FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Security Analysis of the Signal Protocol
Student:	Bc. Jan Rubín
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2018/19

Instructions

1) Research the current instant messaging protocols, describe their properties, with a particular focus on security.
2) Describe the Signal protocol in detail, its usage, structure, and functionality.
3) Select parts of the protocol with a potential for security vulnerabilities.
4) Analyze these parts, particularly the adherence of their code to their documentation.
5) Discuss your findings. Formulate recommendations for the users.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 27, 2018

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Master's thesis

Security Analysis of the Signal Protocol

Bc. Jan Rubín

Supervisor: Ing. Josef Kokeš

1st May 2018

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my thesis supervisor, Ing. Josef Kokeš, for his guidance, engagement, extensive knowledge, and willingness to meet at our countless consultations. I would also like to thank my brother, Tomáš Rubín, for proofreading my thesis. I cannot express enough gratitude towards my parents, Lenka and Jaroslav Rubínovi, who supported me both morally and financially through my whole studies. Last but not least, this thesis would not be possible without Anna who relentlessly supported me when I needed it most.

Declaration

In Prague on 1st May 2018 . . . .

Citation of this thesis

Rubín, Jan. *Security Analysis of the Signal Protocol.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato diplomová práce se zabývá studiem protokolu Signal. Zaměřuje se především na použitou kryptografii, funkcionalitu a strukturu protokolu. Práce dále obsahuje analýzu zdrojových kódů oficiální implementace a porovnává stav protokolu s jeho dokumentací. Práce také diskutuje potenciální bezpečnostní slabiny protokolu a formuluje jejich zmírnění či odstranění.

Klíčová slova protokol Signal, bezpečnostní analýza, instant messaging, bezpečná komunikace, Double Ratchet, forward secrecy, výměna klíčů

Abstract

This thesis provides a security analysis of the Signal Protocol. The protocol's cryptography, functionality, and structure are discussed. The source codes of the official implementation are analyzed and the protocol's state is compared with the documentation. Finally, the protocol's potential security vulnerabilities are examined and their mitigation or removal is formulated.

Keywords Signal protocol, security analysis, instant messaging, secure communication, Double Ratchet, forward secrecy, key exchange

Contents

List of Figures

List of Tables

Introduction

Instant messaging (IM) is a very popular form of online communication. It allows people from all over the world to connect with each other in real-time, drawing inspiration from the written word. It is an alternative to the well-known email communication which provides a more formal method of correspondence.

The first IM applications started to emerge in the mid-1960s and were originally limited to the local area networks. When the Internet began to grow, IM applications expanded to the global scope and IM communication evolved significantly.

However, with the spread of this trend, people started to realize that not every communication platform is secure. This privacy awareness started to grow since many cases of mass surveillance and communication interceptions from government organizations were published. Edward Snowden, a former NSA employee, exposed one of the biggest mass surveillance cases which caused an increase in privacy awareness among internet users.

In this thesis, we provide a brief overview of the IM protocols which are frequently used or which were very popular during their era. We focus on both unsecured protocols and very robust security solutions, providing an insight into the commonly used cryptographic principles, such as the forward secrecy, the asymmetric cryptography, the Diffie-Hellman key exchange or both the client-server and the end-to-end encryption schemes.

In the second chapter, we analyze the supposedly most secure IM protocol on the planet – the Signal Protocol – in detail. We provide an in-depth description of the protocol, the theoretical understanding of its functionality along with the mathematical background. We explain how these preliminaries improve the security of the cryptographic operations and we describe the ideas behind the protocol which can be actually used separately in other implementations as well.

In the third chapter, we analyze the official implementation of the Signal Protocol libraries which are used in the Signal for Android application. The analysis is based on the theoretical knowledge of the protocol, inspecting the particular implementation and its specifications. We compare the Signal Protocol implementation to the official documentation and we discuss any present discrepancies and undocumented specifications.

Finally, we formulate security considerations regarding the protocol's design and we describe how to mitigate the vulnerabilities or how to remove them completely.

Chapter 1

Security Status of IM Protocols

During the growth of the Internet, many developers worked hard on new instant messaging (IM) protocols which would connect people over the network.

Currently, countless IM protocols are used all over the world which can transmit data from one side to another over the network – either safely using the cryptography methods, or by unsecured communication mostly in a plain text form.

To provide a general perspective, we decided to present a few diametrically different protocols from less secure ones (e.g. the IRC protocol in its original design) to the very robust security solutions (e.g. the Signal Protocol). Furthermore, we decided to select some frequently used protocols for the security inspection.

The OTR was one of the first protocols which supported end-to-end encryption and it also popularized the concept of the forward secrecy, as opposed to the Pretty Good Privacy protocol (PGP). Its design allowed it to be used as a plugin to other IM clients as well.

Jabber was very popular during the spread of the IM communication. It provided a free and open alternative to the proprietary IM services of the day, such as ICQ or AOL Instant Messenger [1].

On the other hand, the MTProto is a very robust protocol which is currently used in the Telegram instant messenger. It has its own cipher system design which is partly based on well known cryptographic standards. It presents two types of encryption approaches – client-server and end-to-end encryption.

1.1 IRC

Internet Relay Chat (IRC) is a client-server protocol which supports real-time communication with people from all over the world.

IRC was created by Jarkko Oikarinen in August 1988. In cooperation with Darren Reed, he proposed a Request for Comments (RFC) 1459 [2] from May

1993 regarding this protocol. Even though this RFC presents the IRC as an *experimental* protocol, it became more popular every year and it is active to this day. It is designed to mainly provide a group communication in so called *channels*, but it can be used for one-to-one communication as well. However, all the communication always goes through a server.

The typical setup involves a server forming the central point for the clients (or other servers) to connect to [3]. Each server has to have a copy of the global state information. This is a limiting factor for the maximum reachable size of the network and it is heavily hardware dependent.

In comparison to the Simple Mail Transfer Protocol (SMTP) or the Extensible Messaging and Presence Protocol (XMPP), IRC uses a multicast when sending a message. This means the message travels to the server only once and it is then redistributed to all recipients.

1.1.1 Security Considerations

The communication over the IRC protocol was originaly unencrypted [2]. In addition, client can be a very simple socket program capable of connecting to the server [4]. This means that the original design is very simple with almost no restrictions on the client side.

Because the IRC protocol is centralized, it can be more vulnerable to denial-of-service attacks (DoS), sometimes called "nukes". The typical result of the DoS attack made network computers disconnect or crash [5].

