org/>

<sup>9</sup><https://gnupg.org/software/libgcrypt/index.html>

<sup>10</sup><https://michaelsweet.github.io/mxml/>

<sup>11</sup><https://wiki.gnome.org/Projects/GLib>

# Implementation

## 4.1 General Notes

For several reasons, the implementation was divided in several more or less distinct parts. As hinted at before, *libpurple*'s *XMPP* protocol implementation is missing many newer *XEP*s. Specifically this concerns the extensions previously described as complementing *OMEMO*: *XEP-0280: Message Carbons* and *XEP-0313: Message Archive Management*. While the latter can be regarded as simply providing the comfort function of history synchronization, the former is definitely necessary for multi-device support, and was therefore implemented as a separate plugin. This *carbons* plugin is described in section 4.2.

Before *libsignal-protocol-c* can be used in a program, several application-specific interfaces have to be implemented. In order to enable reuse for a possibly differing multi-party chat implementation, this code was put into its own library. It also contains all utility functions which turned out to be necessary, which resulted in the complete abstraction of the *Signal Protocol* library. The result is called *axc* (AXOLOTL CLIENT) and discussed in section 4.3. It is named that way because when implementation started, the *Double Ratchet* was still called *Axolotl*, and its library therefore *libaxolotl-c*.

While not likely to be of use in a different part of this work, the code for handling the protocol specified in *OMEMO* was also put in its own library, making it possible for others to reuse it, should interest arise. This library was simply named *libomemo* and is described in more detail in section 4.4.

Lastly, the previous two pieces of software were combined into one plugin for *libpurple*, called *lurch*. How exactly that was done is outlined in section 4.5.

### 4.2 carbons

This plugin can be found at <https://github.com/gkdr/carbons>.

#### 4.2.1 XEP-0280: Message Carbons

The goals of this extension have been discussed already, however the technical details were not. This will be done in this short section, based on the *XEP* found at [Joe].

##### Installation

Initially, aside from announcing support for this feature through adding it to its service discovery replies, a client needs to send a message to the sever to enable this functionality. After such a message was sent, an example of which can be seen in listing 4.1, the server can acknowledge the enabling, or send an error message. A similar request can be sent to disable this feature again, switching out the `<enable/>` for a `<disable/>`.

Listing 4.1: Enabling Message Carbons

```
<iq xmlns="jabber:client"
  from="alice@example.com/desktop"
  type="set">
  <enable xmlns="urn:xmpp:carbons:2"/>
</iq>
```

##### Receiving Carbon Copies

Now, a client can expect two different sorts of messages: copies of incoming messages sent by other users, and copies of outgoing messages sent by another client of the same account.

In the case of the former, the incoming message, i.e. the `<message/>` stanza, is wrapped by a `<forwarded/>` element, which is in turn wrapped by a `<received/>` element of the `carbons` namespace. This ‘envelope’ is put into a `<message/>` stanza which is directly addressed to the client by its resource. An example can be found in listing 4.2. In it Alices desktop client receives a message Bob originally sent to her mobile phone. Copies of sent messages are wrapped analogously, just the `<received/>` element is replaced by a `<sent/>` element, as can be seen in listing 4.3. There, Alice sends a reply to Bob from her mobile device, and the server sends a copy to the desktop client. To have a sent message copied to other devices, no additional hints are necessary – for clients which previously enabled message carbons, this is done automatically based on a set of rules. A client can exclude sent messages from carbon-copying by adding hints though if necessary.

Listing 4.2: A carbon-copied received message

```

<message xmlns="jabber:client"
  from="alice@example.com"
  to="alice@example.com/desktop"
  type="chat">
  <received xmlns="urn:xmpp:carbons:2">
    <forwarded xmlns="urn:xmpp:forward:0">
      <message xmlns="jabber:client"
        from="bob@example.com/mobile"
        to="alice@example.com/mobile"
        type="chat">
        <body> Are you home? No one is answering the door. </body>
      </message>
    </forwarded>
  </received>
</message>

```

Listing 4.3: A carbon-copied sent message

```

<message xmlns="jabber:client"
  from="alice@example.com"
  to="alice@example.com/desktop"
  type="chat">
  <sent xmlns="urn:xmpp:carbons:2">
    <forwarded xmlns="urn:xmpp:forward:0">
      <message xmlns="jabber:client"
        to="bob@example.com/mobile"
        from="alice@example.com/mobile"
        type="chat">
        <body>
          Yes, sorry, I'm in the garden and did not hear the bell. I'm coming.
        </body>
      </message>
    </forwarded>
  </sent>
</message>

```

## Security Considerations

It is important for a client to check that the outer `<message/>`'s `from` attribute's value is equal to the user's bare JID (e.g. `alice@example.com`), as seen in the examples in listings 4.2 and 4.3. Otherwise, any user can fake such a forwarded message. Even though this fact is noted in the *XEP*'s security considerations, it was recently discovered that a number of popular *XMPP* clients did not follow the specification and thus were vulnerable to this type of attack [Geoa].

### 4.2.2 Implementation Details

#### Initialization

The plugin exposes a simple interface to the user by registering one command and its command handler function `carbons_cmd_func()` at startup. This command can then be called by typing `/carbons` in any message window. It takes either `on` or `off` as argument, turning message carbons on or off for the issuing account, which is done by calling the `carbons_switch_do()` function with the corresponding argument from inside the command handler. A stanza like in listing 4.1 is then created and sent to the server, and `carbons_switch_cb()` registered as callback function for the result returned by it. Depending on whether it was a request to enable or disable the functionality and whether the server's reply is positive or negative, this callback function saves the `carbons_enabled` account setting as `true` or `false`. At startup, this setting is checked inside the function `carbons_autoenable()` to decide whether the request to enable message carbons should be sent automatically.

#### Reacting to Events

The main work happens in the `carbons_xml_received_cb()` function for handling of the `jabber-receiving-xmlnode` event, i.e. an incoming *XMPP* stanza.

As no filtering was done so far and any incoming stanza is passed through this handler function, it first needs to check if it needs to do any work in the first place. This is simply done by looking for a either a `<received/>` or `<sent/>` node with the `urn:xmpp:carbons:2` namespace. If none is found, the function does nothing, otherwise the processing continues.

In case a message was determined to be a carbon copy, it needs to be checked for admissability by making sure it was sent by the user's bare JID. To this end, the function `carbons_is_valid()` is called, and in case it returns `false`, processing is aborted.

#### Handling Copies of Incoming Messages

A valid `<received/>` message is now simply stripped of all outer elements, and the contained `<message/>` is passed on by the handler. Essentially, the plugin makes *libpurple* think the received carbon-copy of a received message is a regular received message. The message shown in listing 4.2 would therefore look like in listing 4.4 after passing through this function. Since the bare *JID* still matches the user's account, it is not a problem that the message is addressed to the `mobile` resource instead of the own `desktop` resource name via the `to` attribute.

Listing 4.4: The contents of listing 4.2 with the outer message stripped

```
<message xmlns="jabber:client"
  from="bob@example.com/mobile"
  to="alice@example.com/mobile"
  type="chat">
  <body> Are you home? No one is answering the door. </body>
</message>
```

## Handling Copies of Outgoing Messages

`<sent />` messages are a little more tricky, as *libpurple* does not inherently know what to do with an incoming message that was sent by the own account. `<message />`s without a `<body />` are stripped and passed on like before, which also includes *OMEMO* messages as will be seen later. Ones that do contain a `<body />` however, i.e. ‘normal’ messages, need to be handled by the plugin itself. As *libpurple* does not know what to do with such a message, the corresponding conversation window needs to be found or created, and the contained text written to it manually.

Since this plugin often times only strips the message carbons ‘envelope’ and passes the contents on as if they were originally received by the client, it is important to not register the `carbons_xml_received_cb()` handler at standard priority, as the plugin needs to be one of the first to process the received stanza. Otherwise, it would not work together with e.g. *OMEMO*, which needs this outer `<message />` stripped first in order to understand the contents. Therefore, considering *libpurple* iterates through handlers from lowest to highest in priority, it is registered at `PURPLE_PRIORITY_LOWEST + 100`, both to definitely be in front of the standard priority of 0, but also leave enough room for other plugins which might need to handle incoming stanzas even earlier.

## 4.3 axc

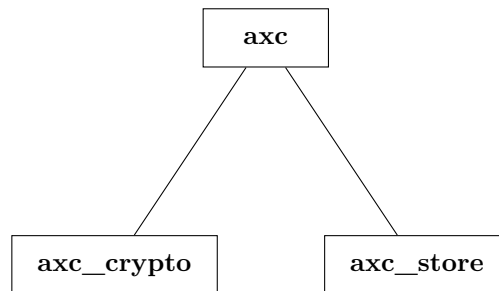
This library can be found at <https://github.com/gkdr/axc>.

### 4.3.1 The libsignal-protocol-c Interface

In order to discuss *axc*, its backend *libsignal-protocol-c*, the reference implementation of the *Signal Protocol*, needs to be briefly explained first.

As the README states [Opea], a global context struct `signal_context` has to be initialized before the library can be used, as this context has to be passed to all functions. Many also additionally require an initialized `signal_protocol_store_context`. For these to be initialized, sub-context structs containing function pointers need to be passed to them. The pointed-to functions are application-specific and need to be implemented first, which is a big part of the code that is contained in *axc*. These functions are used in the higher-level functions exposed by *libsignal-protocol-c*, which might perform additional

Figure 4.1: axc modules



checks or decode the data returned by the database access functions from raw bytes to higher-level data types.

*libsignal-protocol-c* also does not offer any higher-level utility functions. For instance, when a client is initialized, 5 different functions have to be called to create the necessary data. In order to build a session, a `session_builder` has to be created first, and only then the bundle can be passed to it. The same is true for encrypting or decrypting a message, in which case it is a `session_cipher` struct instead. Grouping functions that usually are called together in order to simplify the process of working with the *Signal Protocol* is the other part of *axc*.

### 4.3.2 Implementation Details

#### Design

As can be seen in 4.1, *axc* is divided into three modules.

*axc\_crypto* simply implements all the cryptographic functions which need to be passed to the `signal_context` through the `signal_crypto_provider` struct. These are functions for initializing, updating, finalizing and cleanup of a *SHA256 HMAC* and a *SHA512* hash, as well as encryption and decryption functions for *AES*, and a function providing random bytes.

*axc\_store* mostly does the same for the ‘store’ functions which need to be passed to the `signal_protocol_store_context`, again in their own structs. There are four ‘stores’ resulting from the different groups of data: `signal_protocol_session_store`, `signal_protocol_pre_key_store`, `signal_protocol_signed_pre_key_store`, and `signal_protocol_identity_key_store`. Each of those mostly just contains the rather typical functions to set, load, and delete an item, as well as checking for existence.

The only function which is supposed to contain some sort of logic is `is_trusted_identity()`, which by the interface documentation should check if the provided identity key matches the saved one for the contact, in which case it is trusted. Unfortunately, this is not compatible with *OMEMO* as each device has its own identity key, and therefore the trust



assessment had to be deactivated here by the function `axc_db_identity_always_trusted()` and deferred to the *OMEMO* implementation.

Different from the previous module, this one does need some additional utility functions. A rather obvious first example is `axc_db_create()` which needs to be called at client initialization to create the necessary tables inside the *SQLite* database. `axc_db_destroy()` does the opposite and drops all tables.

For some types of data, *libsignal-protocol-c* requires only reading, but no writing functions, meaning they are additionally contained in this module as well. This concerns the functions for saving the generated own identity key pair and registration ID, `axc_db_identity_set_key_pair()` and `axc_db_identity_set_local_registration_id()`.

When a new pre-key pair is generated, its ID is not automatically set, but has to be supplied as an argument. For *axc*, the design decision was made to generate sequential IDs (as opposed to random ones), therefore one function had to be written which retrieves the highest currently used ID that is not the ‘last resort’ ID’s key, i.e. not `MAX_INT`. Also, client initialization was sped up remarkably by using `axc_db_pre_key_store_list()` which saves the whole generated list of pre-key pairs at once inside a single transmission. The ‘reverse’ function `axc_db_pre_key_get_list()` is useful when assembling a bundle for publishing, and therefore only retrieves the public pre-key.

Finally, there are getters and setters for key-value attributes saved in the `options` table, which are e.g. used to save the database’s initialization state.

All of what remains is realized in the main *axc* module.

The last functions needed to fully initialize the `signal_context` are locking and logging functions, which are implemented in `recursive_mutex_lock()`, `recursive_mutex_unlock()`, and `axc_default_log()`. However, these are optional, and *axc* offers the possibility to exempt the `pthread`-based locking functions by setting `NO_THREADS` at compile-time, as they might not be needed in a non-threaded environment and complicate cross-compatibility. In fact, this is done for working with the single-threaded *libpurple* to simplify compilation for Windows. If they do need to be used, the function `axc_mutexes_create_and_init()` properly initializes them.

Aside from that, all of this library’s public interface is implemented in this module. This includes the two structs `axc_context` which is used to hold the internal state, and `axc_bundle` which holds all the data necessary for publishing the bundle. Both come with functions for constructing and destructing, and since both of these are opaque structs, there also exist some setters and getters for the fields which can be exposed, such as `axc_context_get_db_fn()` for retrieving the previously set path to the DB. Some data structures exposed by *libsignal-protocol-c* are typedef’d for brevity and a consistent interface – `signal_buffer` becomes `axc_buf`, and `signal_protocol_address` becomes `axc_address`. The decision to provide a single consistent interface from one file also required to write simple wrapper functions.

Getting to the main functionality of this library, functions can be found for initializing the context (`axc_init()`), ‘installing’ it by populating the database with the necessary data (`axc_install()`), and functions for simplifying encryption and decryption to a single function call (`axc_message_encrypt_and_serialize()` and `axc_message_decrypt_from_serialized()`).

Utility functions for building sessions from both a requested bundle and an incoming pre-key message are also provided, as well as for deleting them again, and checking for existence.

How exactly these three modules then interact is described in the next sections based on common usage scenarios.

### Client Initialization

At every startup, a client needs to create an `axc_context` by calling `axc_context_create()`. Optionally, the DB path, log function, and log level can then be set by using the `axc_context_set_*`() group of functions. Usually, at least the path to the database file is set.

With this context, the library can then be initialized through `axc_init()`. Essentially, this function just initializes a `signal_context` and a `signal_protocol_store_context` by setting pointers to all required functions, which as previously described are implemented in `axc_crypto` and `axc_store`. References to these two structs are saved in the passed `axc_context`. If needed, the semaphores are also initialized, and the implemented locking and unlocking functions passed to the `signal_context`. As can be seen in listing 4.5, that is all the data contained in this struct. An example of this process can be seen in figure 4.2.

Listing 4.5: The `axc_context` struct

```
struct axc_context {
    signal_context * axolotl_global_context_p;
    signal_protocol_store_context * axolotl_store_context_p;
    axc_mutexes * mutexes_p;
    char * db_filename;
    void (*log_func)(int level, const char * message, size_t len, void * user_data);
    int log_level;
};
```

If the client is being run for the first time (or is to be re-initialized), `axc_install()` needs to be called afterwards. This function first calls `axc_db_create()` to create the database file and the necessary tables inside it, using the specified database path if set in the passed `axc_context`. Then, the necessary data – i.e. the long-term identity keypair and short-term pre-key pairs – is generated using the `signal_protocol_key_helper_generate_*`() family of functions, and saved to the newly created database. With this, the library is ready to use. An example run of `axc_install()` can be seen in figure 4.3.