TLS/SSL connection support was later implemented. The server should listen on port 6697 for any incomming secured communications [6]. If the client connects to this port, a standard TLS/SSL handshake should take place. The tunnel secures the communication between the client and the server.

However, any client connected to the same channel can see all the messages. If the channel is not TLS/SSL restricted, any client who does not use a secured tunnel exposes the plaintext communication from server to her, effectively invalidating all the encryption.

1.1.1.1 Malicious Servers

The standard structure of the network (which is containted of the IRC servers) is a *spanning tree* [3]. Each server acts as a central node for the rest of the network. Every message is routed only through necessary branches, but the network state is sent to every server. This results in a very limited scalability of the IRC network. In addition, every connection between two servers is a serious single point of failure [3].

There is generally a high degree of trust between servers. Furthermore, every server assumes a neighbor server is in the correct state, e.g. its database is consistent [2].

If a server is buggy, misbehaving or malicious, it can cause serious damage to the IRC network. For example, if the connection between the two servers is interrupted, additional network traffic is generated because a network split and a network join is performed. This can result in a complete network congestion or a temorary loss of communication between the users [3].

1.2 OTR

Off-the-Record (OTR) was one of the first protocols supporting end-to-end encryption. It was designed to provide the perfect forward secrecy along with encryption of the contents [7].

The name “Off-the-Record” originates in journalism. Sometimes, a journalist wanted to know more in-depth information about a topic which was considered confidential and thus the source of the information was not published. Following this idea, the OTR was designed to provide private communication with a deniable authentication, digital signatures, and more.

The design of the OTR protocol allows it to be used as a plugin to the already existing IM clients, such as GAIM[1] [7]. Although the OTR was designed mainly for the IM communication, it can be actually used in the email communication as well – using so called *ring signatures.*

1.2.1 Cryptography

The OTR significantly improves a previously unsecured IM communication. In order to do so, it presents a few cryptographic approaches.

To ensure communication is secured, a message has to be encrypted. AES is used for a symmetric encryption/decryption with 128-bit keys. The key is a shared secret which is established using the standard Diffie-Hellman key agreement.

As previously mentioned, OTR presents perfect forward secrecy. Thus, the symmetric keys which are used for encryption and decryption have to be deleted as soon as possible, i.e. immediately after the client is sure that they will not be used at any time in the future. After the old key was deleted, a new key is established using the Diffie-Hellman agreement. The client has to handle out-of-order messages, too, because the messages could be lost or delayed in transit. This potentially weakens the forward secrecy, but the risk is not too high with a proper balance of storage capacity and timeouts to reflect a common traffic loss.

Both parties have to be authenticated. A message authentication code (MAC) is used for authenticating each message. To generate a MAC key, a one-way hash function is applied to the decryption key [7]. This makes it impossible for Eve (an eavesdropper) to convince anyone else that it was Alice

[1]GAIM was a Linux client which is currently called Pidgin.

or Bob and not her who wrote the message (if we assume that Eve could somehow decrypt the message in the first place). SHA-1 is used as a hash function for HMAC (hash-based MAC).

The initial Diffie-Hellman exchange is authenticated as well. For this purpose, digital signatures are created using a standard RSA with long-lived private and public signature keys [7].

Note that the public keys should be verified out-of-band using another communication channel (e.g. in person with a fingerprint written on a piece of paper). Without it, there is absolutely no cryptographic guarantee of the authenticity of both parties because the communication could already be intercepted using a man-in-the-middle attack.

1.3 Jabber

The Extensible Messaging and Presence Protocol (XMPP), which was previously called Jabber, is the communication protocol based on the XML (Extensible Markup Language). Jabber was invented by Jeremie Miller in 1998 and it was later formalized as the XMPP by the Internet Engineering Task Force (IETF) as an Internet Standard for messaging and presence [8]. It supports both one-on-one communication and multi-party messaging [1].

Unlike most of IM protocols, the XMPP is an *open standard*. This means that anyone can implement an XMPP service with the usage of an arbitrary software license. Thus, it can be used for example in the internal communication in a wide organization.

The XMPP is the client-server protocol with a decentralized network model. This means that there is no central (master) server and anyone can run their own server that can be isolated from the public network. This is a very effective solution, e.g. for a company intranet [9].

On the Jabber network, every client is identified by the Jabber ID (JID). This ID is structured as a username and a domain name (i.e. similar to the email addresses) [8].

1.3.1 Cryptography

The XMPP had not been secured for many years. In the original implementation, the XMPP communication used open-ended XML streams over long-lived TCP connections and all messages were in an unencrypted text form. This also meant that the communication had a higher network overhead compared to purely binary solutions. The overhead was later mitigated by serialization methods [10].

1.3.1.1 Signing and Encrypting Messages

Later on, the standard TLS/SSL encryption was introduced to the XMPP which allows to encrypt the communication between *hops*, i.e. the client-server or the server-server communication [11]. Thus, the communication is not end-to-end encrypted. However, the XMPP developer community is actively working on end-to-end encryption to raise the security bar even further [9].

The XML messages can be digitally signed and/or encrypted. In order for the message to be sent, the encrypted contents have to be encapsulated in a XML CDATA section [12]. Thus, the recipient's address (JID) is visible to everyone but the contents of the message are secured by the encryption. In the XMPP, standard AES-128 in CBC mode is used for encryption and the RSA is used for key transport [12].

The signing process is performed using the standard S/MIME which provides the authentication, message integrity and non-repudiation [13]. For example, Jabber.org uses the Let's Encrypt certification authority [14, 8]. Because the Let's Encrypt issuing certificate is bundled with many platforms and applications, Jabber can be used on many devices. The RSA with SHA-1 signature algorithm is used for signing the data [12].

Furthermore, every message should contain a timestamp as a countermeasure to possible replay attacks. A client application must verify that the received timestamp is within five minutes of the current time and the timestamp should be also greater than any previously received timestamp in the last ten minutes (which passed the previous check) [12].

1.3.1.2 Signing and Encrypting Presence Information

As mentioned, the XMPP is also the *presence*[2] protocol. However, the attacker could take advantage of this information. Thus, the presence information should be signed and/or encrypted as well [12]. Because the presence information is also the XML object, the process is very similar to the one described in section 1.3.1.1.

1.4 MTProto

MTProto protocol allows end-to-end encryption of the IM communication. It is currently used in the Telegram Messenger which is available for mobile phones and desktops alike. Users can transmit messages, files and other data such as audio and video calls.

The Telegram team decided to create a brand new IM protocol which uses some of well known cryptography standards (e.g. encryption using AES), but it also represents a new encryption scheme as a whole. The reasoning behind

[2]The *presence* is the ability to see if the other participant is online.

this is based on supposedly better reliability on weak mobile connections as well as better delivery times [15].

Even though the client-side code is open-source, all codes are not published and the server side is closed-source (i.e. proprietary) [16]. However, the Telegram team says that all source codes will eventually be published.

The MTProto contains two types of encryption *layers*. The first is a client-server encryption and the second is a client-client encryption. The second one is end-to-end encrypted. However, the first one, which is a default choice of communication, is not.

1.4.1 Initialization

Before the encryption/decryption starts, a client must be initialized. Initialization usually occurs during the installation time or at the first start-up of the application. In the Telegram application, initialization is performed after the user registers her phone number [17].

During this process, the client creates a random *nonce* (128-bit) which serves as a request to the server. The server then responds with a random *server nonce* (128-bit) and the fingerprint of a RSA public key and a number $n = pq$ (64-bit), where p and q are odd primes. The factorization of the number n must be performed on the client side [17].

During installation, a list of RSA public keys was stored in the Telegram application as well. The client finds the correct RSA public key which matches the fingerprint provided by the server and uses it to encrypt a *payload* which contains:

- A new random *nonce* N (256-bit) and all previously used *nonces*
- Number n and its factors p and q

After this, the client and the server both perform the Diffie-Hellman key agreement (DH). Parameters of the DH exchange are not fixed. Thus, they are sent from the server encrypted by AES-256 in IGE mode[3] where the AES key and the AES initial vector are derived from the *nonce* N and the original *server nonce*. Thus, the client has all the information needed and can perform the DH calculation for establishing the shared secret.

1.4.2 Client-Server Encryption

The client-server encryption (so called *cloud chats* or *regular chats*) is a solution for encrypting the communication between the Telegram application and the server. It uses the shared secret which was established in section 1.4.1. This key is called an `auth_key` [19].

[3]Description of the IGE block cipher mode (Infinite Garble Extension) is outside of the scope of this thesis. An interested reader can find the details in the documentation [18].

However, the `auth_key` is not the only key which is used during the encryption. With every message, a `msg_key` is computed as a SHA-1 hash of the message. These two keys are then used as inputs to the Key derivation function (KDF) which generates the AES key and the AES initialization vector.

In order for the server to decrypt the message, `msg_key` is appended to the encrypted message along with the `auth_key` fingerprint (64-bit hash of this key) [19].

It is important to say that MTProto establishes a new `auth_key` after 100 messages or after one week since the key establishment. After one of these thresholds occurs, a new `auth_key` is established and the old one is destroyed [17]. This mechanism increases the forward secrecy.

1.4.3 Client-Client Encryption

Client-client encryption (so called *secret chats*) is used in end-to-end communication between the two parties. In this case, server is an intermediary point for establishing a master secret between Alice's and Bob's devices. This step is important, because the *secret chats* are device exclusive and they cannot be shared between multiple devices [16].

The master secret is established as an additional DH calculation to the initialization part (see section 1.4.1). This means that both parties exchange their public Diffie-Hellman values through the server and use them to establish the master secret.

In addition, a DH private key s is calculated as a XOR of two values:

$$s = r_{client} \oplus r_{server} \tag{1.1}$$

where r_{client} is a random integer generated by the client, r_{server} is a random integer generated by the server and operator $\oplus$ denotes the XOR operation. This mechanism mitigates weak random number generators on some mobile phones [17].

1.5 Signal Protocol

Previously known as the TextSecure Protocol or Axolotl, the Signal Protocol is one of the most secure IM communication protocols in the world [20]. Edward Snowden recommended the Signal (application/protocol) on many occasions. His words *"Use anything by Open Whisper Systems"* are currently present on the *signal.org* homepage and many other sources present Snowden as a *fan* of the application, for example The Verge [21] or New York Times [22].

The Signal Protocol is a non-federated and completely open-source protocol which uses an end-to-end encryption for every message, voice call, video call, attachments, etc. It is based on commonly used security standards (e.g. AES and Diffie-Hellman over elliptic curves) which the protocol extends with several brand new cryptographic approaches.

Moxie Marlinspike and Trevor Perrin started to develop the Signal Protocol in 2013. The first version was based on the OTR (see section 1.2 for details) and it was called TextSecure v1. Later on, TextSecure v2 was released which also included the combination of the OTR ratchet and the SCIMP[4] ratchet (synchronous KDF forward ratcheting) [23]. This version of the ratchet was later migrated to the Axolotl Ratchet and the protocol as a whole was called Axolotl[5].

In the meantime, TextSecure v3 was published and a research team from the Ruhr-University Bochum provided a security analysis of this protocol version [24]. They presented an *unknown key-share attack* and they also proposed a correction of the vulnerability. This vulnerability was later fixed and because researchers haven't found any other major issues, the protocol was pronounced secure.

Later on (mostly for clarification purposes), the Axolotl Ratchet was renamed to the Double Ratchet and the protocol as a whole (Axolotl) was renamed to the Signal Protocol [25].

1.5.1 Acceptance in Other Applications

For many years of development, the protocol became more and more cryptographically robust. Due to its open-source implementation and strong cryptographic approaches, many developers of IM clients started to think of including this protocol into their applications. And many of them did.

The Signal Protocol is currently used in several IM applications such as WhatsApp [26], Google Allo [27], Facebook Messenger [28], Skype [29], Wire [30], and many more. Unfortunately, some of these applications support this protocol only in *private communication* (e.g. Messenger with the "secret conversations", Google Allo with the "incognito mode" and Skype with the "private conversations"), which still makes secured communication somewhat optional.

1.5.2 Cryptography

Even though a security analysis of the Signal Protocol is presented in chapters 2 and 3, we introduce a brief overview of the protocol's cryptography approaches here as well.

The Signal Protocol uses standard cryptography algorithms such as AES-256, Diffie-Hellman calculations over elliptic curve Curve25519 or HMAC-SHA256. These algorithms are well known and they are constantly reviewed by researchers and specialists all over the world.