However, in the usual, asynchronous mode of operation, the client is not yet fully initialized, as the public parts of the just generated keypairs still need to be published. Therefore, a client will generally want to call `axc_bundle_collect()` to extract this information back from the database in an already serialized, distribution-ready format. The steps done in this function are shown in figure 4.4. For simplicity, all of the data is gathered inside a single `axc_bundle` struct as seen in figure 4.6, and can then be extracted

Figure 4.2: Sequence diagram of client initialization

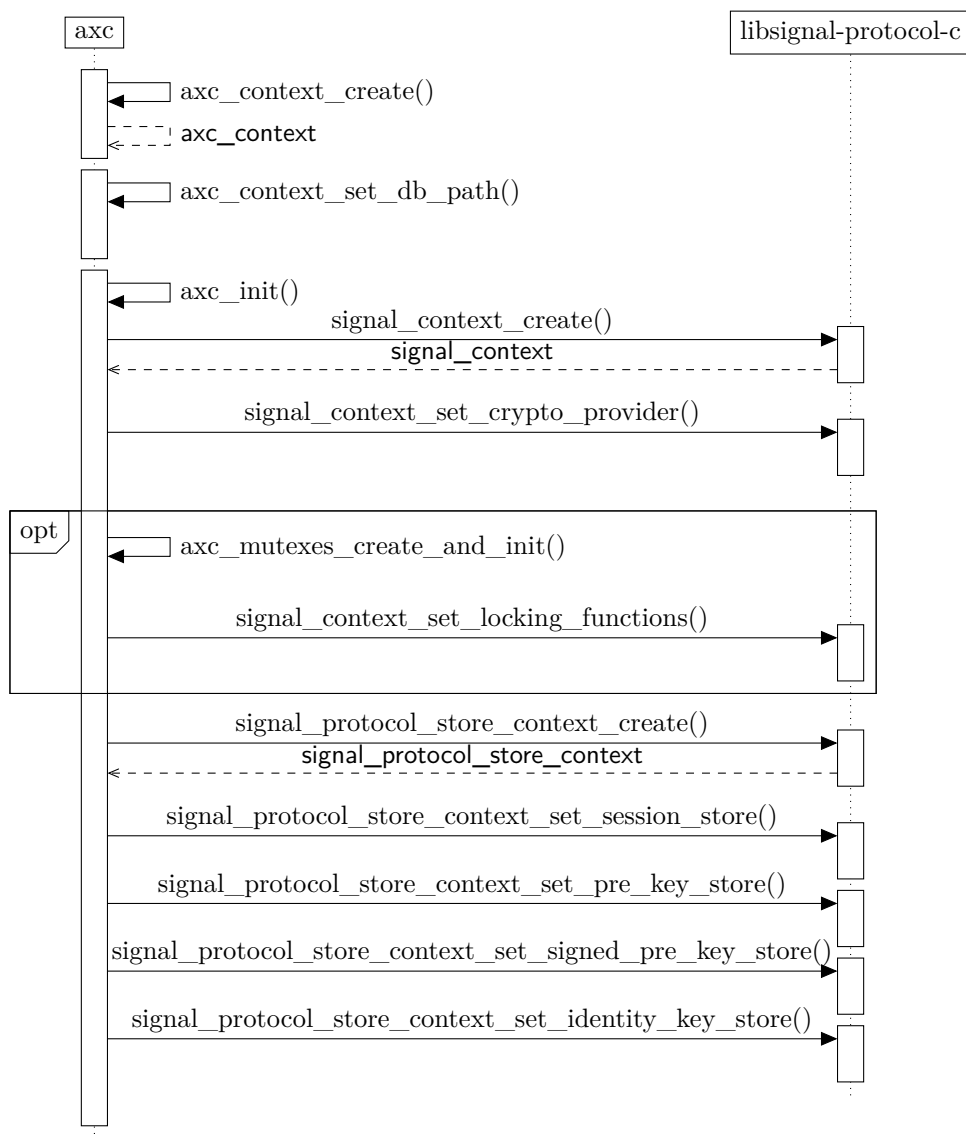


Figure 4.3: Sequence diagram of example client installation

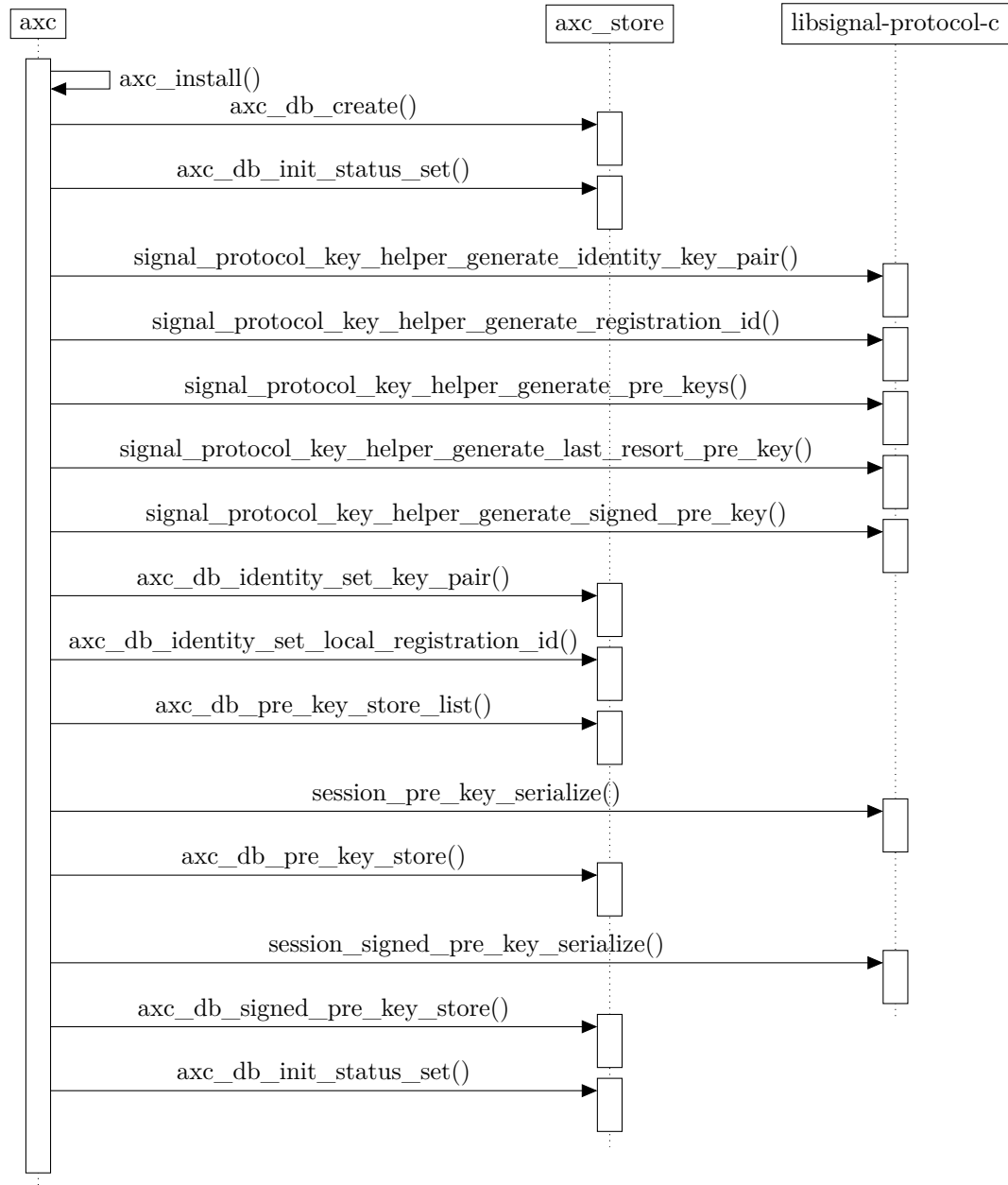
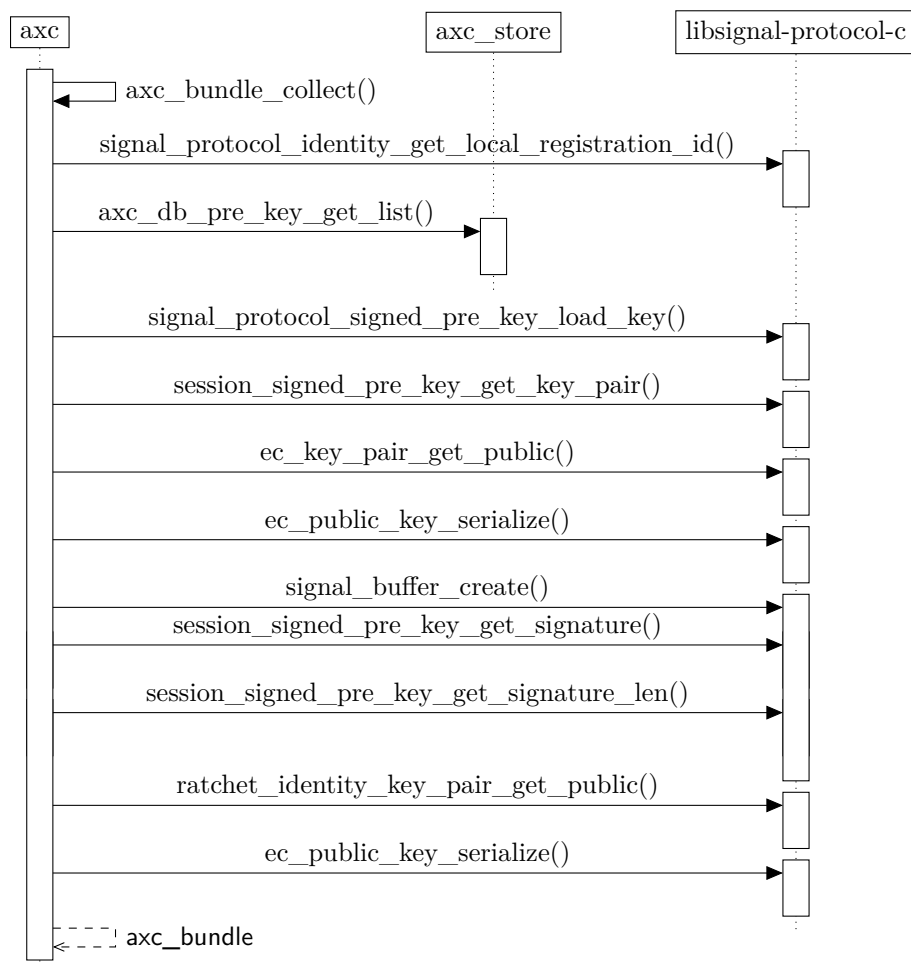


Figure 4.4: Sequence diagram of bundle collection



via getter functions (`axc_bundle_get_*`()).

Listing 4.6: The `axc_bundle` struct

```

struct axc_bundle {
    uint32_t registration_id;
    axc_buf_list_item * pre_keys_head_p;
    uint32_t signed_pre_key_id;
    axc_buf * signed_pre_key_public_serialized_p;
    axc_buf * signed_pre_key_signature_p;
    axc_buf * identity_key_public_serialized_p;
};

```

### (First-Time) Sending

In order to send a message to another device with which a session does not exist yet, a client needs to retrieve that device's bundle, the details of which are application-specific and out of the scope of this library. After this was done, the public keys contained in the received bundle can be passed to `axc_session_from_bundle()` in order to locally establish a session with the bundle's owner device. Inside this function, the given public keys are deserialized using `curve_decode_point()` to create the *libsignal-protocol-c* bundle type `session_pre_key_bundle`. Then, this bundle can be handed to `session_builder_process_pre_key_bundle()`, which does the actual job of constructing a session. Now, all it takes to encrypt a message for that device is to pass `axc_message_encrypt_and_serialize()` its `axc_address`. Internally, a `session_cipher` struct is created, which is then used to encrypt the given message. The resulting `ciphertext_message` is serialized and a copy of this data returned for sending. In case it was the first message after a local session establishment from a bundle (and each next time until a reply is received), the serialized message will be a so-called *pre-key message*. The whole process can be seen in figure 4.5.

### (First-Time) Receiving

As previously explained, pre-key messages contain the necessary information to create a corresponding session on the receiving end and subsequently decrypt the contained (and any following) message. Since *libsignal-protocol-c* does not offer detection of message type, the receiving client can confirm that indeed no session exists by consulting `axc_session_exists_initiated()`. In this case, the pre-key-message-specific `axc_pre_key_message_process()` needs to be called. This function deserializes the pre-key message with the corresponding function so that it can be passed to `session_builder_process_pre_key_signal_message()`. Afterwards, the contained ciphertext can be decrypted via `session_cipher_decrypt_pre_key_signal_message()` and the plaintext is returned. Additionally, this function automatically generates a new pre-key pair to replace the used one, which in turn is automatically deleted by *libsignal-protocol-c*. If a session already exists, `axc_message_decrypt_from_serialized()` should be called instead. At the core, it does the same – deserializing the message, decrypting the ciphertext, and returning the plaintext. Obviously, the session establishment does not need to be done, and the functions not specific to pre-key messages are called. Figure 4.6 shows an example of first-time receiving.

Figure 4.5: Sequence diagram of first-time sending

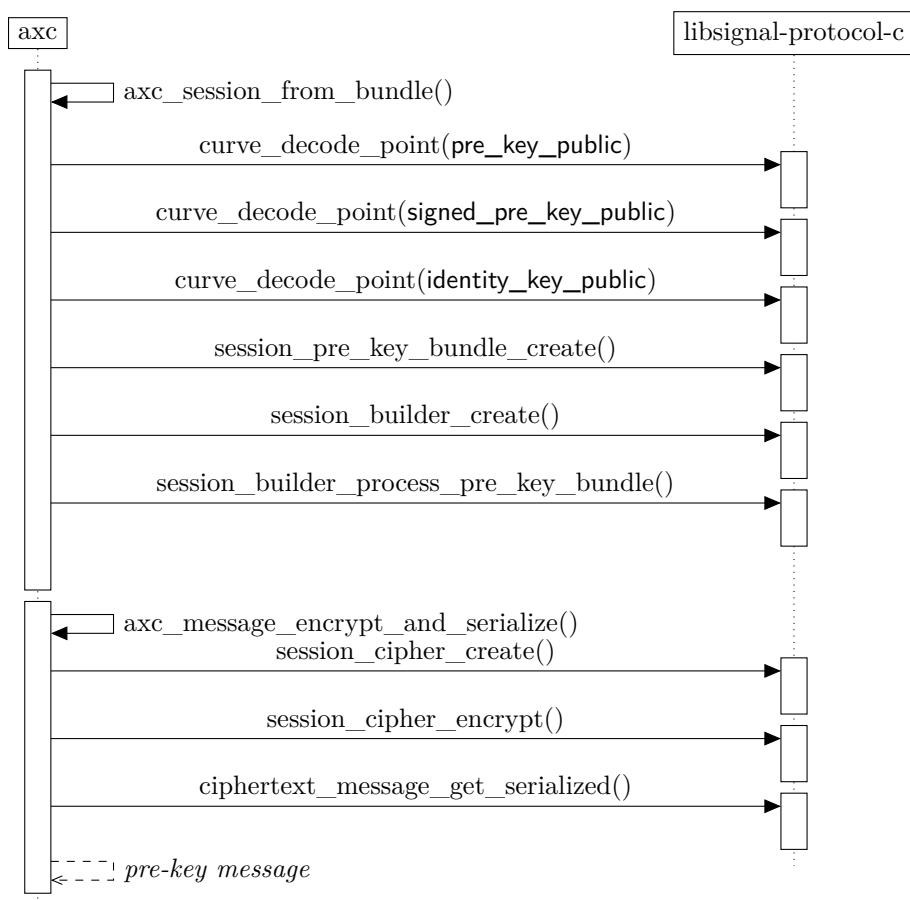
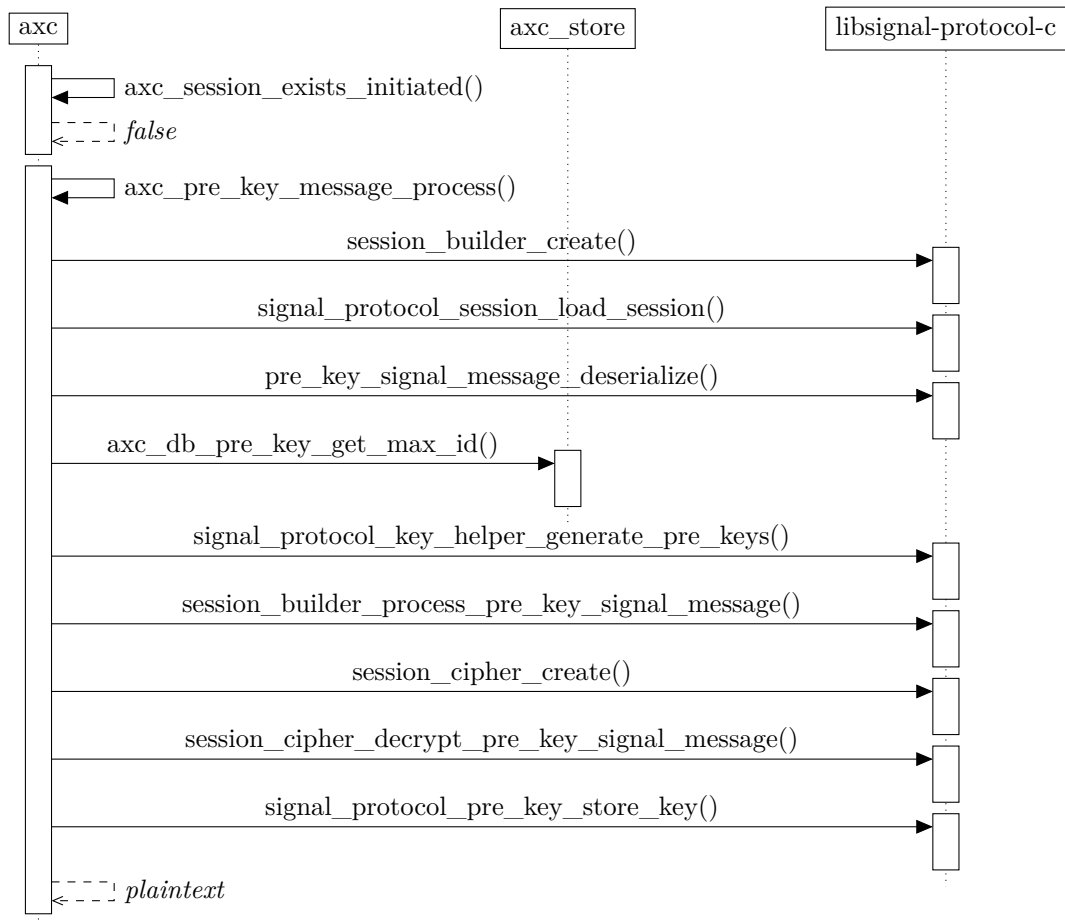


Figure 4.6: Sequence diagram of first-time receiving





## 4.4 libomemo

This library can be found at <https://github.com/gkdr/libomemo>.

### 4.4.1 XEP-0384: OMEMO Encryption

While section 3.2.2 discussed higher-level details such as the mapping between *OMEMO* and *Signal Protocol* elements, this section will deal with the low-level protocol description so that the *libomemo* implementation can be understood more easily. Naturally, this is based on the specification found in the *XEP* [And].

#### Installation

In order to announce *OMEMO* support, a client needs to append the the *Double Ratchet* device ID it generated to the *devicelist PEP* node by sending an `<iq/>` stanza like in listing 4.7. The example assumes Alice does not own any other *OMEMO*-capable client, and that the generated ID is 123456. In order to complete the installation, a second step is necessary. The client now needs to collect all data belonging in a bundle, and publish it to its own *bundle PEP* node via a stanza like shown in listing 4.8.