[4]Silent Circle Instant Messaging Protocol

[5]The name Axolotl referred to a salamander with remarkable self-healing capabilities – same as the protocol could "heal itself" by disabling the attacker from accessing the plaintext of later messages if she compromised the private keys at some point in time

Furthermore, these algorithms are used in the brand new cryptographic mechanisms to ensure (almost) absolute forward secrecy, resilience and break-in recovery, along with digital signatures, authentication, and more.

The essential cryptography approach which is used in the Signal Protocol is called Double Ratchet. It ensures the derivation of the symmetric cryptographic keys for the message encryption in an asynchronous environment. These keys are derived from the shared secret between the two parties which was established using the Extended Triple Diffie-Hellman (X3DH) key agreement protocol. Digital signatures are based on the XEdDSA and VXEdDSA signature schemes.

Chapter 2

Protocol Analysis

The authors of the Signal Protocol from Open Whisper Systems (OWS) decided to create a brand new cryptography protocol which would provide the best security for instant messaging communication over the internet. Their goal was to make the security available for everyone, even to those without an understanding of cryptography [31].

In order to achieve this goal, they had invented several new ideas on how to make communication over the internet more secure with a minimum risk of an intrusion. These new ideas are built on well-known standards and cryptographic methods. This approach ensures community-based feedback on the overall security as well.

2.1 Protocol Overview

The Signal Protocol is used for IM communication of two or more parties in a conversation. All data transmitted between the parties is encrypted using end-to-end encryption. The protocol supports text conversations and voice conversations as well, although the voice calls are restricted only to one-to-one communication.

Signal Protocol is an open-source project. The OWS team presents three libraries of the Signal Protocol implemented in C, Java and JavaScript. A deeper description of the Signal libraries can be found in chapter 3.

The OWS team also presents four technical documents which specify a general functionality and recommendations regarding the Signal Protocol [32]. They represent separate ideas which can be added independently to any other protocol or a security system, if needed.

The first document describes functionality and recommendations for a correct usage of the XEdDSA and the VXEdDSA signature schemes, ensuring valid digital signatures. The second document describes the Extended Triple Diffie-Hellman (X3DH) key agreement protocol for establishing a shared secret key between the two parties.

The Double Ratchet algorithm is specified in the third document. It describes an exchange of the encrypted messages based on the previously accepted shared key. Both parties derive new keys for every message. Thus, earlier keys cannot be calculated (with the present computational power) from the later ones. In addition, every Double Ratchet message also carries the Diffie-Hellman public values which are mixed into the newly derived keys. This also ensures that the later keys cannot be calculated from the earlier ones. This procedure brings an additional protection in case of a compromise of the party's keys.

The Signal Protocol is used in many applications on many devices. For example, Alice may wish to use the Signal application on her mobile phone and on a desktop computer as well. This is provided by the session management of multiple clients or devices communicating in an asynchronous communication, called the Sesame algorithm. It is described in the fourth document. The session management and the Sesame algorithm are outside of the scope of this thesis. An interested reader can find more information in the documentation [33].

2.2 Mathematical Notations

To better relay an understanding of the mathematical expressions used in this thesis, we define a set of notations which are respected across the whole thesis. If there is an exception in some expression, it is noted explicitly.

2.2.1 Basic Operations

We assume standard notation for operators. The addition and subtraction of two arbitrary elements is given by $A + B$ and $A - B$. The multiplication of two arbitrary elements is given by either $A * B$ or AB.

Elements in a form of numbers (i.e. integers or scalars) are denoted as lowercase letters. Integer a modulo integer b is denoted as $a \pmod{b}$. The division of two integers $a/b \pmod{p}$ where p is a prime number is calculated as $ab^{-1} \pmod{p}$. Number b^{-1} is called the multiplicative inversion of the number b and can be calculated using the Extended Euclidean Algorithm in a polynomial time.

2.2.2 Elliptic Curve Parameters

Elliptic curves are used widely in the Signal Protocol. Hence we assume the same notation from [34] to preserve consistency with these documents. Parameters of an elliptic curve can be found in table 2.1.

Table 2.1: Elliptic curve parameters notation

Name	**Definition**
B	Base point
I	Identity point
p	Field prime
q	Order of the base point
c	Cofactor
d	Twisted Edwards curve constant
A	Montgomery curve constant
n	Nonsquare integer modulo p
$\|x\|$	$\lceil \log_2(x) \rceil$
b	$8 * (\lceil (\|p\| + 1)/8 \rceil)$ (= Bit-length for encoded point or integer)

The product of any scalar a with the point P is denoted as aP. The sum of two points P and Q is denoted as $P + Q$. The coordinates of the point P are given in brackets, i.e. (x, y).

2.3 XEdDSA and VXEdDSA Signature Schemes

In an asymmetric cryptography, we use a private key to digitally sign a message and a public key to verify the sender. The public key is advertised publicly and the private key never leaves the user's device. Digital signing can be achieved by using DSA ("Digital Signature Algorithm") and its other mathematical schemes, e.g. using the elliptic curves.

In the Signal Protocol, two specific elliptic curves are used for cryptographic calculations – Curve25519 and Curve448 [34]. The parametres of both of these curves are briefly specified in section 2.3.2. They are used for both Elliptic Curve DSA (ECDSA) and Elliptic Curve Diffie-Hellman (ECDH) calculations. The Curve448 is more secure than Curve25519 but sacrifices some performance.

For their proper usage, an Edwards curve representation of these curves is specified (see section 2.3.1 below), called Ed25519 and Ed448 respectively [35]. The design of these elliptic curves provides faster mathematical computations without sacrificing the security [36].

ECDSA with the usage of either of these curves is called the EdDSA. Names X25519 and X448 denote functions used in the ECDH calculation with Curve25519 and Curve448 respectively.

An abbreviation XEdDSA denotes a signature scheme which enables the use of a single key pair format for both ECDH and ECDSA [34], used with either the Ed25519 or the Ed448. It is also possible use the XEd25519 or XEd448 names, if a closer specification of the used elliptic curve is needed.

VXEdDSA is an abbreviation for the "Verifiable XEdDSA" which is an extension to the XEdDSA scheme and provides a *verifiable random function*

(VRF [37]). If a signature was successfully verified, this function returns a unique value for the message and the public key and it is indistinguishable from a random value.

Both XEdDSA and VXEdDSA require a cryptographic hash function. The standard SHA-512 is the default choice in the Signal Protocol.