Listing 4.7: A device adding itself to the previously empty *devicelist PEP* node

```
<iq from="alice@example.com" type="set">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="urn:xmpp:omemo:0:devicelist">
      <item>
        <list xmlns="urn:xmpp:omemo:0">
          <device id="123456" />
        </list>
      </item>
    </publish>
  </pubsub>
</iq>
```

Listing 4.8: A device publishing public key information on its bundle *PEP* node

```
<iq from="alice@example.com" type="set">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="urn:xmpp:omemo:0:bundles:123456">
      <item>
        <bundle xmlns="urn:xmpp:omemo:0">
          <signedPreKeyPublic signedPreKeyId="1">
            <!-- base64-encoded data -->
          </signedPreKeyPublic>
          <signedPreKeySignature>
            <!-- base64-encoded data -->
          </signedPreKeySignature>
          <identityKey>
            <!-- base64-encoded data -->
          </identityKey>
          <prekeys>
            <preKeyPublic preKeyId="1">
              <!-- base64-encoded data -->
            </preKeyPublic>
            <preKeyPublic preKeyId="2">
              <!-- base64-encoded data -->
            </preKeyPublic>
            <!-- ... -->
          </prekeys>
        </bundle>
      </item>
    </publish>
  </pubsub>
</iq>
```

## Receiving A Contact's Information

Suppose Bob is both already an *OMEMO* user and Alice's contact, and his client is subscribed to the `urn:xmpp:omemo:0:devicelist` namespace as stated in the *XEP*. When Alice updates her `devicelist` node like in listing 4.7, Bob is automatically notified by his server in a message similar to the one seen in listing 4.9. It contains the complete `devicelist`, which Bob's client is supposed to cache locally in order to be able to determine changes to Alice's devices later.

Listing 4.9: Bob receiving Alice’s updated devicelist

```
<message from="alice@example.com" to="bob@example.com">
  <event xmlns="http://jabber.org/protocol/pubsub#event">
    <items node="urn:xmpp:omemo:0:devicelist">
      <item>
        <list xmlns="urn:xmpp:omemo:0">
          <device id="123456" />
        </list>
      </item>
    </items>
  </event>
</message>
```

Before Bob can send an *OMEMO*-encrypted message to Alice, his client needs to manually request all of her device’s bundles. In order to avoid collisions of used pre-keys, this ideally should happen only shortly before a session is to be established. Luckily for Bob, Alice only owns one *OMEMO*-enabled device, so a single request like in listing 4.10 suffices. From the server’s reply, which is excluded for brevity as it contains the whole bundle, Bob can then build a *Double Ratchet* session.

Listing 4.10: Bob requesting the bundle of Alice’s 123456 device

```
<iq from="bob@example.com" to="alice@example.com" type="get">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <items node="urn:xmpp:omemo:0:bundles:123456" />
  </pubsub>
</iq>
```

## Message Format

With the session established, Bob can now send encrypted messages to Alice. These look as follows: The parent element of all *OMEMO*-related information is an `<encrypted/>` element of the `urn:xmpp:omemo:0` namespace, which is a direct child of the message stanza’s top-level `<message/>` element. The text Bob enters is encrypted using *AES-128* in *GCM* with a randomly generated key and initialization vector. The resulting ciphertext is *base64*-encoded and put inside a `<payload/>` element. Afterwards, the `<header/>` can be constructed, whose `sid` (‘sender ID’) attribute is set to Bob’s device’s ID. The used encryption key is concatenated with the authentication tag which was automatically produced by *GCM* encryption, and this data is then again encrypted using the previously established *Double Ratchet* session. Again, the result is *base64*-encoded for sending, and put inside a `<key/>` element, which is added as a child of the `<header/>` element. This procedure is repeated using every recipient device’s *Double Ratchet* session, and receiving clients can later discern which of the `<key/>` elements is meant for them by the `rid` (‘recipient id’) attribute. Finally, the *IV* is also added to the header by first encoding it in *base64*, and then putting that data inside an `<iv/>` element. An example of the resulting `<message/>` stanza can be seen in listing 4.11.

Listing 4.11: An OMEMO message

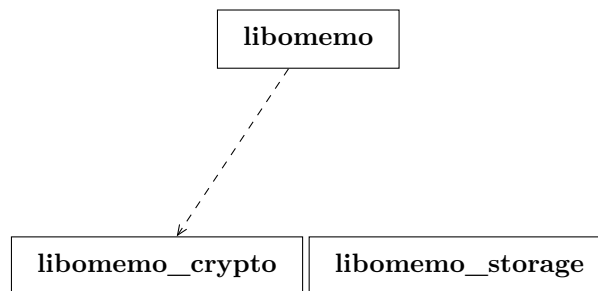
```
<message to="alice@example.com" from="bob@example.com">
  <encrypted xmlns="urn:xmpp:omemo:0">
    <header sid="654321">
      <key rid="123456"><!-- base64-encoded data --></key>
      <iv><!-- base64-encoded data --></iv>
    </header>
    <payload><!-- base64-encoded data --></payload>
  </encrypted>
</message>
```

The specification also describes `KeyTransportElements` for just transporting key material, but these are essentially the same as the just described `MessageElements`, only missing the `<payload/>`.

#### 4.4.2 Implementation Details

##### Design

Figure 4.7: libomemo modules



Similarly to *axc*, *libomemo* consists of three modules. However, as can be seen in figure 4.7, these do not explicitly interact, as the latter two are meant to be easily exchangeable example implementations.

Aside from initialization and teardown, `libomemo_crypto` only provides three functions: encrypting and decrypting using *AES-128* in *GCM*, and providing random bytes. These are implemented using *gcrypt*.

The `libomemo_storage` module implements the caching of devicelists mandated in the specification on one hand, and ‘chat lists’ which can be used for application-specific functions on the other hand. This is done by offering ‘save’, ‘get’, ‘delete’ and ‘exists’ functions working on a *SQLite* database. In order to simplify usage and make database initialization unnecessary, the optional creation of the simple database scheme is included in all query transactions.

Lastly, `libomemo` contains the main functionality, and to this end defines four data structures: `omemo_bundle`, `omemo_devicelist`, `omemo_message`, and `omemo_crypto_provider`. The first three obviously constitute an internal representation of abstract *OMEMO* types. Through those, dealing with incoming and outgoing instances of these types is assisted, as can be seen further below.

The last of these struct types is used to add a layer of indirection between the necessary cryptographic operations and the implementation, which is represented by the dotted line seen in figure 4.7. Usually, a client will want to set the contained function pointers to the implementations provided in `libomemo_crypto` when passing it to functions that require this struct. However, it can also choose to use an own implementation – the few needed functions can be seen in listing 4.12.

Listing 4.12: The `omemo_crypto_provider` struct

```
struct omemo_crypto_provider {
    int (*random_bytes_func)(uint8_t ** buf_pp, size_t buf_len, void * user_data_p);
    int (*aes_gcm_encrypt_func)(const uint8_t * plaintext_p, size_t plaintext_len,
                               const uint8_t * iv_p, size_t iv_len,
                               const uint8_t * key_p, size_t key_len,
                               size_t tag_len,
                               void * user_data_p,
                               uint8_t ** ciphertext_pp, size_t * ciphertext_len_p,
                               uint8_t ** tag_pp);
    int (*aes_gcm_decrypt_func)(const uint8_t * ciphertext_p, size_t ciphertext_len,
                                const uint8_t * iv_p, size_t iv_len,
                                const uint8_t * key_p, size_t key_len,
                                uint8_t * tag_p, size_t tag_len,
                                void * user_data_p,
                                uint8_t ** plaintext_pp, size_t * plaintext_len_p);

    void * user_data_p;
};
```

When *XML* data is to be passed to any of the functions (e.g. for creating the internal representation of a message), it is expected as a string. This is done in order to not force programs which use this library to depend on a specific library. Internally, *mxml* is used for dealing with this data.

## Devicelists

A devicelist can be either created from scratch by using `omemo_devicelist_create()`, or imported from an incoming *PEP* update message using `omemo_devicelist_import()`. In most cases, the devicelist will be imported from an incoming *PEP* update. The corresponding function expects the *XML* data to start at the `<items/>` element as it is considered the actual data of the update message, but then looks for the `<list/>` element which is the first *OMEMO* element inside it. This `<list/>` element is then ‘unhinged’ from its parent elements, and saved in the `list_node_p` pointer of the struct. As IDs are of `uint32_t` type, but come as character strings inside the *XML* data, the device IDs are parsed into the integer type and put into a separate linked list, which is saved under the `id_list_p` pointer, greatly simplifying handling. Since a devicelist’s owner’s

name is not contained in any of its elements, it needs to be taken from the `<message/>`'s `from` attribute and saved separately. The resulting struct used to represent devicelists is shown in listing 4.13.

If a new devicelist is to be constructed, aside from of course allocating the memory, the internal `<list/>` *XML* element also has to be created. The linked list can stay empty however, as a null pointer is a valid empty list in *glib*. For a client, creating a new devicelist is really only useful if it is the user's first *OMEMO* device and the first-time publishing of the devicelist *PEP* node has to be performed. Otherwise, the own devicelist is sent by the server in a *PEP* update like any other. But internally, the `libomemo_storage` module uses the constructor function in order to return this higher-level devicelist type when asked for a user's devicelist.

There exist several functions to work with existing `omemo_devicelist` structs. For one, they can be modified by `omemo_devicelist_add()` and `omemo_devicelist_remove()`. In addition, information can be retrieved, such as whether a devicelist is empty, contains a specific ID, or who the owner is. Often, it will be necessary to work on the list of IDs as integers, which is why a copy of the internal ID list can be retrieved using `omemo_devicelist_get_id_list()`. The `omemo_devicelist_diff()` function provides the possibility to compare two devicelists, for example a user's cached devicelist to an incoming one. Finally, the devicelist can be exported back to *XML* for publishing on the *PEP* node, but most of the time it will likely just be removed from memory by calling the `omemo_devicelist_destroy()` function. Even though the *PEP* node name is static and available as a constant, there is also a `omemo_devicelist_get_pep_node_name()` returning a string. This is to match the interface to the non-static bundle node name, as will be explained below.

Listing 4.13: The `omemo_devicelist` struct

```
struct omemo_devicelist {
    char * from;
    GList * id_list_p;
    mxml_node_t * list_node_p;
};
```

## Bundles

Unlike the devicelist, a bundle does not need to be worked with – it just exists to either parse incoming information, or publish data already collected by the *Double Ratchet* code. Because of this, the internal data is held as *XML* elements, as visible in listing 4.14: the `omemo_bundle` struct consists of four pointers to *XML* elements, the device ID as a character string, and the number of pre-keys contained in it. That way, the incoming bundle can simply be deconstructed into its parts when the struct is created, and the contained information decoded when needed. In turn, exporting the bundle to *XML* for sending to the server is made simple, as the elements contained in the struct just need to be put together.

As explained before, a device's bundle usually needs to be published when new pre-keys are generated, for example after the client has been newly installed. To do this,

the client first has to create an empty bundle by using `omemo_bundle_create()`. Then, its *Double Ratchet* database must be queried for all necessary data – the public identity key, the signed pre-key and its signature, a certain number of pre-keys, and the device ID. The retrieved information can be added to the `omemo_bundle` using the `omemo_bundle_set_*`() functions, which create the respective *XML* element and set the passed data as the element’s value after converting it to its *base64* representation. When that is done, `omemo_bundle_export()` can be used to get the bundle in *XML* format, starting at the `<publish/>` element as seen in listing 4.8. For exporting, first the three ‘outer’ elements `<publish/>`, `<item/>` and `<bundle/>` are created, and then the already created ‘inner’ elements are simply hooked in at the right place.

In order to create a *Double Ratchet* session, this bundle is then requested by other devices. The reply containing the bundle can be imported with `omemo_bundle_import()`. Inside this function, the inner elements containing the different keys are unhooked from the outer elements and saved as the corresponding struct member. These can then be retrieved using the `omemo_bundle_get_*`() functions. This group of functions simply decodes the *base64*-encoded data and returns a byte buffer. `omemo_bundle_get_random_pre_key()` is the exception, as it contains some additional logic to retrieve a random key, as the name says. For this reason the number of pre-key elements is also counted when importing. Since the bundle *PEP* node’s name contains the owning device’s ID, it cannot be a constant and has to be put together individually. For this case, the bundle interface provides the `omemo_bundle_get_pep_node_name()` utility function, which does exactly that.

Listing 4.14: The `omemo_bundle` struct

```
struct omemo_bundle {
    char * device_id;
    mxml_node_t * signed_pk_node_p;
    mxml_node_t * signature_node_p;
    mxml_node_t * identity_key_node_p;
    mxml_node_t * pre_keys_node_p;
    size_t pre_keys_amount;
};
```

## Messages

Most of the data objects handled by this library will be of the type seen in listing 4.15, as it represents normal messages which will be sent between users. Unlike the previous two types, it actually contains data which does not directly end up in the final *XML* representation. It is not intended that a client constructs an `omemo_message` from scratch – it is made from either an incoming or an outgoing message. The exception is the `KeyTransportElement`, whose creation is not triggered by user input as it does not contain a payload. This kind of message can be created by using `omemo_message_create()`, and in fact this function is also called internally when preparing an outgoing message for encryption. The constructor function generates the random key and initialization vector, saving pointers to these in the newly allocated struct that will be returned. Since the

*IV* is not going to be modified after this point, it is already *base64*-encoded and put in its `<iv/>` element inside the `<header/>` element at this point. Both of these *XML* elements are created for this purpose and naturally also saved under the appropriate struct member.

Listing 4.15: The `omemo_message` struct

```
struct omemo_message {
    mxml_node_t * message_node_p;
    mxml_node_t * header_node_p;
    mxml_node_t * payload_node_p;
    uint8_t * key_p;
    size_t key_len;
    uint8_t * iv_p;
    size_t iv_len;
    size_t tag_len;
};
```

So when `omemo_message_prepare_encryption()` is called, it first creates a new `omemo_message`, and then encrypts the plaintext contained in the passed `<message/>` using the key and *IV* contained in that struct. For this, it needs to use functions contained in an `omemo_crypto_provider`. This resulting ciphertext is then *base64*-encoded and set as the value of the newly created `<payload/>` element. The `<body/>` node which contained the plaintext is deleted from the `<message/>` stanza as it is not needed anymore, but the latter is saved for later reuse, including all child elements it still contains. Since *AES* in *GCM* is used for encryption of the payload, an authentication tag is also automatically generated. As noted in section 3.2.2, this tag was initially appended to the ciphertext, basically rendering it useless. Because of this, all implementations switched to appending it to the key data, including this one. In order to keep the interface simple, the authentication tag is simply appended to the key data inside the same buffer, so that the client does not have to concatenate this data itself. So when calling `omemo_message_get_key()`, the whole buffer is returned: the key used for encrypting the payload as well as the tag. The combined data should then be encrypted using the previously established *Double Ratchet* session with a recipient device. The encrypted key and tag can then be added to the *OMEMO* `<header/>` using `omemo_message_add_recipient()`, which takes the recipient device's ID as one of its arguments.

After this process is repeated for every recipient, the resulting message stanza can be exported as *XML* through `omemo_message_export_encrypted()`. Since the real work was already done, it is just a matter of creating the necessary `<encrypted/>` node, setting the saved `<header/>` and `<payload/>` elements as its children, and adding it to the `<message/>` node which was saved in the beginning. Optionally, a hint can be added which helps the receiving client and user to correctly handle such an encrypted message. This can be done by adding a `<body/>` with a standard message which will then be displayed to the user, an *XEP-0380: Explicit Message Encryption*<sup>1</sup> `<encryption/>` element, or both.

---

<sup>1</sup><https://xmpp.org/extensions/xep-0380.html>



When an *OMEMO* message arrives and is imported with the `omemo_message_prepare_decryption()` function, these optional *XML* elements are stripped again, if present. Additionally, `<header/>` and `<payload/>` elements are removed from the `<message/>` as its children, and pointers to all three are saved. The client can then access the information inside the `omemo_message` through functions which parse the contained *XML* data. This includes the sender's device ID or JID, and most importantly, the encrypted key and tag for the own device ID, which are also *base64*-decoded before being returned. After identifying the necessary *Double Ratchet* session from the retrieved information, it can be used to decrypt the key and tag. By then giving this data to `omemo_message_export_decrypted()`, the *OMEMO* message is transformed into a 'regular' message and returned as an *XML* string. The conversion happens by *base64*-decoding the value of the `<payload/>` element, decrypting the resulting ciphertext using the passed key and `omemo_crypto_provider`, and putting the plaintext inside a newly created `<body/>` node, which is then set as a child element of the previously saved `<message/>`.

## 4.5 lurch

This plugin can be found at <https://github.com/gkdr/lurch>.

### 4.5.1 Design

Combining *libomemo* for *XML* handling and *axc* for the necessary *Signal Protocol* sessions, *lurch* provides an *OMEMO* implementation for *libpurple* through a plugin. The resulting structure can be seen in figure 4.8.

The basic idea of the plugin is to make use of *libpurple*'s signals to intercept outgoing messages before they are sent, and incoming messages directly after they are received to encrypt and decrypt them respectively. In the background, the *PEP* and *IQ* interfaces of *libpurple*'s *XMPP* protocol plugin are employed to publish and retrieve the necessary information. A client-agnostic user interface for some necessary functionality is offered through the command interface.

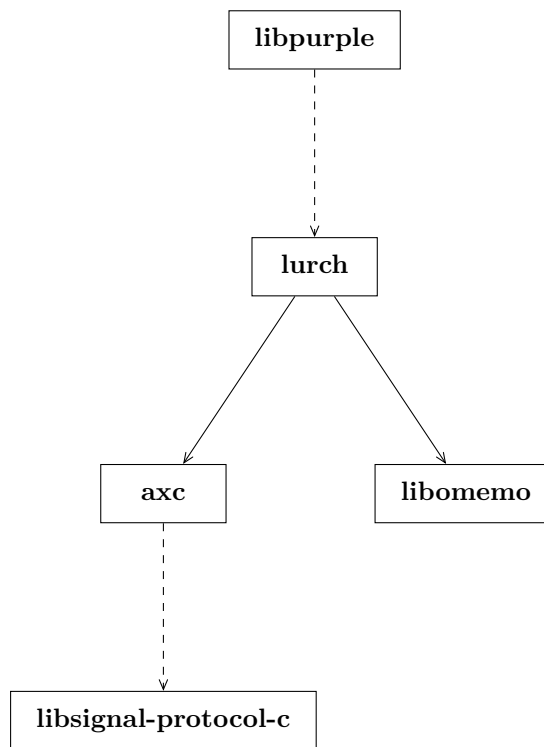
For most cases encountered during the development, the public types defined in the mentioned libraries were adequate. Still, two new types had to be added: `lurch_addr` is employed in order to not abuse `axc_address` and neatly divide the domains, and `lurch_queued_msgs` (seen in listing 4.16) are used to save all information necessary to hold back an outgoing message until sessions with all receiving devices could be established.