2.3.1 Curves Overview

The **Edwards curve** is an alternate form of elliptic curves. In comparison to the Weierstrass form[6] the Edwards curves have better general performance [36]. The following equation is called the Edwards curve:

$$x^2 + y^2 = 1 + dx^2y^2 \tag{2.1}$$

where K is a (finite) field, $d \in K, d \notin \{0,1\}$ and $x, y \in K$. In addition to (2.1), twisted Edwards curve is given in [38] as:

$$ax^2 + y^2 = 1 + dx^2y^2 \tag{2.2}$$

where $a, d \notin \{0,1\}$. The Edwards curve is a twisted Edwards curve with $a = 1$.

When we take a neutral element[7] of the twisted Edwards curve as the point $(0,1)$, the sum of the points (x_1, y_1) and (x_2, y_2) is given by formula (2.3):

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right) \tag{2.3}$$

The equation (2.3) is called the *Edwards addition law* [39, 40, 38] which gives an explicit formula for computing the addition of **any** two points[8]. The inverse of an arbitrary point (x_1, y_1) is $(-x_1, y_1)$. Other forms (like Weierstrass) need more equations for this calculation. As stated in [39], this provides a significant computational advantage because while calculating, there are no exceptions for doubling, no exceptions for the neutral element, no exceptions for negatives, etc. This results in a good protection against side-channel attacks, because doubling a point takes no extra cost.

Another form of the curves used in the Signal Protocol is the **Montgomery curve** which is defined by equation (2.4):

$$By^2 = x^3 + Ax^2 + x \tag{2.4}$$

where K is a (finite) field, $A, B \in K$ and $B(A^2 - 4) \neq 0$ [42].

[6]Weierstrass equation: $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$

[7]Neutral element $\mathcal{O}$ of the elliptic curve E is a point where $P + \mathcal{O} = \mathcal{O} + P = P$ for every point $P \in E$.

[8]If the neutral element (i.e. identity point) is $(0,1)$ and d is not square in K. See [41] for details.

Edwards form is birationally equivalent[9] to an elliptic curve in the Montgomery form [42, 35].

Furthermore, the Montgomery form allows to use only a u-coordinate from the point (u, v) for the use of a Montgomery ladder for calculations. These calculations are frequently needed in the ECDH. This results in smaller public keys without the expense of a point decompression [34].

However, the EdDSA signatures are defined on twisted Edwards curves. Thus, we need to be able to convert both forms to each other. To achieve this, a slightly different representation of the twisted Edwards point is created. Instead of using a point with two coordinates $P = (x, y)$, we only use the y-coordinate and a *sign bit* s [43]. The x-coordinate can be computed later using the equation (2.5):

$$x = \pm\sqrt{(y^2 - 1)/(dy^2 + 1)} \tag{2.5}$$

where $\pm$ depends on the sign bit $s \in \{0, 1\}$. The sign bit is 1 if and only if x is negative.

For converting the Montgomery u-coordinate of the point (u, v) to a twisted Edwards point P containing the y-coordinate and the sign bit s, we can apply the *birational map* to compute the y-coordinate. The sign bit s is implicitly chosen as zero [34]. How to construct this birational map is outside of the scope of this thesis. An interested reader can find more information in the documentation [35].

This conversion can be represented by the pseudo-code[10] below:

```
convert_mont(u):
    u_masked = u (mod $2^{|p|}$)
    P.y = u_to_y(u_masked)
    P.s = 0
    return P
```

Firstly, the u-coordinate has to be masked by specification of the particular elliptic curve which is also described in [35]. Function `u_to_y` applies the birational map. Then, the sign bit is set for the twisted Edwards point P.

We define the twisted Edwards private key as a scalar a. The public key A is then computed as $A = aB$ where B is the base point of the twisted Edwards elliptic curve. Morover, in accord with the previous paragraph, this public key has a zero sign bit [34].

The Montgomery private key is a scalar k. We can convert this private key to the twisted Edwards public key and private key (i.e. both A and a).

[9]Curves E_1 and E_2 are birationally equivalent when there is a map $\phi : E_1 \to E_2$ defined at every point of E_1 *with a small set of exceptions* and an inverse map $\phi^{-1} : E_2 \to E_1$ defined at every point of E_2 *with a small set of exceptions.* In other words, we can say that the curves are *almost the same.*

[10]`P.y` denotes the y-coordinate of the point P. `P.s` is the sign bit of the point P.

To achieve this, we need to multiply the twisted Edwards curve base point B with the Montgomery private key k. The y-coordinate of the result is the y-coordinate of the public key A. In accordance with the previous paragraphs, we set the the sign bit to zero once again.

The problem is that the multiplication of the Montgomery private key k with the Edwards curve base point B as described above does not always put the sign bit to zero (we force it to be zero every time), making the private key invalid in these cases. Thus, the private key value has to be adjusted to reflect the sign bit [44] (see the pseudo-code below).

The pseudo-code [34] of the conversion (called `calculate_key_pair`) of the Montgomery private key k to the twisted Edwards public and the private keys with the notation preserved form the paragraphs above can look like this:

```
calculate_key_pair(k):
    E = kB
    A.y = E.y
    A.s = 0
    if E.s == 1:
        a = -k (mod q)
    else:
        a = k (mod q)
    return A, a
```

It is important to mention that this function works with private keys directly. Due to conditional branching, it is crucial to implement it in a constant time so it will be resistant against side-channel attacks.

2.3.2 Curve25519 and Curve448

As previously mentioned, both Curve25519 and Curve448 can be used in the Signal Protocol as they provide a very good cryptographic security and a great performance as well. According to the SafeCurves [45], both curves are considered *Safe* in every tested aspect. Both curves are also designed so that fast, constant-time implementations are easier to produce. Thus, they are resistant to a wide range of side-channel attacks, including timing and cache attacks [35].

The definition of the Curve25519 can be found in table 2.2 below. Mathematical notations regarding the curves can be found in section 2.2.2.

Function `convert_mont` is defined in section 2.3.1 which takes the Montgomery u-coordinate ($= 9$) and converts it to the twisted Edwards point, i.e. the base point in this case.

The Curve448 (sometimes called the Ed448-Goldilocks) is a more secure curve than the Curve25519 [46]. Its security is estimated at around 224 bits, rather than 128 bits of Curve25519.

Table 2.2: Curve25519 parameters [34]

Name	**Definition**
Equation	$-x^2 + y^2 = 1 + dx^2y^2$
B	`convert_mont(9)`
I	$(x = 0,\ y = 1)$
p	$2^{255} - 19$
q	$2^{252} + 27742317777372353535851937790883648493$
c	8
d	$-121665/121666 \pmod p$
A	486662
n	2
$\|p\|$	255
$\|q\|$	253
b	256

A definition of the Curve448 can be found in the table 2.3. The backslash in the definition of the order q denotes a continuity of the number on the next line so it fits the page – it is too large.