### 4.5.2 Implementation Details

#### Plugin Startup and Installation

When the plugin is to be loaded, *libpurple* calls the `lurch_plugin_load()` function which was supplied to it via the `PurplePluginInfo` struct. Aside from initializing the cryptography library, it registers the handlers for all signals, the devicelist *PEP* handler, and handlers

Figure 4.8: lurch design

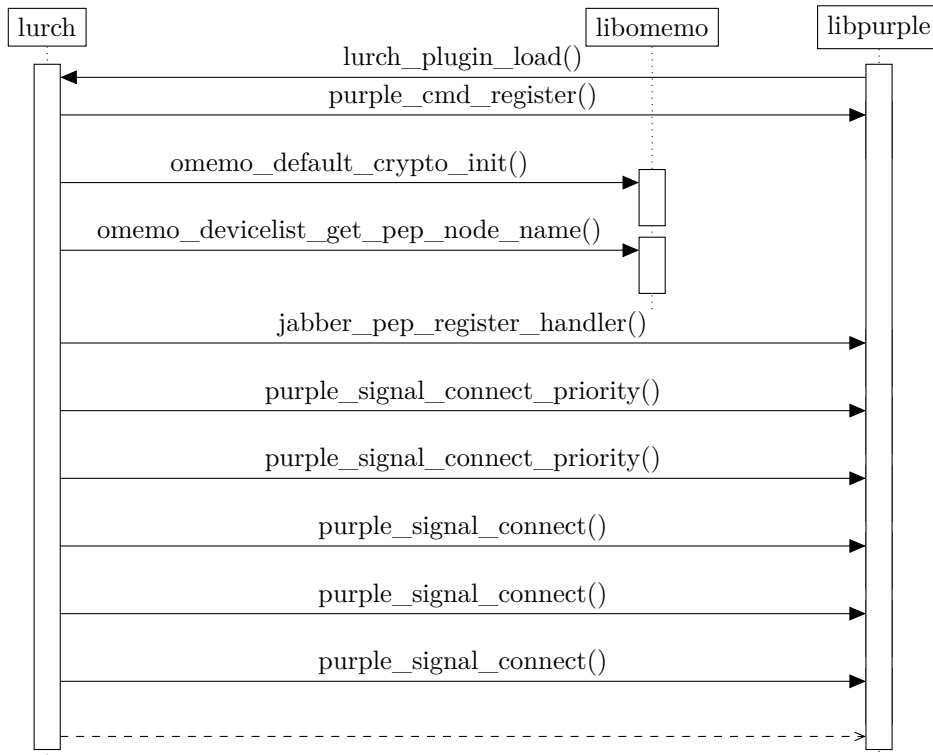


for the commands which are discussed in section 4.5.2. An important detail is that by calling `jabber_peg_register_handler()`, *libpurple* also automatically subscribes to updates of the named *PEP* node. The whole procedure can be observed in figure 4.9. The first two signals are `jabber-receiving-xmlnode` and `jabber-sending-xmlnode` for decryption and encryption, and are connected to using `purple_signal_connect_priority()` because it is important to register these handlers with a higher priority than usual so that most other processing is already finished by the time the stanzas reach these handlers (reminder: ‘higher’ priority means *later* handling in *libpurple*). For instance, the *XEP-0184: Message Delivery Receipts* plugin<sup>2</sup> only adds the receipt request to messages which have a `<body/>` element, but it is removed by *OMEMO*. In the other direction, the *XEP-0280: Message Carbons* ‘envelope’ needs to be stripped first when receiving. This is also the reason the *carbons* plugin is registered with a lower priority, meaning it will be processed first, as described in section 4.2. The latter three signals are connected to using the regular `purple_signal_connect()`. They are `conversation-created` and `conversation-updated` for updating the window title as it will be described in section 4.5.2, and more importantly to the `account-signed-on` signal which triggers the greater part of the actual *OMEMO* initialization. The callback function registered for this last signal is `lurch_account_connect_cb()`, which

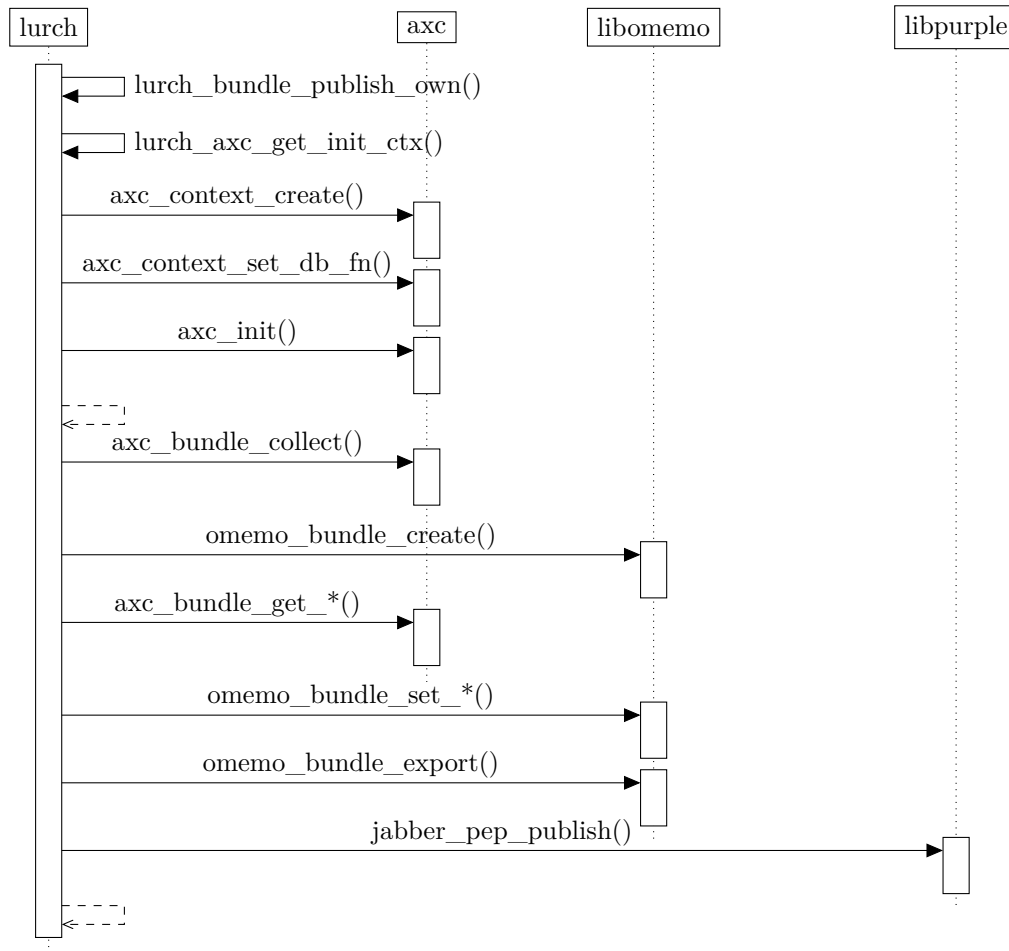
---

<sup>2</sup><https://app.assembla.com/spaces/pidgin-xmpp-receipts/>

Figure 4.9: Sequence diagram of lurch plugin load



simply checks if the connected account is an *XMPP* account, and in case it is requests that account's devicelist from the server. `lurch_pep_own_devicelist_request_handler()`, the handler for the previously mentioned request, is where the key and database initialization happens. By consulting an account option, this function checks whether the client is already initialized. If not, it calls `lurch_axc_prepare()`, which mostly just calls `axc_install()`, but also makes sure the generated device ID does not exist yet. Depending on whether the request returned a devicelist or not, i.e. if the client is the first *OMEMO* device of the account, the own ID is either added to the received list of devices, or to a newly created one, and then sent back to the server. All that is left now is publishing the bundle like it can be seen in figure 4.10. This is actually done at every startup to make sure the bundle node exists, since a server restart might also delete all saved *PEP* nodes. Afterwards, the *initialized* option is set to true. However, before cleaning up and returning, the `lurch_devicelist_process()` function is called on the just generated devicelist, whose purpose is described in the next section. An example of the whole process can be found in figure 4.11. It assumes that it is the first run of the client, and the client is the only *OMEMO* device.

Figure 4.10: Sequence diagram of `lurch_bundle_publish_own()`

## Devicelist Updates

The handler function for incoming devicelist updates is `lurch_peg_devicelist_event_handler()` and is not very complicated. It has a simple task: If the update is from the own account, it calls the previously described, more specific handler. Otherwise, it calls `lurch_devicelist_process()`, which simply updates the stored devicelist for a user with the incoming changes. To achieve this, the saved devicelist is recovered from the database using `omemo_storage_user_devicelist_retrieve()`, compared with the incoming devicelist by calling `omemo_devicelist_diff()`, and finally saving the added IDs, or deleting the removed ones. A sequence diagram for a situation which assumes a contact added a new device can be found in figure 4.12.

Figure 4.11: Sequence diagram of example lurch initialization

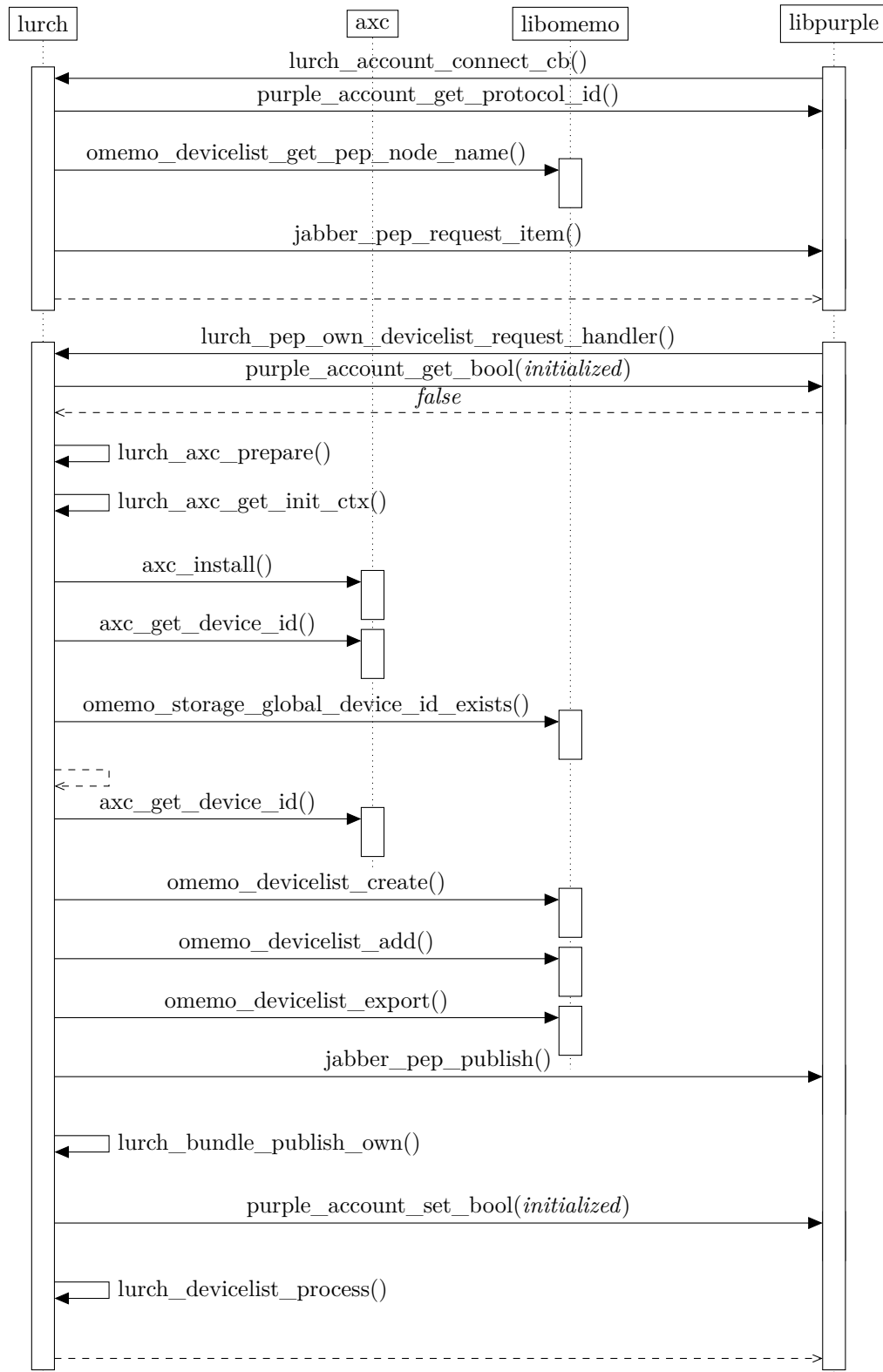
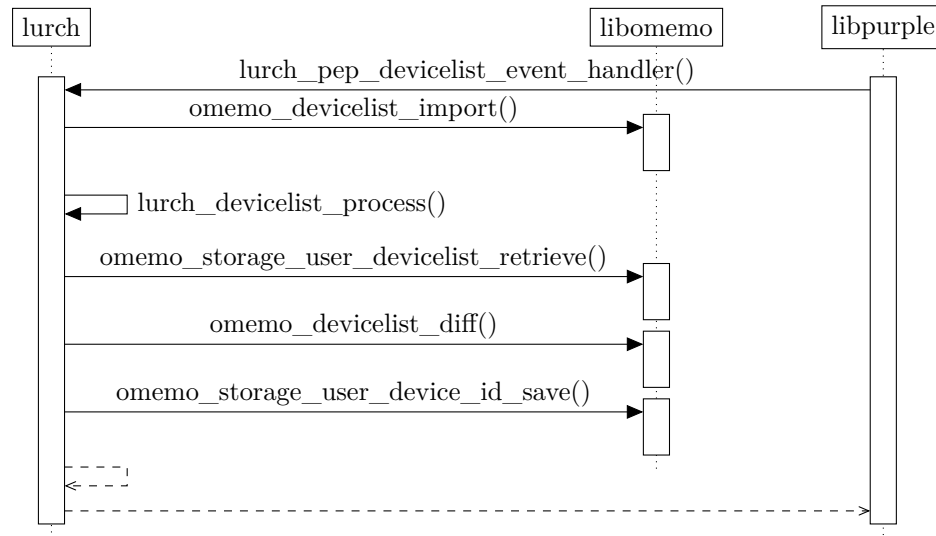


Figure 4.12: Sequence diagram of example devicelist handling



### Message Encryption

As previously described, the plugin ‘catches’ messages before they are sent in order to encrypt them. The registered handler function is `lurch_xml_sent_cb()`, but it only checks whether to further process the message by checking if it is a `<message/>` stanza with a `<body/>`. If this is the case, it then calls either `lurch_message_encrypt_im()` or `lurch_message_encrypt_groupchat()`, depending on the `type` attribute of the message (`chat` or `groupchat` respectively). The former case will now be inspected.

Before doing any processing, it is established whether the recipient is an *OMEMO* user by checking the stored devicelist using `omemo_storage_user_devicelist_retrieve()`. As *PEP* devicelist updates are received automatically and the changes written to the database by the handler, the devicelist returned at this point is current already and does not need to be requested from the server again. This also means that if there is no devicelist, the message recipient does not use *OMEMO* and the message does not need to be processed any further. Otherwise, a linked list of `lurch_addrs` is assembled, containing the full addresses of all *OMEMO*-using devices owned by both the recipient and the sender. This list is then passed to `lurch_msg_finalize_encryption()`, along with the imported `omemo_message`.

By first calling `lurch_axc_sessions_exist()`, this function finds out which of the devices passed to it do not have a session yet. In case they all do have a session already, a call to `lurch_msg_encrypt_for_addrs()` will encrypt the symmetric key contained in the passed `omemo_message` for all of them, after which the original `<message/>` stanza can be replaced by the output of `omemo_message_export_encrypted()`. The other case is a bit more involved, as the *OMEMO* specification dictates to only request bundles right before session establishment, so they cannot be simply retrieved from the database like the devicelists. Therefore, the bundle of each session-less device is requested using

`lurch_bundle_request_do()`. All data collected so far is saved in a `lurch_queued_msg` struct which is passed as data to this callback function. As can be seen in listing 4.16, the data which is necessary to continue processing at a later time is the imported message, the list of recipients, the list of recipients without a session, and a hashtable for marking recipients as handled.

Listing 4.16: The `lurch_queued_msg` struct

```
struct lurch_queued_msg {
    omemo_message * om_msg_p;
    GList * recipient_addr_l_p;
    GList * no_sess_l_p;
    GHashTable * sess_handled_p;
};
```

In order to be able to pass data such as this struct to a callback function, the more general *IQ* interface has to be used, as the *libpurple PEP* interface does not allow that. This is exactly what the bundle requesting function does. Finally, the message is held back from sending by setting it to `NULL`. When a response arrives, it is handled by `lurch_bundle_request_cb()`, which calls `lurch_bundle_create_session()` to create a session from the received bundle, and marks the device as ‘handled’ in the `lurch_queued_msg`. It is also marked as handled in case the server returns an error, e.g. because no bundle was uploaded, so that the process can still be finished and the message encrypted for the other, working devices. When all devices are ‘handled’, `lurch_msg_encrypt_for_addrs()` is called, as before. However, the final exported message cannot be simply passed on like in the previous case. Instead, the `jabber-sending-xmlnode` signal is emitted in order to send it. An example of the part leading up to the request can be seen in figure 4.13, while the handling of the response is depicted in 4.14. Both of the examples assume that the client wants to send a message to a device for which no session exists yet.