Table 2.3: Curve448 parameters [34]

Name	**Definition**
Equation	$x^2 + y^2 = 1 + dx^2y^2$
B	`convert_mont(5)`
I	$(x = 0,\ y = 1)$
p	$2^{448} - 2^{224} - 1$
q	$2^{446} - 13818066809895115352007386748515\backslash$ $42688033669247488217860989454750388$5
c	4
d	$39082/39081 \pmod p$
A	156326
n	-1
$\|p\|$	448
$\|q\|$	446
b	456

As mentioned, the Curve448 is more secure than the Curve25519. This is the consequence of sacrificing some performance over security. However, it is expected that large quantum computers will be able to crack both Curve25519 and Curve448. It is also expected that classic computers will never be able to crack neither of these curves, so usage of the Curve25519 is recommended [35].

2.3.3 Hash Functions

Both XEdDSA and VXEdDSA require a cryptographic hash function. The standard SHA-512 is the default choice in the Signal Protocol [34]. This cryptographic hash takes a byte sequence as an input and returns an integer (in the hexadecimal form) as the output [34].

For the cryptographic domain separation, hashes are indexed by a non-negative integer i such that $2^{|p|} - 1 - i > p$, where p is the field prime of the elliptic curve. The indexed hash is then computed by:

```
hash_i(X):
    return hash(2^b - 1 - i || X)
```

where X is an input byte sequence, the $||$ symbol is a concatenation of two byte sequences and b is a number of bits needed for encoding a point or an integer:

$$b = 8 * (\lceil (|p| + 1)/8 \rceil) \tag{2.6}$$

This procedure of the cryptographic domain separation provides a great diversification of hash outputs to the separate domain [34].

Let's say we need to hash the bytes sequence `ThisIsMyPrivateKey` which is our private key. If we hash it using e.g. SHA-256, we get this output:

```
04dd970aa9189e3100a0efb72547e1fadf6f6ce45b6be8f478ba4e9524a9426d
```

Now, we would like to use our private key for two different security functions. Since we trust our selected SHA-256, we do not want to use any other hash function. How to pass a different hash with the same private key to both security functions? The cryptographic domain separation above solves this problem, because it diversifies the output for the same input (i.e. the private key).

If we choose the indexed hash $\texttt{hash}_0$ with the $b = 8$, we actually hash a private key *FFThisIsMyPrivateKey*. If we hash using the $\texttt{hash}_1$, we actually hash a private key *FEThisIsMyPrivateKey*[11], etc. Then the actual result is:

```
hash_0 FFThisIsMyPrivateKey 318d750b9c82...69fbe5d4e650
hash_1 FEThisIsMyPrivateKey 9143f5521b4f...8df421bc711b
```

which gives the different hash outputs for a single private key and a single hash function.

2.3.4 Signing and Verification with XEdDSA

Signing and verification is performed similarly to the standard ECDSA. In XEdDSA signing, Alice has to have these values:

- Montgomery private key k (integer mod q)

[11]If we use a little-endian.

- Message M to sign (byte sequence)
- Secure random data Z (64 bytes)

and uses this algorithm [34] written in pseudo-code:

```
xeddsa_sign(k, M, Z):
    A, a = calculate_key_pair(k)
    r = hash$_1$(a || M || Z) (mod q)
    R = rB
    h = hash(R || A || M) (mod q)
    s = r + ha (mod q)
    return R || s
```

Function `calculate_key_pair` is used for converting the Montgomery private key to the twisted Edwards public key A and the private key a (see section 2.3.1 for details). Function `hash`$_1$ is an indexed hash function defined in section 2.3.3 with $i = 1$. The $\|$ symbol is a concatenation of two byte sequences.

The r value is a nonce calculated from the private key, the message and the random sequence. It is critical to create a new random sequence for every new signature. If the *nonce* r is used twice due to signing the same message repeatedly while not generating a unique Z, we can solve a simple system of equations in a constant time:

$$s_1 = r + h_1 a \pmod q \tag{2.7a}$$

$$s_2 = r + h_2 a \pmod q \tag{2.7b}$$

The private key can be calculated as $a = (s_1 - s_2)/(h_1 - h_2) \pmod q$. However, if done properly, adding this random sequence Z improves the security resilience in comparison to the standard deterministic signing schemes, i.e. in comparison when only the private key a and the message M is used while hashing.

In XEdDSA verification, Bob has to have these values:

- Montgomery public key u
- Message M to verify (byte sequence)
- Digital signature ($R \| s$) to verify (concatenated byte sequence of $2b$ bits)

and uses the algorithm [34] written in pseudo-code below.

Function `convert_mont` is a function defined in section 2.3.1 which takes the Montgomery u-coordinate and converts it to the twisted Edwards point A. The function `on_curve` checks if the converted point lies on the curve. Values `true` or `false` are returned depending on the validity of the signature.

```
xeddsa_verify(u, M, (R || s)):
    if u >= p or R.y >= $2^{|p|}$ or s >= $2^{|q|}$:
        return false
    A = convert_mont(u)
    if not on_curve(A):
        return false
    h = hash(R || A || M) (mod q)
    R_check = sB - hA
    if bytes_equal(R, R_check):
        return true
    return false
```

2.3.5 VXEdDSA

As stated in [37], the VRF is a (pseudo)random generator which will allow other parties to verify a random value without compromising its unpredictability. In other words, a *seed owner* can generate a value which can then be verified by another party, i.e. that the value is truly generated from the (unknown) seed.

The VXEdDSA signing algorithm takes the same inputs as XEdDSA [34]. The difference is that the digital signature consists of three values (V, h, s), where V is a twisted Edwards point, h and s are integers modulo q. In addition, value v is returned with the signature which is the output of the VRF.

The verification of VXEdDSA is the same as XEdDSA. If the signature is valid, v value is returned and it equals the value that was created during the signing process.

2.4 X3DH Key Agreement Protocol

The X3DH (”Extended Triple Diffie-Hellman”) is an asynchronous key agreement protocol which establishes a shared secret key between the two parties who mutually authenticate each other based on the public keys [47]. This means that one party can be offline while the other party tries to establish a shared secret key. To achieve this, a server is used for storing the information between the two parties. Thus, there are 3 parties in X3DH protocol:

- **Alice** sends an encrypted initial message to Bob so she can establish a shared secret which may be used later for further communication.
- **Bob** wants to establish a shared secret with Alice as well by receiving the initial data from her. He can use the shared secret later for further communication.