## Message Decryption

Like before, an incoming message is first passed to a very general handler function, in this case `lurch_xml_received_cb()`. If it does not contain an *OMEMO* `<encrypted/>` element, it does not do anything, but calls `lurch_message_warn()`. This function checks if a session with the sending device exists, and if so, warns the user of the unencrypted incoming message. For messages that do contain the `<encrypted/>` element, `lurch_message_decrypt()` is called for further processing. There, the message is first imported by calling `omemo_message_prepare_decryption()`. After retrieving the own device ID through `axc_get_device_id()`, the payload key encrypted for this client can be fetched from the parsed message by utilizing `omemo_message_get_encrypted_key()`. The encrypted payload key is nothing other than a *Signal Protocol* message. Unfortunately, *libsignal-protocol-c* cannot discern between a regular message and a pre-key message. Therefore, `axc_pre_key_message_process()` is always called first on the ciphertext. If it fails because it was not a pre-key message, and a session already exists (checked by `axc_session_exists_initiated()`), it must be a regular message, so `axc_message_decrypt_from_serialized()` is called on it. The decrypted payload key, can

Figure 4.13: Sequence diagram of example lurch bundle request

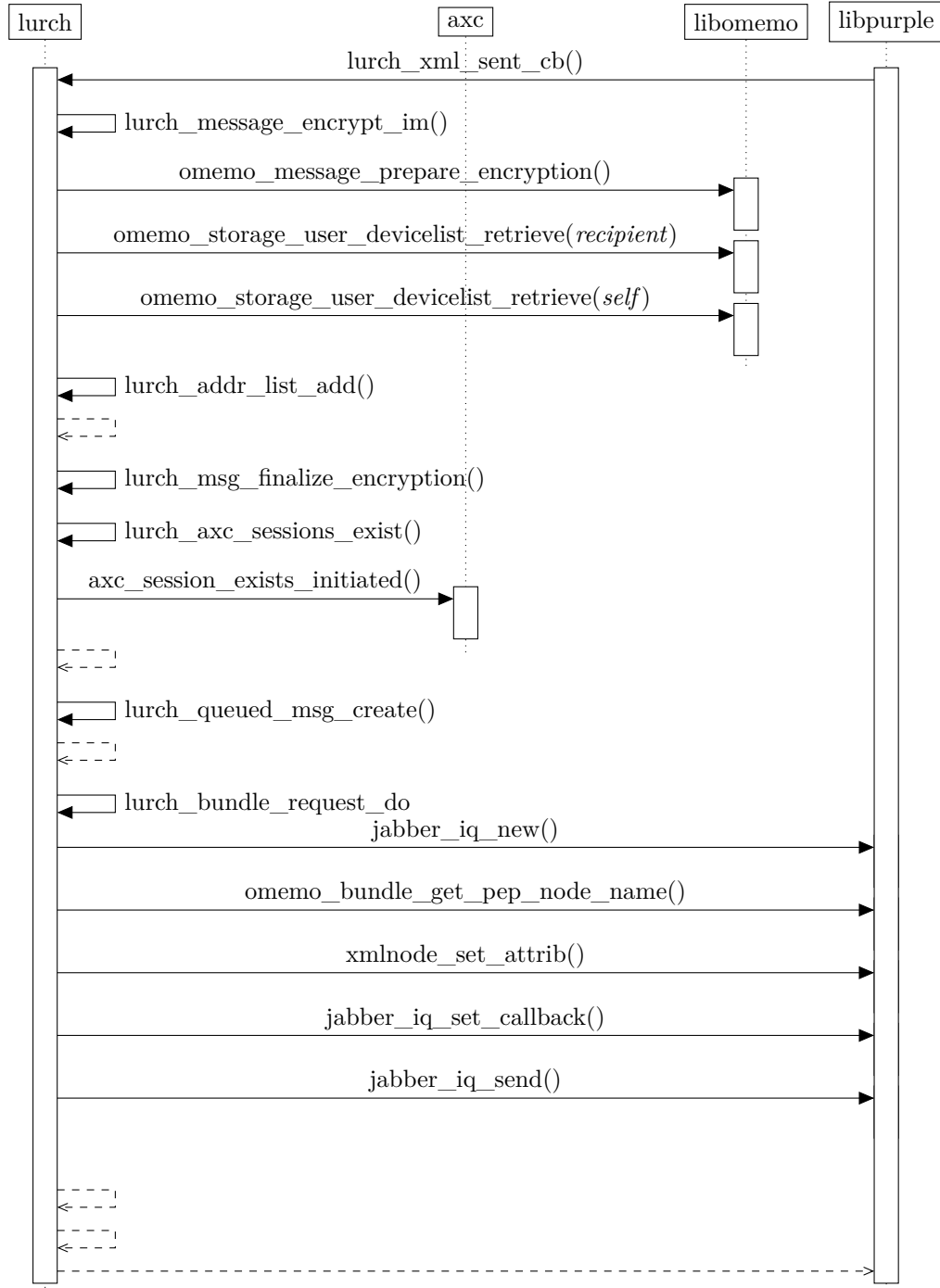
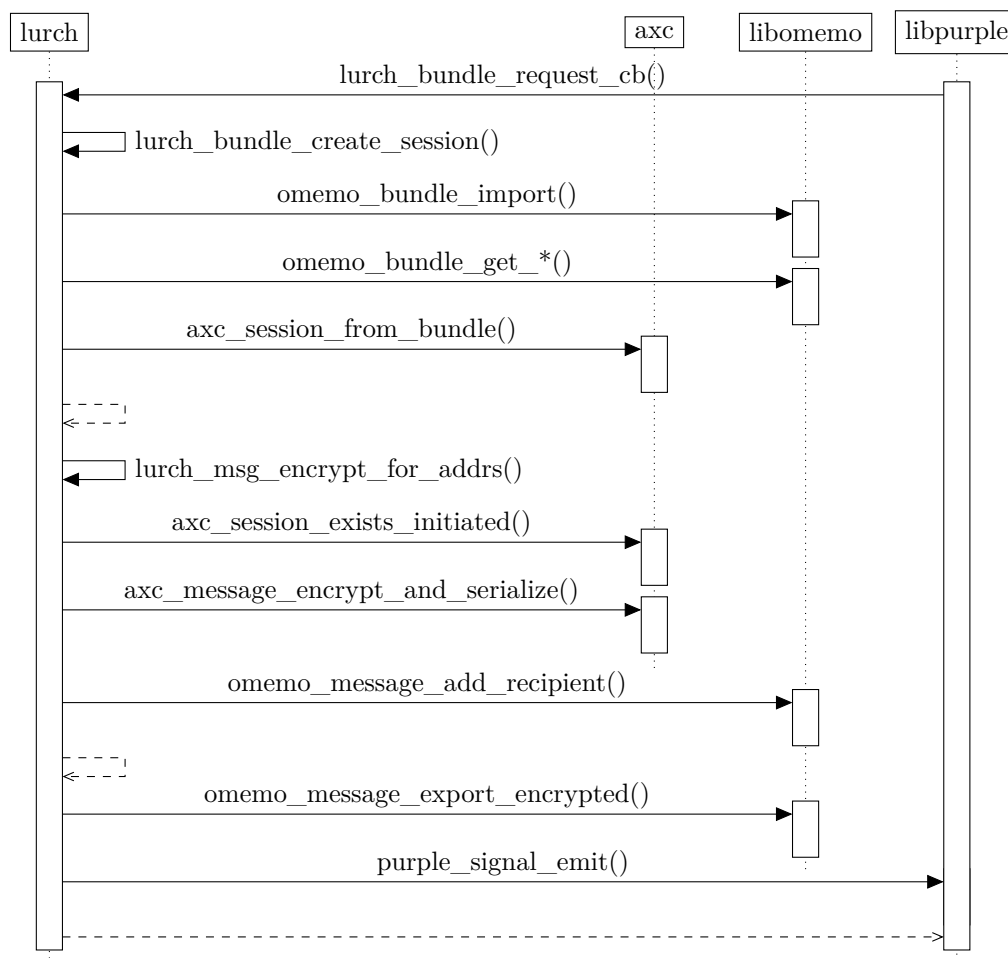




Figure 4.14: Sequence diagram of example lurch bundle request handling

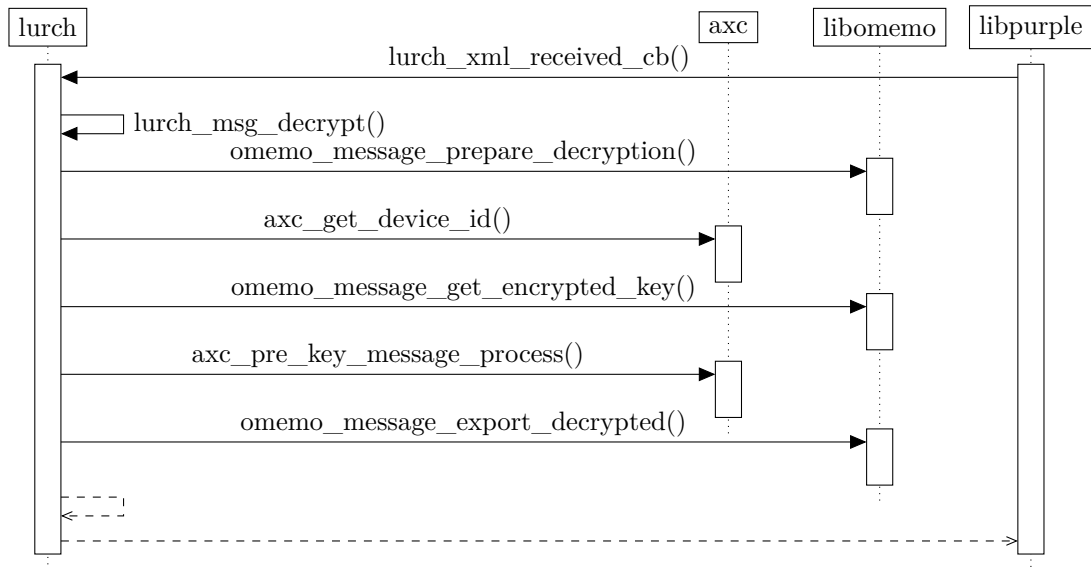


now be passed to `omemo_message_export_decrypted()`, which returns the decrypted message as discussed in section 4.4.2.

In case the message was sent by another account, this decrypted message can be simply passed on to *libpurple*, as it looks like a regular incoming message. However, in case the sender is the own account, as it can happen when the *carbons* plugin is also active, *libpurple* will not know what to do. When that happens, the decrypted plaintext is manually written to the conversation window, and the message then dropped.

Processing a pre-key message can also fail because the pre-key was used before and the keypair was already deleted. In this case, the contained text is unfortunately lost, but a working session can be set up automatically by requesting the bundle of the sending device and then sending a `KeyTransportMessage` after building the session locally. Unlike before, the more limited *PEP* interface can be used this time, so the bundle is requested using `jabber_peg_request_item()`. The handler function set for the response,

Figure 4.15: Sequence diagram of example lurch pre-key message decryption



`lurch_peg_bundle_for_keytransport()`, then does exactly what was previously described to establish a working session.

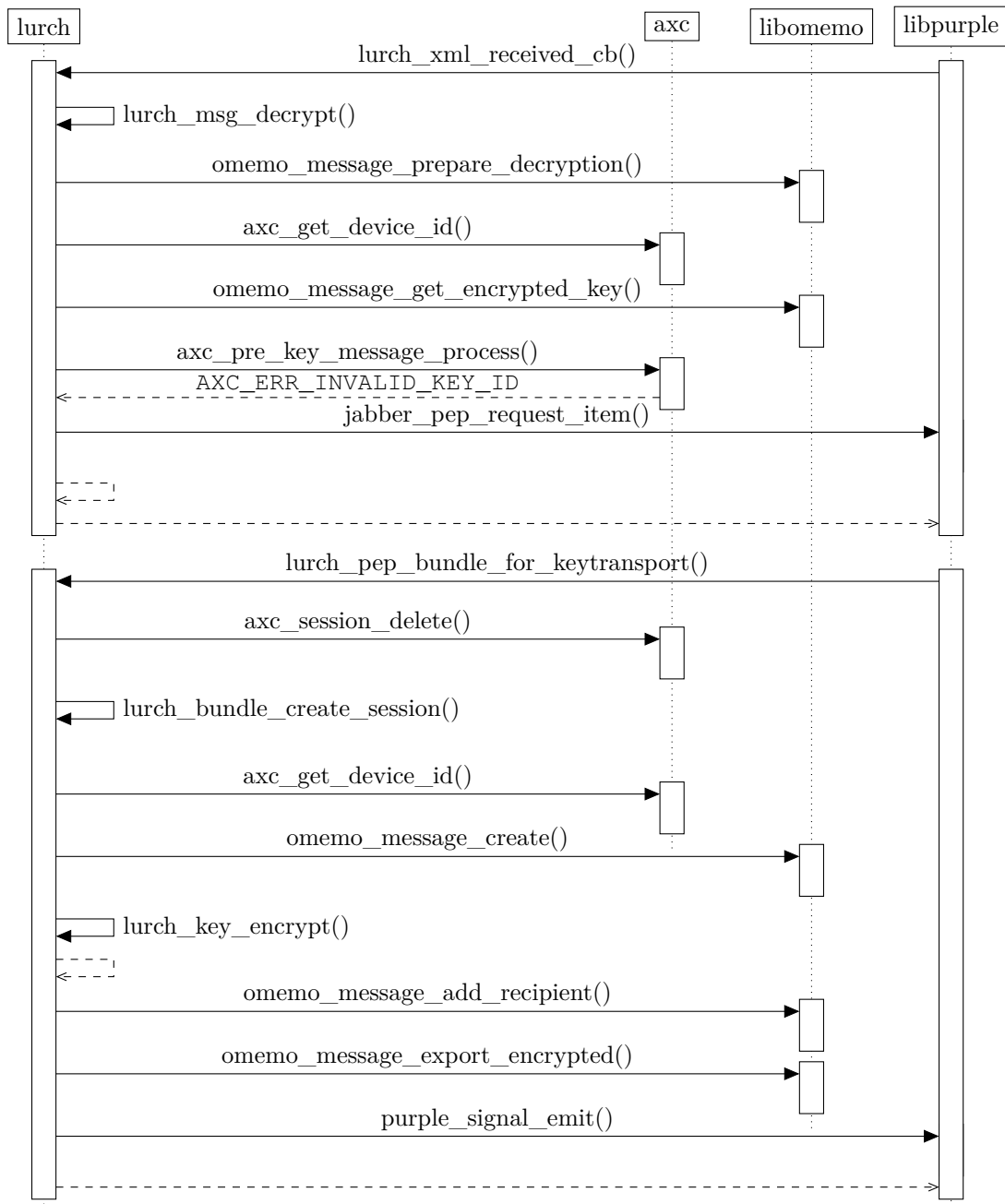
### Group Chats

Considering an *OMEMO* conversation between two users is essentially a group conversation between all of their devices, it is clear that in theory actual group chats using *OMEMO* are not hard to implement. However, they are not contained in the specification, as current limitations only allow for a very specific type of usage. *PEP* nodes can only be seen and queried by clients which are subscribed to their owner's presence, i.e. are in that user's 'buddy list' (*roster* in *XMPP* terms). However, the extension allowing group chats – *XEP-0045: Multi-User Chat*<sup>3</sup> – generally allows anonymous chats between strangers. In other words, it might not even be the case that the *JID* of other users in the chat room is known, as only the self-chosen alias is shown. Therefore, *OMEMO* encrypted group chats are only possible if the chatroom is set to *non-anonymous* which allows seeing *JIDs*, and every user is in every other user's roster so that the necessary *PEP* nodes can be accessed. Thus, the resulting realistic usage case is a chat between friends, rather than adding encryption to any MUC.

In addition to that, *libpurple* itself presents another implementation obstacle, as it does not fill its own `PurpleConvChatBuddy` struct used to represent a MUC participant with all available data. More specifically, even in non-anonymous rooms the *JID* is not available. This made it necessary to parse the incoming `<presence/>` stanzas manually, and keep the information in-memory. In fact, the previously described `lurch_xml_received_cb()` not

<sup>3</sup><https://xmpp.org/extensions/xep-0045.html>

Figure 4.16: Sequence diagram of example lurch session repair



only looks for *OMEMO* messages in order to decrypt them, but also for `<presence/>` stanzas, which are then passed to `lurch_presence_handle()`. This function checks whether the stanza contains any information it is interested in, i.e. an `<x/>` element in the `http://jabber.org/protocol/muc#user` namespace which includes an `<item/>` node having a `jid` attribute. These are received when a non-anonymous *MUC* room is joined by the client, or a new user joins the *MUC* later on. If the stanza does indeed contain the necessary data, the global hash table is consulted, which saves another hash table for every *MUC*. This second hash table contains the mapping between the alias used in the *MUC* and the actual *JID* of the user, and the received information is saved to it. After the preparatory work is done, messages can actually be encrypted. When `lurch_xml_sent_cb()` catches a message whose `type` attribute is `groupchat`, it calls `lurch_message_encrypt_groupchat()`, which just compiles a list of addresses to encrypt the message for before calling `lurch_msg_finalize_encryption()` on it, much like the function for regular conversations. The difference is of course that this has to happen for every *MUC* participant, and that a user's *JID* has to be retrieved from the room's hash table before the database can be queried for the devicelist. Message decryption is so similar to the two-party case that it is done in the same function, `lurch_message_decrypt()`. Determining the sender of the message is not as straightforward, however. The `from` attribute usually containing the sender's *JID* will usually look like `room@conference.example.com/useralias` in `<message/>`s of type `groupchat`, i.e. the *MUC*'s full name with the sender user's alias appended. Using this information, the hash table for the *MUC* can be found, and thus the user's *JID* from the alias. After the sender's *JID* is retrieved, decryption continues as usual.

### User Interface

Most of the user interface is not graphical in nature – the interaction happens inside a message window through commands. For instance, to display a help message which offers information about the available commands, the user can enter `/lurch help` in a message window, and the plugin will write the requested information to the same message window. Commands exist for enabling and disabling the plugin for single conversations or completely, displaying fingerprints and device IDs, and manually change the devicelist on the server.

However, there is one rather graphical feature as well: the plugin indicates whether a conversation is *OMEMO*-encrypted. As this had to be done without writing GUI code, the conversation's title was chosen to convey this information. To do this, *(OMEMO)* is appended to it. The original title is usually either the contact's *JID* or the alias the user chose. Because of how *libpurple* works, this is surprisingly complicated, as the title changes every time the conversation is focused. This is not noticable since the title stays the same, but it is updated constantly. Thus, like described in section 4.5.2, handlers for the `conversation-created` and `conversation-updated` signals are registered. `lurch_conv_created_cb()`, the handler for the former, checks whether the concerning conversation's protocol is *XMPP*, and depending on whether it is a regular or a group conversation, calls `lurch_topic_update_im()` or `lurch_topic_update_chat()`.

`lurch_conv_updated_cb()`, the handler for the update event, does essentially the same, but has to check whether the update is of type `PURPLE_CONV_UPDATE_TITLE`. The two functions for updating the title only differ in how it is determined whether *OMEMO* is used. For a single user, it is checked whether a session exists and is not blacklisted. Since encryption has to be manually activated for MUCs, for groupchats this setting is simply consulted.

## 4.6 Evaluation

The implementation will now be practically evaluated by performing actions of common usage cases. *Pidgin* offers a debug window which displays a multitude of helpful information and will be used to confirm the necessary actions are actually taken. This is possible because *lurch* makes extensive use of the debug interface. In the examples below, Alice registered the account `a@localhost` and added her friend Bob, `b@localhost`. Bob had previously installed the *lurch* plugin already and now tells Alice to do the same as he wants their communication to be encrypted end-to-end. After confirming that it works, Alice wishes to chat on her second device as well, so she also installs the *carbons* plugin.

### Startup

The first-time startup can be observed in the debug log seen in listing 4.17. As previously mentioned, the installation is triggered in the `lurch_pep_own_devicelist_request_handler()`, as the plugin needs to know whether there are other *OMEMO* devices in order to avoid using the same device ID. Aside from that, the necessary long- and short-term keypairs are also generated in this step.

As it is Alice's first device, the request for the own devicelist causes an empty response. A devicelist, which for clarity is shown separately in listing 4.18, is created and published. It also shows the namespace which is actually used for compatibility with other *OMEMO* clients and that it differs from the one specified in the *XEP*. To finish the installation, the bundle also shown separately in listing 4.19 is then published, but as described before this also happens at every startup. The plugin then begins to process the incoming *PEP* updates, the first of which is the own devicelist published a moment ago. Being subscribed to all devicelist updates, this is not surprising, and will become necessary when multiple devices connect to the same account. For now the second update coming from Bob is more important though, and can be observed in listing 4.20.

Listing 4.17: *lurch* initialization debug log

```
lurch: lurch_pep_own_devicelist_request_handler: preparing installation...
lurch: lurch_pep_own_devicelist_request_handler: ...done
lurch: lurch_pep_own_devicelist_request_handler: no devicelist yet, creating it
lurch: lurch_pep_own_devicelist_request_handler: devicelist needs publishing...
jabber: Sending (ssl) (a@localhost/24f3944e-6386-4676-b4d4-615dd9396cb5): ...
lurch: lurch_pep_own_devicelist_request_handler: ...done
jabber: Sending (ssl) (a@localhost/24f3944e-6386-4676-b4d4-615dd9396cb5): ...
lurch: lurch_bundle_publish_own: published own bundle for a@localhost
lurch: lurch_devicelist_process: processing devicelist from a@localhost
      for a@localhost
lurch: lurch_devicelist_process: cached devicelist is empty
lurch: lurch_devicelist_process: saving 942218035 for a@localhost to db
      /home/alice/.purple/a@localhost_omemo_db.sqlite
jabber: Recv (ssl) (332): ...
lurch: lurch_pep_devicelist_event_handler: a@localhost received devicelist update
      from b@localhost
lurch: lurch_devicelist_process: processing devicelist from b@localhost
      for a@localhost
lurch: lurch_devicelist_process: cached devicelist is empty
lurch: lurch_devicelist_process: saving 1877784789 for b@localhost to db
      /home/alice/a@localhost_omemo_db.sqlite
```

Listing 4.18: Alice's published devicelist

```
<iq type="set" id="purple69bf560a">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="eu.siacs.conversations.axolotl.devicelist">
      <item>
        <list xmlns="eu.siacs.conversations.axolotl">
          <device id="942218035"/>
        </list>
      </item>
    </publish>
  </pubsub>
</iq>
```

Listing 4.19: Alice's published bundle

```

<iq type="set" id="purple69bf560b">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="eu.siacs.conversations.axolotl.bundles:942218035">
      <item>
        <bundle xmlns="eu.siacs.conversations.axolotl">
          <signedPreKeyPublic signedPreKeyId="0">
            BQiNPqg4DKvukx1KjMfDmrYN/e1UqWMgRoSoIZn3BSop
          </signedPreKeyPublic>
          <signedPreKeySignature>
            6Tmd8RhyoVFuy8488xbi0cDQtGt1uHzFjMBGHMow2pDB
            viZSdq0KTdrKewMkIQttfWPnkMuasnfwUXgHUXDNCw==
          </signedPreKeySignature>
          <identityKey>
            BTd8erhvsyqq26nIkx/fJhJZ/zQRnbFjTJAulFoGN351
          </identityKey>
          <prekeys>
            <preKeyPublic preKeyId="1">
              BZ6WUUQbp4TDskbQxWI35uO6+PnhASSXsUo5qOA4LSsE
            </preKeyPublic>
            <!-- ... -->
            <preKeyPublic preKeyId="100">
              BXTi1l8dFzjY6b2DfdlZaBIUmABU9nFvN/UA0Ki/V7Me
            </preKeyPublic>
          </prekeys>
        </bundle>
      </item>
    </publish>
  </pubsub>
</iq>

```

Listing 4.20: The devicelist update received from Bob

```

<message type="headline" to="a@localhost/24f3944e-6386-4676-b4d4-615dd9396cb5"
  from="b@localhost">
  <event xmlns="http://jabber.org/protocol/pubsub#event">
    <items node="eu.siacs.conversations.axolotl.devicelist">
      <item id="1">
        <list xmlns="eu.siacs.conversations.axolotl">
          <device id="1877784789"/>
        </list>
      </item>
    </items>
  </event>
</message>

```

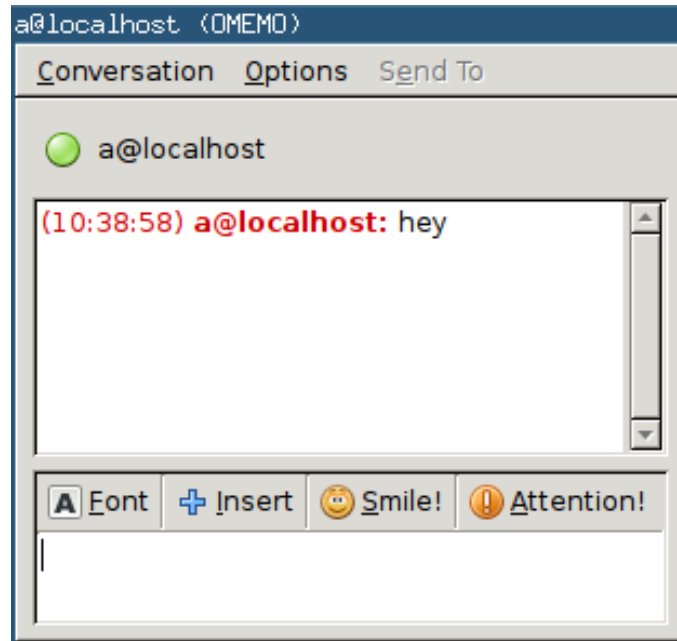
## Sending a message

Now that everything is set up, the client is ready to send and receive *OMEMO*-encrypted messages to and from *OMEMO*-using contacts. When Alice opens a message window with

Figure 4.17: The window title is informing Alice that *OMEMO* is available for Bob.



Figure 4.18: Bob receives the first encrypted message from Alice.



Bob, the plugin informs her via the window title that *OMEMO* is available, but there is no session yet. This can be observed in figure 4.17. She now sends him a simple first message - “hey”, which causes a number of events, as evident from the debug log of listing 4.21. The first two messages seen in the log are actually just ‘chat state’ notifications, which in this case inform Bob that Alice is typing something, and then stopped (because she hit ‘send’). Now, both the recipient’s and the own devicelist are retrieved in order to know which devices the message needs to be encrypted for. In this case it is only Bob’s single device with the ID 1877784789. As naturally there is no session with Bob’s device yet, the bundle needs to be requested. It can also be observed that the message was set to NULL in order to prevent sending of plaintext until all sessions are established. The rather big response containing the bundle is received in three parts, following which a session is established locally, and the held back message is sent. Since it needs to contain all the information Bob needs to compute the session as well, the *Signal Protocol*-encrypted key is quite long, as is evident in listing 4.22. On Bob’s side, the message is decrypted transparently and displayed like a regular one, however the window title still makes it apparent that the message was encrypted in transport. This is pictured in figure 4.18.



Listing 4.21: *lurch* first-time message sending debug log

```

jabber: Sending (ssl) (a@localhost/00798bb1-4f86-4e01-9631-5f894b2e550c):
<message type="chat" id="purple4c85762c" to="b@localhost">
  <composing xmlns="http://jabber.org/protocol/chatstates"/>
</message>
jabber: Sending (ssl) (a@localhost/00798bb1-4f86-4e01-9631-5f894b2e550c):
<message type="chat" id="purple4c85762d" to="b@localhost">
  <paused xmlns="http://jabber.org/protocol/chatstates"/>
</message>
conversation: typed...
lurch: retrieved devicelist for b@localhost:
<publish node="eu.siacs.conversations.axolotl.devicelist">
  <item>
    <list xmlns="eu.siacs.conversations.axolotl">
      <device id="1877784789" />
    </list>
  </item>
</publish>
lurch: retrieved own devicelist:
<publish node="eu.siacs.conversations.axolotl.devicelist">
  <item>
    <list xmlns="eu.siacs.conversations.axolotl">
      <device id="942218035" />
    </list>
  </item>
</publish>
lurch: lurch_msg_finalize_encryption: b@localhost has device without session
1877784789, requesting bundle
lurch: lurch_bundle_request_do: a@localhost/ is requesting bundle from
b@localhost:1877784789
jabber: Sending (ssl) (a@localhost/00798bb1-4f86-4e01-9631-5f894b2e550c):
<iq type="get" to="b@localhost" id="b@localhost#1877784789#-121318556">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <items node="eu.siacs.conversations.axolotl.bundles:1877784789"
      max_items="1" />
  </pubsub>
</iq>
lurch: lurch_bundle_request_do: ...request sent
g_log: xmlnode_to_str_helper: assertion "node_!=_NULL" failed
g_log: jabber_send_raw: assertion "data_!=_NULL" failed
jabber: Recv (ssl) (4095): ...
jabber: Recv (ssl) (4095): ...
jabber: Recv (ssl) (1171): ...
lurch: lurch_bundle_request_cb: a@localhost received bundle update from
b@localhost:1877784789
lurch: lurch_bundle_create_session: creating a session between a@localhost
and b@localhost from a received bundle
lurch: lurch_bundle_create_session: bundle's device id is 1877784789
lurch: lurch_msg_encrypt_for_addrs: trying to encrypt key for 1 devices
lurch: lurch_key_encrypt: encrypting key for b@localhost:1877784789
lurch: sending encrypted msg
jabber: Sending (ssl) (a@localhost/00798bb1-4f86-4e01-9631-5f894b2e550c): ...

```

Listing 4.22: The first encrypted message: “hey”

```
<message type="chat" id="purple4c85762e" to="b@localhost">
  <active xmlns="http://jabber.org/protocol/chatstates"/>
  <encrypted xmlns="eu.siacs.conversations.axolotl">
    <header sid="942218035">
      <key rid="1877784789">
        MwhIEiEFUZ3lNQ2vlrT0o9O33k9hn496wPSMaIalGxXjiHInIRcaIQU3fHq4b
        7MqqtupyJMf3yYSWf80EZ2xY0yQLpRaBjd+ZSJiMwobBWZxYU2UazWlJ5Bm12
        Ktc19Xj5/lmgwSvzZmm16+R+M5EAAyACIwbGOh+TSg5rVjhcBB0ggwGXWzKBR
        VDjJ6laV2vubwbpJTIjTcy790yqfzkLbkk23vtVCLMA9FEa8os7akwQMwAA==
      </key>
    </header>
    <iv>
      P0EbShgyHqiDxcoUzhZGWA==
    </iv>
  </encrypted>
  <payload>
    EYYT
  </payload>
  <encryption xmlns="urn:xmpp:eme:0"
    namespace="eu.siacs.conversations.axolotl"
    name="OMEMO"/>
  <store xmlns="urn:xmpp:hints"/>
</message>
```

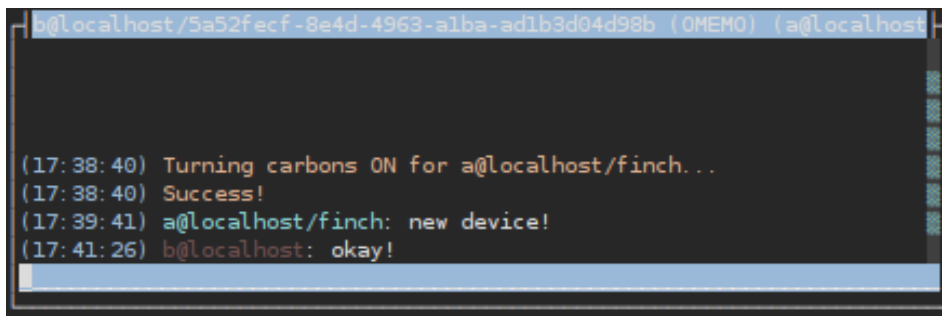
### Adding another device with *carbons*

After confirming that the plugin works, Alice wants to use *OMEMO* encryption on her second device as well, so in addition to *lurch* she also installs *carbons*. There is one difference, however: As no graphical interface is available, she runs *Finch* instead of *Pidgin* on her second device. Since it is also based on *libpurple*, the plugins work equally well. When Alice starts up *Finch*, it thus adds itself to Alice’s *OMEMO* devicelist. This can be observed on *Pidgin*’s debug log, as every client receives this update. The relevant debug log snippet is shown in listing 4.23.

Following the installation and activation of the *carbons* plugin, the message carbons feature has to be manually enabled by typing `/carbons on` in any message window. Alice does this on both devices, and informs Bob that she connected a new device. As can be seen in figure 4.19, *Finch* also automatically established an *OMEMO* session with Bob, notifying Alice about it through the window title. The screenshot in figure 4.20 shows *Pidgin* running at the same time, and displaying the same messages as *Finch*. It also shows a limitation of implementing *XEP-0280* as a plugin: *libpurple* does not offer the possibility to set a different sender name, which is why it displays the own randomly generated string identifying the *Pidgin* resource as a sender.

To investigate what exactly happened at that moment, it is necessary to take a look at the debug log produced by *Pidgin*, provided in listing 4.24. First, the simple process of activating message carbons can be seen. Then, a copy of the message sent on *Finch* is received, which is shown in 4.25. Inside the *carbons* ‘envelope’ is the message containing

Figure 4.19: Alice activates *carbons* and sends a message from her new device using *Finch*.



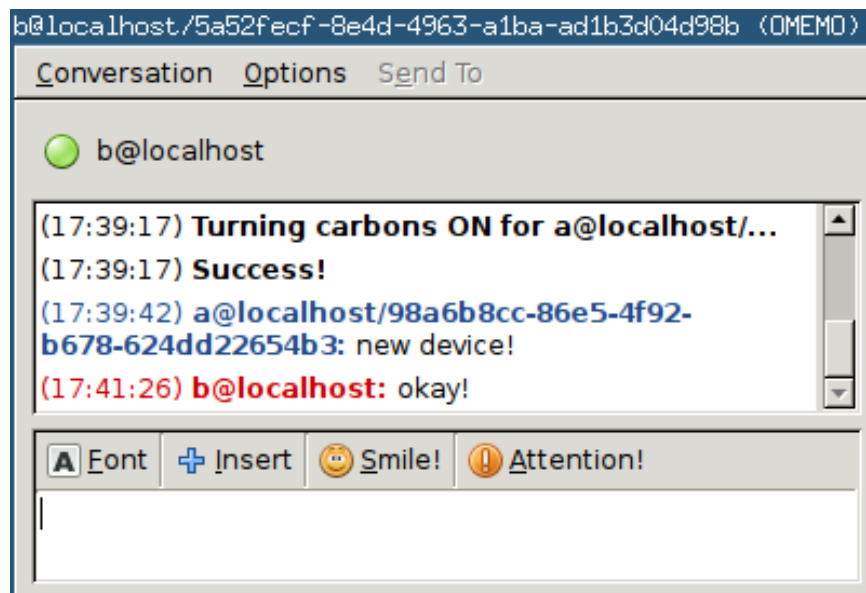
```

b@localhost/5a52fecf-8e4d-4963-a1ba-ad1b3d04d98b (OMEMO) (a@localhost)

(17:38:40) Turning carbons ON for a@localhost/finch...
(17:38:40) Success!
(17:39:41) a@localhost/finch: new device!
(17:41:26) b@localhost: okay!