- **Server** stores the initial message from Alice to Bob, so it can be later received by Bob asynchronously. Because of the server, Bob can be offline when Alice sends the initial message. Security considerations about a *server trust* can be found separately in section 2.4.6.

The X3DH can use both the Curve25519 and the Curve448 curves which were described in section 2.3.2. When we talk about these curves in the context of the X3DH, we call them X25519 and X448 respectively.

The hash functions which are used in the X3DH are 256 or 512 bit functions. The strongly recommended functions are SHA-256 or SHA-512.

In addition to that, the X3DH needs one additional parameter (information) – an ASCII string which identifies the application. This string is later used as the *associated data*.

2.4.1 Keys

Both parties have several keys for a specific purpose at their disposal [47]. These keys are used for the asymmetric cryptography, so every key pair is comprised of a private key and a corresponding public key:

- **Identity key pair** (IKP) – Long term keys which are tied up with a device.

- **Ephemeral key pair** (EKP) – A random key pair which is used in a single X3DH run.

- **Signed prekey pair** ($SPKP$) – Keys which are signed with a private key from the IKP. Public key from $SPKP$ is periodically uploaded on the server. "Prekey" denotes that it is the key which was published to the server **before** any communication with the other party was performed (i.e. before Alice contacted Bob or vice versa).

- **One-time prekey pair** ($OPKP$) – Public keys from $OPKP$ are a set of one-time prekeys which **can** be published to the server as well. Only a single one-time prekey pair can be used in one X3DH protocol run, and only optionally.

To summarize the keys, we use the key pairs according to the table 2.4 in a single X3DH protocol run. In this table, we consider that Alice initializes the communication with Bob.

Thus, there are five key pairs used for establishing a shared secret. From now on, we denote the public keys by capital letters (e.g. IK_A is Alice's public identity key) and private keys with the `priv` designation (e.g. $privIK_A$ is Alice's private key).

Table 2.4: X3DH keys [47]

Name	**Definition**
IKP_A	Alice's identity key pair
EKP_A	Alice's ephemeral key pair
IKP_B	Bob's identity key pair
$SPKP_B$	Bob's signed prekey pair
$OPKP_B$	Bobs's one-time prekey pair (optional)

2.4.2 Elliptic Curve Diffie-Hellman Function

The elliptic curve Diffie-Hellman Function (ECDH) used in the Signal Protocol is one of the two X25519 or X448 functions using the Curve25519 or the Curve448 respectively [35].

These functions take two parameters – a private key and a public key. The private key is a scalar k and the public key is a point P. However, for the ECDH calculation, we use a Montgomery form of the elliptic curve (see section 2.3.1 for details). Thus, we considered a u-coordinate from the (public) point P as the value of the public key. All calculations are performed over the Galois field $GF(p)$, where p is a prime number.

Let's say Alice has a private key a and Bob has a private key b and they decide to use the Curve25519 for DH calculation. Both of them know the public point P which has the u-coordinate equal to a number 9 (denoted as a *public string*). Figure 2.1 demonstrates how the shared secret can be established:

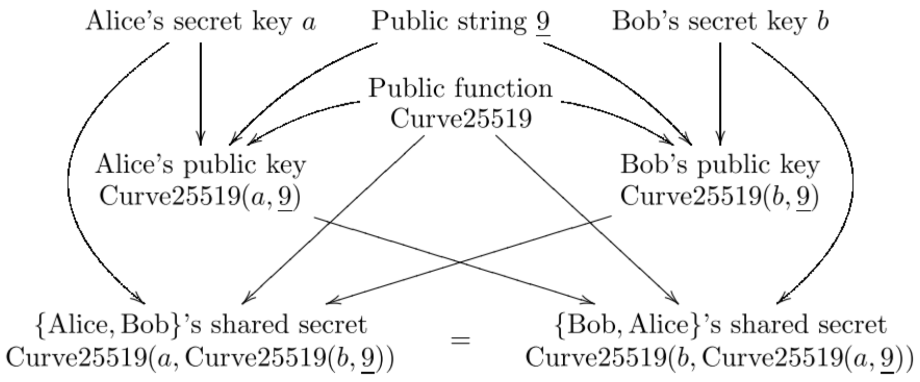
Alice's secret key a
Public string $\underline{9}$
Bob's secret key b
Public function Curve25519
Alice's public key Curve25519($a, \underline{9}$)
Bob's public key Curve25519($b, \underline{9}$)
{Alice, Bob}'s shared secret Curve25519(a, Curve25519($b, \underline{9}$))
=
{Bob, Alice}'s shared secret Curve25519(b, Curve25519($a, \underline{9}$))

Figure 2.1: The shared secret establishment using Curve25519 [48].

2.4.3 Key Derivation Function

A Key Derivation Function (KDF) is a function which derives one or more secret keys from the secret values given as an input (so called "input keying material"). One of its common usages is to *convert* the result of a Diffie-Hellman key exchange into a symmetric key which is then used in a symmetric cryptography.

A few security measures should be considered with the KDF usage:

- KDF gives the same output length for any input length.
- All bits of the input are processed to the output.
- A salt has to be used for deriving different outputs for the same input.
- KDF should be slow as a countermeasure to the brute-force attacks.
- KDF should use a lot of memory.

The Signal Protocol uses a HMAC-based Key Derivation Function (HKDF) with the "extract-then-expand" paradigm [49]. The first stage takes the input and "extracts" a fixed-length pseudorandom key K from it. The second stage "expands" the key K into the several additional pseudorandom keys as outputs of the KDF. These outputs also have a desired length.

The "extract" stage is important in those cases when Eve (an eavesdropper) may have some partial knowledge about the input. This is typically a Diffie-Hellman value computed by the key exchange protocol. If Eve does not have any information about the input, the "extract" stage can be skipped.

HKDF allows to optionally include an additional information in the KDF output. It serves a very practical purpose [49], as it allows to carry a protocol number, algorithm identifiers, etc.

In the Signal Protocol, the KDF function takes these inputs [47]:

- HKDF input keying material which is created as a concatenation of a byte sequence F and a keying material KM. The byte sequence F is used as a cryptographic domain separation which was explained in section 2.3.3.
- HKDF salt.
- HKDF additional information.