```

Figure 4.20: Through *carbons*, both clients produce the same log of sent and received messages.



the text “new device!”, encrypted for both Bob and Alice’s other device. Because this was the first message received from the new device, it means a pre-key was used to establish a session. The corresponding key pair is deleted after use, replaced with a new one, and the new bundle containing the new public pre-key is published, which can be seen next. Afterwards, *Pidgin* receives a copy of Bob’s reply, presented in listing 4.26. This time, the copied message is inside a `<received/>` element instead of a `<sent/>` element. One other notable difference is the length of the data inside the `<key/>` element – as Bob already has sessions with both devices, the payload key is contained inside a regular *Signal Protocol* ciphertext message, instead of a pre-key message. Such a regular message is noticeably shorter.

Listing 4.23: Notified of a second device.

```
lurch: lurch_devicelist_process: processing devicelist from a@localhost
      for a@localhost
lurch: lurch_devicelist_process: cached devicelist is
      <publish node="eu.siacs.conversations.axolotl.devicelist">
        <item>
          <list xmlns="eu.siacs.conversations.axolotl">
            <device id="942218035" />
          </list>
        </item>
      </publish>
lurch: lurch_devicelist_process: saving 1679735306 for a@localhost to db
      /home/alice/.purple/a@localhost_omemo_db.sqlite
```

Listing 4.24: Receiving carbon-copied messages on *Pidgin*

```
jabber: Sending (ssl) (a@localhost/98a6b8cc-86e5-4f92-b678-624dd22654b3):
      <iq type="set" id="purple3596c3f6">
        <enable xmlns="urn:xmpp:carbons:2"/>
      </iq>
carbons: Sent enable request for a@localhost/
jabber: Recv (ssl) (93):
      <iq id="purple3596c3f6"
        type="result"
        to="a@localhost/98a6b8cc-86e5-4f92-b678-624dd22654b3"/>
jabber: Recv (ssl) (1194): ...
carbons: Received carbon copy of a sent message.
carbons: Carbon copy of sent message does not contain a body - stripping and passing it thr
jabber: Sending (ssl) (a@localhost/98a6b8cc-86e5-4f92-b678-624dd22654b3):
      <iq type="set" id="purple3596c3f8">
        <pubsub xmlns="http://jabber.org/protocol/pubsub">
          <publish node="eu.siacs.conversations.axolotl.bundles:942218035">
            <!-- ... -->
          </publish>
        </pubsub>
      </iq>
lurch: lurch_bundle_publish_own: published own bundle for a@localhost
jabber: Recv (ssl) (1007): ...
carbons: Received carbon copy of a received message.
```

Listing 4.25: The message sent on *Finch*, arriving on *Pidgin*

```

<message type="chat"
  to="a@localhost/98a6b8cc-86e5-4f92-b678-624dd22654b3"
  from="a@localhost">
  <sent xmlns="urn:xmpp:carbons:2">
    <forwarded xmlns="urn:xmpp:forward:0">
      <message type="chat"
        to="b@localhost"
        from="a@localhost/finch"
        id="purple4a81ef66"
        xmlns="jabber:client">
        <active xmlns="http://jabber.org/protocol/chatstates"/>
        <encrypted xmlns="eu.siacs.conversations.axolotl">
          <header sid="1679735306">
            <key rid="942218035">
              MwG8EiEFspIUFIh6X82TepBY1dj11krUhC0pkxKiUIrGv/wYXAaIQVAWlZqL
              BZA6Wcf4LZKa901zlo/sgcuQT0Qx5Yz+Yi+XyJiMwohBetJpyQ3LNX8LUF2T/
              zBV39l5l2sdgnxjQXfpz4o22UrEAAYACIwtD8wzU0kPnFcIZgl7yGRxWwj0+u
              uZImCjZZwccR4TOUo2NzOZWl7MgCFwaQAZjsZR7DeNQfuClYoivT6oAYwAA==
            </key>
            <key rid="1877784789">
              MwGgEiEFkaSzPDEnvX+CSvGfY83CL5pjiT5YMkipDUEwSqXPKCcaIQVAWlZqL
              BZA6Wcf4LZKa901zlo/sgcuQT0Qx5Yz+Yi+XyJiMwohBY1WDqyPRQVc3cxuwe
              Ubv/1x73OrNergo92ToRGJwPpUEAAYACIwGFcjgou/bAd2WtgOP5o6TqW4X9D
              vGkQx3TnqyhopX+7eliNQED8UW294puGxN3m9GKoIbna+T6IoivT6oAYwAA==
            </key>
            <iv>
              mdgn4e7+vai7zfwU+PQ0Lw==
            </iv>
          </header>
          <payload>
            owEO1lh12C/aFjg=
          </payload>
        </encrypted>
        <encryption namespace="eu.siacs.conversations.axolotl"
          name="OMEMO"
          xmlns="urn:xmpp:eme:0"/>
        <store xmlns="urn:xmpp:hints"/>
      </message>
    </forwarded>
  </sent>
</message>

```

Listing 4.26: The message sent by Bob and copied to both clients, as received by *Pidgin*

```
<message type="chat"
  to="a@localhost/98a6b8cc-86e5-4f92-b678-624dd22654b3"
  from="a@localhost">
  <received xmlns="urn:xmpp:carbons:2">
    <forwarded xmlns="urn:xmpp:forward:0">
      <message type="chat"
        to="a@localhost/finch"
        from="b@localhost/5a52fecf-8e4d-4963-alba-ad1b3d04d98b"
        id="purple3cb992ce"
        xmlns="jabber:client">
        <active xmlns="http://jabber.org/protocol/chatstates"/>
        <encrypted xmlns="eu.siacs.conversations.axolotl">
          <header sid="1877784789">
            <key rid="942218035">
              MwOhBaAYpOWJIIdYJx9pO2KdPAGEYbeCSEggi7GD1JLt
              ctRXEAAAYACIwnAL3Mc4Wpi1/ia50CqQMqZp6Fbk6ZjOy
              PA+8TcgeWHSbTa1Lrbr/BqB8amwRjGUpULKGmbx9CRU=
            </key>
            <key rid="1679735306">
              MwOhBVyrLnwAho4FhqOxr5JEBtDgivLW+/9y4DYp0rN4
              FFx+EAAAYACIwZvqQINCQftJhtaM5wzzm9QGxfAoVjvdW
              wx1luMKY6Nqwn05I7pyTajqx7H0KHZueFySFcCmN+dA=
            </key>
            <iv>
              2lA6KWGKwEWuMW9vla/Tpg==
            </iv>
          </header>
          <payload>
            wPOys50=
          </payload>
        </encrypted>
        <encryption namespace="eu.siacs.conversations.axolotl"
          name="OMEMO"
          xmlns="urn:xmpp:eme:0"/>
        <store xmlns="urn:xmpp:hints"/>
      </message>
    </forwarded>
  </received>
</message>
```

# Discussion

## 5.1 Conclusion

While more general overviews over ‘secure’ messaging like in [UDB<sup>+</sup>15] had already existed, this work picks the most relevant schemes and offers an in-depth comparison of the individual parts they comprise: session establishment, cryptographic ratchet, and trust establishment. The result is not only a verdict about the best scheme, but also an understanding of which assumptions impact design decisions, and which trade-offs exist in regard to protocol design.

Naturally, the ‘winner’ was still picked for the implementation. After compiling the additional requirements documented in the design chapter, a young *XMPP* extension protocol fulfilling these could be found: *OMEMO*. Even though it was still in draft phase, presenting a huge improvement over *OTR* it gathered a lot of attention. For example, this is evidenced by pages which track the inclusion in existing *XMPP* clients<sup>1</sup>. The *lurch* plugin implementing this protocol for the widespread *libpurple* messaging library is the result of this thesis’ practical part. At the moment of writing, it has 113 stars on *GitHub*, and appears in user-curated package repositories for the *Arch*<sup>2</sup> and *Exherbo Linux*<sup>3</sup> distributions. With some mutual assistance, it was made available for Windows<sup>4</sup> and MacOS (as a plugin for *Adium*)<sup>5</sup>. The hundreds of users helped with bug hunting, so that most of the open issue tickets are only feature requests. This work can therefore be considered successful in enabling and spreading modern, end-to-end encrypted messaging.

---

<sup>1</sup><https://omemo.top/>

<sup>2</sup><https://aur.archlinux.org/packages/libpurple-lurch-git/>

<sup>3</sup><https://git.exherbo.org/summer/packages/net-im/lurch/index.html>

<sup>4</sup><https://eion.robbsmob.com/lurch/>

<sup>5</sup><https://github.com/shtrom/Lurch4Adium>

## 5.2 Related Work

Throughout the thesis, comparisons have already been necessary. Since ‘privacy’ became somewhat of a selling point, there exist many messengers which claim to offer it. These have briefly been mentioned in section 2.1.4. As the reference implementation of the *Signal Protocol*, the *Signal Messenger* is inspected throughout chapter 2 and in section 3.1. The latter section also mentions the *Wire Messenger*, which uses its own implementation of the *Signal Protocol* as part of its own protocol. *libolm*, which was mentioned in section 2.3.2 for its *Megolm* group chat scheme, is an alternative, permissively licensed *Double Ratchet* implementation.

Because of *OMEMO*’s popularity, there is a multitude of projects intending to add it to existing *XMPP* messengers. A site which tracks this *XEP*’s progress has previously been mentioned<sup>6</sup>.

## 5.3 Future Work

There are some ways in which the *lurch* plugin itself could be improved. For instance, it is missing an interface for saving trust decisions at the moment. A graphical user interface has also been requested. More generally, work can be done to either add support for more *XEPs* to *libpurple*’s *XMPP* protocol plugin, or create plugins for these, as it happened for *Message Carbons*. Concretely, *Message Archive Management* could enable history synchronization between devices. On a more organizational level, a next version of the *OMEMO* protocol is currently being discussed on the *XMPP Standards Foundation*’s mailing list<sup>7</sup> and requires help from people who know about cryptography, protocol design, and ideally also the *XMPP* ecosystem. One specific problem currently being discussed is that the *Signal Protocol*’s *X3DH* key conversion algorithm does not have any alternative implementations, and is therefore only available under the *GPL* license. Adding this scheme to a permissively licensed crypto library and having it audited could thus be of help.

---

<sup>6</sup><https://omemo.top/>

<sup>7</sup><https://mail.jabber.org/pipermail/standards/>



# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Structure of a <i>PGP</i> encrypted message. . . . .   | 8  |
| 2.2  | Structure of a <i>PGP</i> signed message. . . . .  | 9  |
| 2.3  | Establishing the <i>DH</i> shared secret $g^{ab}$ between Alice and Bob. The short notation on the right will be useful later. . . . .                                 | 12 |
| 2.4  | An <i>OTR</i> ratchet example. . . . .   | 14 |
| 2.5  | <i>OTR</i> 's initial 'authenticated DH'. $A$ and $B$ are Alice's and Bob's <i>DSA</i> signing keypairs respectively. . . . .  | 15 |
| 2.6  | The basis for <i>OTR</i> 's improved handshake, the <i>SIGMA-R</i> authenticated key exchange protocol. . . . .  | 17 |
| 2.7  | Establishing the three <i>DH</i> shared secrets $g^{ab}$ , $g^{aB}$ , and $g^{Ab}$ . Short notation on the right. . . . .  | 19 |
| 2.8  | In <i>X3DH</i> , Bob's signed pre-key replaces his regular pre-key, which can then optionally be used for a fourth secret computed with Alice's ephemeral key. . . . . | 20 |
| 2.9  | The Double Ratchet. . . . .  | 23 |
| 2.10 | <i>np1sec</i> group key agreement between four users. . . . .  | 40 |
| 2.11 | Calculation of a <i>GOTR</i> circle key. . . . .   | 42 |
| 4.1  | <i>axc</i> modules . . . . .   | 58 |
| 4.2  | Sequence diagram of client initialization . . . . .  | 61 |
| 4.3  | Sequence diagram of example client installation . . . . .  | 62 |
| 4.4  | Sequence diagram of bundle collection . . . . .  | 63 |
| 4.5  | Sequence diagram of first-time sending . . . . .   | 65 |
| 4.6  | Sequence diagram of first-time receiving . . . . .   | 66 |
| 4.7  | <i>libomemo</i> modules . . . . .  | 70 |
| 4.8  | <i>lurch</i> design . . . . .  | 76 |
| 4.9  | Sequence diagram of <i>lurch</i> plugin load . . . . .   | 77 |
| 4.10 | Sequence diagram of <i>lurch_bundle_publish_own()</i> . . . . .  | 78 |
| 4.11 | Sequence diagram of example <i>lurch</i> initialization . . . . .  | 79 |
| 4.12 | Sequence diagram of example <i>devicelist</i> handling . . . . .   | 80 |
| 4.13 | Sequence diagram of example <i>lurch</i> bundle request . . . . .  | 82 |
| 4.14 | Sequence diagram of example <i>lurch</i> bundle request handling . . . . .   | 83 |
| 4.15 | Sequence diagram of example <i>lurch</i> pre-key message decryption . . . . .  | 84 |
| 4.16 | Sequence diagram of example <i>lurch</i> session repair . . . . .  | 85 |
|      |  | 99 |

|      |   |    |
|------|---|----|
| 4.17 | The window title is informing Alice that <i>OMEMO</i> is available for Bob. . . . .                 | 90 |
| 4.18 | Bob receives the first encrypted message from Alice. . . . .  | 90 |
| 4.19 | Alice activates <i>carbons</i> and sends a message from her new device using <i>Finch</i> . . . . . | 93 |
| 4.20 | Through <i>carbons</i> , both clients produce the same log of sent and received messages. . . . .   | 93 |

# List of Tables

|     |   |    |
|-----|---|----|
| 2.1 | General attributes of the evaluated protocols . . . . .       | 34 |
| 2.2 | Cryptographic attributes of the evaluated protocols . . . . . | 34 |

# Listings

|      |  |    |
|------|--|----|
| 4.1  | Enabling Message Carbons . . . . .   | 54 |
| 4.2  | A carbon-copied received message . . . . .   | 55 |
| 4.3  | A carbon-copied sent message . . . . .   | 55 |
| 4.4  | The contents of listing 4.2 with the outer message stripped . . . . .                      | 57 |
| 4.5  | The <code>axc_context</code> struct . . . . .  | 60 |
| 4.6  | The <code>axc_bundle</code> struct . . . . .   | 63 |
| 4.7  | A device adding itself to the previously empty <code>devicelist</code> <i>PEP</i> node . . | 67 |
| 4.8  | A device publishing public key information on its <code>bundle</code> <i>PEP</i> node . .  | 68 |
| 4.9  | Bob receiving Alice's updated <code>devicelist</code> . . . . .                            | 69 |
| 4.10 | Bob requesting the bundle of Alice's 123456 device . . . . .                               | 69 |
| 4.11 | An OMemo message . . . . .   | 70 |
| 4.12 | The <code>omemo_crypto_provider</code> struct . . . . .                                    | 71 |
| 4.13 | The <code>omemo_devicelist</code> struct . . . . .   | 72 |
| 4.14 | The <code>omemo_bundle</code> struct . . . . .   | 73 |
| 4.15 | The <code>omemo_message</code> struct . . . . .  | 74 |
| 4.16 | The <code>lurch_queued_msg</code> struct . . . . .   | 81 |
| 4.17 | <code>lurch</code> initialization debug log . . . . .                                      | 88 |
| 4.18 | Alice's published <code>devicelist</code> . . . . .  | 88 |
| 4.19 | Alice's published bundle . . . . .   | 89 |
| 4.20 | The <code>devicelist</code> update received from Bob . . . . .                             | 89 |
| 4.21 | <code>lurch</code> first-time message sending debug log . . . . .                          | 91 |

|      |  |    |
|------|--|----|
| 4.22 | The first encrypted message: “hey” . . . . .                                     | 92 |
| 4.23 | Notified of a second device. . . . .   | 94 |
| 4.24 | Receiving carbon-copied messages on <i>Pidgin</i> . . . . .                      | 94 |
| 4.25 | The message sent on <i>Finch</i> , arriving on <i>Pidgin</i> . . . . .           | 95 |
| 4.26 | The message sent by Bob and copied to both clients, as received by <i>Pidgin</i> | 96 |

# Bibliography

- [AB00] Michel Abdalla and Mihir Bellare. Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-Keying Techniques. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 546–559. Springer, 2000.
- [ACMP10] Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. Flexible Group Key Exchange with On-Demand Computation of Subgroup Keys. In *International Conference on Cryptology in Africa*, pages 351–368. Springer, 2010.
- [Ada] Adam Langley. Pond README. <https://github.com/agl/pond/blob/7bb06244b9aa121d367a6d556867992d1481f0c8/README.md>. Accessed on 2017-02-11.
- [AG07] Chris Alexander and Ian Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, pages 41–47, New York, NY, USA, 2007. ACM.
- [And] Andreas Straub. XEP-0384: OMEMO Encryption. <https://xmpp.org/extensions/xep-0384.html>.
- [ASZ96] Derek Atkins, William Stallings, and Philip Zimmermann. PGP Message Exchange Formats. RFC 1991, 1996. <https://tools.ietf.org/html/rfc1991>.
- [BB15] Julia Buxton and Tim Bingham. The rise and challenge of dark net drug markets. *Policy Brief*, 7, 2015.
- [BD95] Mike Burmester and Yvo Desmedt. A Secure And Efficient Conference Key Distribution System. In *Advances in Cryptology – EUROCRYPT'94*, pages 275–286. Springer, 1995.
- [Bee] Kristina Beer. Vorsicht beim skypen - microsoft liest mit. <http://www.heise.de/security/meldung/Vorsicht-beim-Skypen-Microsoft-liest-mit-1857620.html>. Accessed on 2017-05-20.

- [BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84. ACM, 2004.
- [BM] Joseph Bonneau and Andrew Morrison. Finite-State Security Analysis of OTR Version 2.
- [BMP04] Colin Boyd, Wenbo Mao, and Kenneth G Paterson. Key Agreement Using Statically Keyed Authenticators. In *International Conference on Applied Cryptography and Network Security*, pages 248–262. Springer, 2004.
- [BST01] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A Fair and Efficient Solution to the Socialist Millionaires Problem. *Discrete Applied Mathematics*, 111(1-2):23–36, 2001. Coding and Cryptology.
- [BST07] Jiang Bian, Remzi Seker, and Umit Topaloglu. Off-the-record instant messaging for group conversation. In *IEEE International Conference on Information Reuse and Integration, 2007.*, pages 79–84. IEEE, 2007.
- [BWJM97] Simon Blake-Wilson, Don Johnson, and Alfred Menezes. Key Agreement Protocols and their Security Analysis. In *IMA International Conference on Cryptography and Coding*, pages 30–45. Springer, 1997.
- [Car00] Germano Caronni. Walking the Web of Trust. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.(WET ICE 2000). Proceedings. IEEE 9th International Workshops on*, pages 153–158. IEEE, 2000.
- [CDF<sup>+</sup>07] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. OpenPGP Message Format. RFC 4880, 2007. <https://tools.ietf.org/html/rfc4880>.
- [CDFT98] Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer. OpenPGP Message Format. RFC 2440, 1998. <https://tools.ietf.org/html/rfc2440>.
- [CF11] Cas Cremers and Michele Feltz. One-Round Strongly Secure Key Exchange with Perfect Forward Secrecy and Deniability. *IACR Cryptology ePrint Archive*, 2011:300, 2011.
- [CGCD<sup>+</sup>16] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. Technical report, Cryptology ePrint Archive. International Association for Cryptologic Research (IACR), 2016.
- [CGCG16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On Post-Compromise Security. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 164–178. IEEE, 2016.

- [Chr09] Christopher Williams. UK jails schizophrenic for refusal to decrypt files. [http://www.theregister.co.uk/2009/11/24/ripa\\_jfl/](http://www.theregister.co.uk/2009/11/24/ripa_jfl/), 2009. Accessed on 2017-02-17.
- [Dan] Daniel Gultsch. commit f0c3b31a42ac6269a0ca299f2fa470586f6120be. <https://github.com/siacs/Conversations/commit/f0c3b31a42ac6269a0ca299f2fa470586f6120be>. Accessed on 2017-03-23.
- [Dav] David Wind and Christoph Rottermann. An Analysis of Signal-Desktop and WhatsApp Web.
- [Dec09] Declan McCullagh. Judge orders defendant to decrypt PGP-protected laptop. <https://www.cnet.com/news/judge-orders-defendant-to-decrypt-pgp-protected-laptop/>, 2009. Accessed on 2017-02-17.
- [DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, volume 22, pages 644–654. IEEE, 1976.
- [DRGK05] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Secure Off-The-Record Messaging. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, pages 81–89, New York, NY, USA, 2005. ACM.
- [DRGK06] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Deniable Authentication and Key Exchange. In *Proceedings of the 13th ACM conference on Computer and Communications Security*, pages 400–409. ACM, 2006.
- [ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.
- [eQua] eQualit.ie. (N+1)SEC. <https://learn.equalit.ie/wiki/Np1sec>. Accessed on 2017-04-17.
- [eQub] eQualit.ie. (n+1)sec protocol specification — draft. <https://github.com/equalitie/np1sec/blob/master/doc/protocol.pdf>. Accessed on 2017-04-17.
- [eQuc] eQualit.ie. (n+1)sec Test Report. <https://github.com/equalitie/np1sec/raw/master/doc/np1sec-test-report.pdf>. Accessed on 2017-05-01.
- [fai10] fail0verflow. Console Hacking 2010 – PS3 Epic Fail. [https://events.ccc.de/congress/2010/Fahrplan/attachments/1780\\_27c3\\_console\\_hacking\\_2010.pdf](https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf), 2010.

- [fip] fippo. No, it's not the end of XMPP for Google Talk. <https://xmpp.org/2015/03/no-its-not-the-end-of-xmpp-for-google-talk/>. Accessed on 2017-07-15.
- [FLK<sup>+</sup>13] Michael Farb, Yue-Hsun Lin, Tiffany Hyun-Jin Kim, Jonathan McCune, and Adrian Perrig. Safeslinger: Easy-to-Use and Secure Public-Key Exchange. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 417–428. ACM, 2013.
- [Flo] Florian Schmaus and Dominik Schürmann and Vincent Breitmoser. XEP-0374: OpenPGP for XMPP Instant Messaging. <https://xmpp.org/extensions/xep-0374.html>.
- [FMB<sup>+</sup>14] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Joerg Schwenk, and Thorsten Holz. How secure is textsecure? Cryptology ePrint Archive, Report 2014/904, 2014. <http://eprint.iacr.org/2014/904>.
- [Geoa] Georg Lukas. CVE-2017-5589+ Multiple XMPP Clients User Impersonation Vulnerability. [https://rt-solutions.de/de/2017/02/cve-2017-5589\\_xmpp\\_carbons-2/](https://rt-solutions.de/de/2017/02/cve-2017-5589_xmpp_carbons-2/). Accessed on 2017-03-25.
- [Geob] George Kadianakis. flute Specification. [https://github.com/asn-the-goblin-slayer/flute/blob/master/flute\\_spec.txt](https://github.com/asn-the-goblin-slayer/flute/blob/master/flute_spec.txt). Accessed on 2017-05-01.
- [Gle13] Glenn Greenwald and James Ball. The top secret rules that allow NSA to use US data without a warrant. <https://www.theguardian.com/world/2013/jun/20/fisa-court-nsa-without-warrant>, 2013. Accessed on 2017-02-17.
- [GM] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google and others. <http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data>. Accessed on 2015-11-20.
- [GUVGC09] Ian Goldberg, Berkant Ustaoglu, Matthew D Van Gundy, and Hao Chen. Multi-Party Off-the-Record Messaging. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 358–368. ACM, 2009.
- [H D08] H D Moore. Debian OpenSSL Bug. <https://hdm.io/tools/debian-openssl/>, 2008. Accessed on 2016-10-11.
- [Jak] Jake Edge. The perils of federated protocols. <https://lwn.net/Articles/687294/>. Accessed on 2017-03-20.



- [Jam] Jamie Zawinski. Signal. <https://www.jwz.org/blog/2017/03/signal-leaks-your-phone-number-to-everyone-in-your-contacts/>. Accessed on 2017-06-25.
- [JO15] Jakob Jakobsen and Claudio Orlandi. *A practical cryptanalysis of the Telegram messaging protocol*. PhD thesis, Master Thesis, Aarhus University (Available on request), 2015.
- [Joe] Joe Hildebrand and Matthew Miller. XEP-0280: Message Carbons. <https://xmpp.org/extensions/xep-0280.html>.
- [JY96] Markus Jakobsson and Moti Yung. Proving without knowing: On oblivious, agnostic and blindfolded provers. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 186–200, London, UK, UK, 1996. Springer-Verlag.
- [KAF<sup>+</sup>10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K Lenstra, Emmanuel Thomé, Joppe W Bos, Pierrick Gaudry, Alexander Kruppa, Peter L Montgomery, Dag Arne Osvik, et al. Factorization of a 768-bit RSA Modulus. In *Annual Cryptology Conference*, pages 333–350. Springer, 2010.
- [KP05] Caroline Kudla and Kenneth G Paterson. Modular Security Proofs for Key Agreement Protocols. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 549–565. Springer, 2005.
- [KR02] Vlastimil Klíma and Tomáš Rosa. Attack on private signature keys of the openpgp format, pgp (tm) programs and other applications compatible with openpgp. *IACR Cryptology ePrint Archive*, 2002:76, 2002.
- [Kra03] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003.
- [Kra05] Hugo Krawczyk. HMQV: A High-Performance Secure Diffie-Hellman Protocol. In *Annual International Cryptology Conference*, pages 546–566. Springer, 2005.
- [KY03] Jonathan Katz and Moti Yung. Scalable Protocols for Authenticated Group Key Exchange. In *Annual International Cryptology Conference*, pages 110–125. Springer, 2003.
- [LVH13] Hong Liu, Eugene Y Vasserman, and Nicholas Hopper. Improved Group Off-The-Record Messaging. In *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society*, pages 249–254. ACM, 2013.

- [Man09] Mark Manulis. Group Key Exchange Enabling On-Demand Derivation of Peer-to-Peer Keys. In *International Conference on Applied Cryptography and Network Security*, pages 1–19. Springer, 2009.
- [Mas12] Stephen Mason. Electronic Signatures in Law, 3rd Edition. pages 229–213, 318–322. Cambridge University Press, 2012. Accessed on 2017-02-18.
- [mat] matrix.org. Megolm group ratchet. <https://matrix.org/docs/spec/megolm.html>. Accessed on 2017-04-17.
- [Mau96] Ueli Maurer. Modelling a Public-Key Infrastructure. In *European Symposium on Research in Computer Security*, pages 325–350. Springer, 1996.
- [Men] Joseph Menn. Exclusive: Yahoo secretly scanned customer emails for u.s. intelligence - sources. <http://www.reuters.com/article/us-yahoo-nsa-exclusive-idUSKCN1241YT>. Accessed on 2017-05-20.
- [Moxa] Moxie Marlinspike. Advanced Cryptographic Ratcheting. <https://whispersystems.org/blog/advanced-ratcheting/>. Accessed on 2016-10-08.
- [Moxb] Moxie Marlinspike. Facebook Messenger deploys Signal Protocol for end to end encryption. <https://whispersystems.org/blog/facebook-messenger/>. Accessed on 2017-03-12.
- [Moxc] Moxie Marlinspike. Forward Secrecy for Asynchronous Messages. <https://whispersystems.org/blog/asynchronous-security/>. Accessed on 2016-10-07.
- [Moxd] Moxie Marlinspike. Just Signal. <https://whispersystems.org/blog/just-signal/>. Accessed on 2017-03-09.
- [Moxe] Moxie Marlinspike. Open Whisper Systems partners with Google on end-to-end encryption for Allo. <https://whispersystems.org/blog/allo/>. Accessed on 2017-03-12.
- [Moxf] Moxie Marlinspike. Private Group Messaging. <https://whispersystems.org/blog/private-groups/>. Accessed on 2017-04-17.
- [Moxg] Moxie Marlinspike. Safety number updates. <https://whispersystems.org/blog/safety-number-updates/>. Accessed on 2017-03-12.
- [Moxh] Moxie Marlinspike. Saying goodbye to encrypted SMS/MMS. <https://whispersystems.org/blog/goodbye-encrypted-sms/>. Accessed on 2016-10-06.

- [Moxi] Moxie Marlinspike. Signal on the outside, Signal on the inside. <https://whispersystems.org/blog/signal-inside-and-out/>. Accessed on 2017-03-09.
- [Moxj] Moxie Marlinspike. Simplifying OTR deniability. <https://whispersystems.org/blog/simplifying-otr-deniability/>. Accessed on 2016-10-06.
- [Moxk] Moxie Marlinspike. WhatsApp’s Signal Protocol integration is now complete. <https://whispersystems.org/blog/whatsapp-complete/>. Accessed on 2017-03-12.
- [Mox16] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol. <https://whispersystems.org/docs/specifications/x3dh/>, 2016. Accessed on 2017-03-05.
- [Nat01] National Institute of Standards and Technology. Advanced Encryption Standard (AES), 2001.
- [Nat10] Nate Lawson. DSA requirements for random k value. <https://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/>, 2010. Accessed on 2016-10-11.
- [Nat13] National Institute of Standards and Technology. Digital Signature Standard (DSS), 2013.
- [Opea] Open Whisper Systems. libsignal-protocol-c. <https://github.com/WhisperSystems/libsignal-protocol-c>.
- [Opeb] Open Whisper Systems. Signal-Server. <https://github.com/WhisperSystems/Signal-Server>.
- [OTR] OTR Development Team. Off-the-Record Messaging Protocol version 3. <https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html>. Accessed on 2016-09-25.
- [Pau] Paul Grosso and Daniel Veillard. XML Fragment Interchange. <https://www.w3.org/TR/xml-fragment>.
- [Peta] Peter Millard and Peter Saint-Andre and Ralph Meijer. XEP-0060: Publish-Subscribe. <https://xmpp.org/extensions/xep-0060.html>.
- [Petb] Peter Saint-Andre. XEP-0118: User Tune. <https://xmpp.org/extensions/xep-0118.html>.
- [Petc] Peter Saint-Andre and Kevin Smith. XEP-0163: Personal Eventing Protocol. <https://xmpp.org/extensions/xep-0163.html>.

- [RAZS15] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client. *arXiv preprint arXiv:1510.08555*, 2015.
- [Roe10] Michael Roe. Cryptography and Evidence. Technical Report UCAM-CL-TR-780, University of Cambridge, Computer Laboratory, May 2010.
- [Rus] Dominic Rushe. Google: don't expect privacy when sending to gmail. <https://www.theguardian.com/technology/2013/aug/14/google-gmail-users-privacy-email-lawsuit>. Accessed on 2017-05-20.
- [SA11a] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Address Format. RFC 6122, 2011. <https://xmpp.org/rfcs/rfc6122.html>.
- [SA11b] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, 2011. <https://xmpp.org/rfcs/rfc6120.html>.
- [SA11c] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121, 2011. <https://xmpp.org/rfcs/rfc6121.html>.
- [SBKH06] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J. Hyland. Why Johnny Still Can't Encrypt: Evaluating the Usability of Email Encryption Software, 2006.
- [Seb] Sebastian Verschoor. OMEMO: Cryptographic Analysis Report. <https://conversations.im/omemo/audit.pdf>.
- [Sil12] Silent Circle. What is Silent Phone? (archived). <https://web.archive.org/web/20160304091202/https://support.silentcircle.com/customer/en/portal/articles/2118686-what-is-silent-phone->, 2012.
- [Ste] Steff. Facebook schaltet XMPP API ab! <https://www.jabber.de/facebook-schaltet-xmpp-api-ab/>. Accessed on 2017-07-15.
- [Ste06] Marc Stevens. Fast collision attack on md5. *IACR Cryptology ePrint Archive*, 2006:104, 2006.
- [str] strb. Introduce ODR, improve specification. <https://github.com/xsf/xeps/pull/460>. Accessed on 2017-07-22.
- [sur] surespot. how surespot works. [https://www.surespot.me/documents/how\\_surespot\\_works.html](https://www.surespot.me/documents/how_surespot_works.html). Accessed on 2017-03-14.
- [SYG08] Ryan Stedman, Kayo Yoshida, and Ian Goldberg. A User Study of Off-The-Record Messaging. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 95–104. ACM, 2008.

- [Tel] Telegram. Secret chats, end-to-end encryption. <https://core.telegram.org/api/end-to-end>. Accessed on 2017-03-14.
- [The] The Free Software Foundation. The GNU Privacy Handbook – Encrypting and decrypting documents. <https://www.gnupg.org/gph/en/manual.html#AEN111>. Accessed on 2017-04-17.
- [Thr17] Threema. Threema Cryptography Whitepaper. [https://threema.ch/press-files/cryptography\\_whitepaper.pdf](https://threema.ch/press-files/cryptography_whitepaper.pdf), 2017.
- [Tim] Tim Bray and Jean Paoli and C. M. Sperberg-McQueen and Eve Maler and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). <https://www.w3.org/TR/xml/>.
- [Toxa] Tox Wiki. Frequently Asked Questions. <https://wiki.tox.chat/users/faq>. Accessed on 2017-05-25.
- [Toxb] Tox Wiki. Technical FAQ. <https://wiki.tox.chat/users/techfaq>. Accessed on 2017-05-25.
- [toxc] toxcore Github Issues. How does the toxcore work, is there a brief and complete description about how does it work in detail? <https://github.com/irungentoo/toxcore/issues/1637>. Accessed on 2017-05-25.
- [Tre16a] Trevor Perrin. The XEdDSA and VEdDSA Signature Schemes. <https://whispersystems.org/docs/specifications/xeddsa/>, 2016. Accessed on 2017-03-06.
- [Tre16b] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. <https://whispersystems.org/docs/specifications/doubleratchet/>, 2016. Accessed on 2017-02-20.
- [UDB<sup>+</sup>15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure Messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249. IEEE, 2015. [http://www.jbonneau.com/doc/UDBFPGS15-IEEEESP-secure\\_messaging\\_sok.pdf](http://www.jbonneau.com/doc/UDBFPGS15-IEEEESP-secure_messaging_sok.pdf).
- [UHC11] Alexander Ulrich, Ralph Holz, Peter Hauck, and Georg Carle. Investigating the OpenPGP Web of Trust. In *European Symposium on Research in Computer Security*, pages 489–507. Springer, 2011.
- [Vin12] Vinnie Moscaritolo and Gary Belvin and Phil Zimmermann. Silent Circle Instant Messaging Protocol Specification (archived). [https://web.archive.org/web/20150402122917/https://silentcircle.com/sites/default/themes/silentcircle/assets/downloads/SCIMP\\_paper.pdf](https://web.archive.org/web/20150402122917/https://silentcircle.com/sites/default/themes/silentcircle/assets/downloads/SCIMP_paper.pdf), 2012.

- [Wha] WhatsApp Developers. WhatsApp Encryption Overview. <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>. Accessed on 2017-04-17.
- [Whi] Whisper Systems Development Team. Accouncing The Public Beta. <https://web.archive.org/web/20100530011131/http://www.whispersys.com/updates.html>. Accessed on 2016-10-06.
- [Wir] Wire Developer Team. Wire Server. <https://github.com/wireapp/wire-server>.
- [WT99] Alma Whitten and J Doug Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Usenix Security*, volume 1999, 1999.
- [xmp] An Overview of XMPP. <https://xmpp.org/about/technology-overview.html>.
- [Yao82] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, SFCS '82*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.
- [Zima] Philip Zimmerman. PGP Marks 10th Anniversary. [https://www.philzimmermann.com/EN/news/PGP\\_10thAnniversary.html](https://www.philzimmermann.com/EN/news/PGP_10thAnniversary.html). Accessed on 2016-09-20.
- [Zimb] Philip Zimmerman. Why I Wrote PGP. <https://www.philzimmermann.com/EN/essays/WhyIWrotePGP.html>. Accessed on 2016-09-20.