2.4.4 Authenticated Encryption with Associated Data

Authenticated Encryption with Associated Data (AEAD) is a solution to the problem of sending an *associated data* (AD) bound to the encrypted message. Even though AD is in a plaintext form, it is authenticated along with the message [50]. For example, AD can be a packet header or an additional information in the security protocol which does not have to be encrypted (or must not be encrypted) but is still covered by the authenticity protection.

For authenticated-encryption (AE), the OCB scheme ("Offset Codebook scheme") is used. This block cipher mode provides confidentiality and authenticity for the message and authenticity for the associated data. I.e. the OCB scheme *is* the AEAD scheme [51]. However, AD can be zero-length, so the OCB can be used to only authenticate-encrypt the message, too.

Generally, there are two methods how to create the AEAD. The first method is called a *nonce stealing*, but it is limited by the length of the AD. The second method is called a *ciphertext translation* which is less restrictive. How to implement these schemes is outside of the scope of this thesis. An interested reader can find more information in the documentation [50].

2.4.5 The X3DH Protocol

The X3DH protocol has three phases:

1. Bob publishes his IK_B and prekeys to a server (see the table 2.4) along with the SPK_B signed e.g. using the XEdDSA
2. Alice fetches the prekeys from the server and uses them to send an initial message to Bob
3. Bob receives the initial message and uses it to calculate a shared secret

Bob's IK_B is uploaded to the server only once. However, Bob can re-upload the other keys at any time. A typical situation is when the server is out of Bob's OPK_B keys or Bob wants to *refresh* his SPK_B for a better security (e.g. once a month). If Bob changes his signed key, he may keep his corresponding $privSPK_B$ for some period of time, because some messages can be delayed in a transit. Moreover, Bob must delete the old keys to achieve a forward secrecy[12]. The $privOPK_B$ has to be deleted immediately after a message with the corresponding public key is received.

After Alice contacts the server and fetches the keys from Bob, she verifies the the prekey signature and she aborts the communication immediately if the verification fails. If the verification is successful, Alice performs three Diffie-Hellman calculations[13]. How a single ECDH calculation is performed can be found in section 2.4.2:

$$\mathtt{DH_1 = DH(privIK_A,\ SPK_B)}$$
$$\mathtt{DH_2 = DH(privEK_A,\ IK_B)}$$
$$\mathtt{DH_3 = DH(privEK_A,\ SPK_B)}$$

The `DH` function is either the X25519 or the X448 elliptic Diffie-Hellman function with implementation details described in [35].

[12]If Bob's device is compromised by Trudy (an intruder), she cannot read the older messages – keys do not exist anymore.

[13]The X3DH protocol is named after this step.

Additionally, if the (optional) OKB_B key was fetched from the server, the fourth ECDH calculation is performed as well:

$\mathtt{DH_4 = DH(EK_A,\ OPK_B)}$

If the OKB_B key is not present on the server, it is not fetched by Alice and this step is not performed. Note that DH_1 and DH_2 are used for authentication and DH_3 and DH_4 are used to provide the forward secrecy [47]. Thus, the OKB_B is used as an additional security measure.

The shared secret SK can be then calculated as follows:

$\mathtt{SK = KDF(DH_1 \parallel DH_2 \parallel DH_3\ [\parallel DH_4])}$

Where $\parallel$ is a concatenation of the results and KDF is a Key Derivation Function (see section 2.4.3). After the shared secret SK is computed, Alice deletes all the DH outputs and the ephemeral private key.

As the last step, Alice creates associated data AD that contains identity information about both parties. Another information can be added as well, such as usernames, certificates, etc.

$\mathtt{AD = IK_A \parallel IK_B \parallel ...}$

Now Alice has all the information needed to create the initial message containing:

- Alice's identity public key IK_A
- Alice's ephemeral public key EK_A
- Information about which of Bob's prekeys were used for calculating the SK
- An initial ciphertext encrypted with the AEAD encryption scheme (see section 2.4.4 for details) using AD as the associated data. SK can be used as an encryption key

Initial ciphertext is usually the very first *real* message which Alice sends to Bob and it carries the initial message with all the needed information for establishing the shared secret, too [47].

Upon receiving the initial message from Alice, Bob computes the SK using all the private keys corresponding to the public keys which Alice used:

$\mathtt{DH_1 = DH(privSPK_B,\ IK_A)}$
$\mathtt{DH_2 = DH(privIK_B,\ EK_A)}$
$\mathtt{DH_3 = DH(privSPK_B,\ EK_A)}$
$\mathtt{[DH_4 = DH(privOPK_B,\ EK_A)]}$
$\mathtt{SK = KDF(DH_1 \parallel DH_2 \parallel DH_3\ [\parallel DH_4])}$

After this, Bob tries to decrypt the initial ciphertext using SK and AD. If Bob fails to decrypt the initial ciphertext, he immediately aborts the communication and deletes the SK.

If Bob is successful with the decryption, Bob deletes any $privOPK_B$ he used to ensure the forward secrecy and X3DH is complete [47].

2.4.6 Security Considerations

The role of the server can be *decentralized* – the stored information can be located on many different servers. This achieves a better security against compromising the server, but for simplicity, we describe the server side as a one instance.

It is advised to use a separate authentication channel for verification of the public keys IK_A and IK_B [47]. This can be done by a visual comparison of the two public key fingerprints, or by scanning a QR code – both provided by the Signal application. If this comparison is not performed, the two parties have no cryptographic guarantee as to who they are communicating with.

As mentioned in section 2.4.5, the OPK_B is used to provide a better forward secrecy. Another reason to distribute the OPK_B to the server is to protect Bob against a replay attack [47]. Assume that Mallory (a malicious man-in-the-middle attacker) can capture the encrypted messages between Alice and Bob. Without the OPK_B key (which is used only once per X3DH run), she would be able to send the same message to Bob repeatedly. If the OPK_B cannot be used, other solutions could mitigate this problem. For example, Bob could change his SPK_B more rapidly or some post-X3DH protocol (see section 2.5) should negotiate a new SK for Alice based on Bob's new keys.

A malicious server could cause the communication between Alice and Bob to fail for several reasons. It could refuse to hand out OPK_B, so the forward secrecy would depend only on the SPK_B and its lifetime [47]. The same problem could also occur if Mallory would *drain* all the OPK_B keys from the server. In this case, server should set rate limits on fetching OPK keys for every communicating party.

2.5 Double Ratchet

The Double Ratchet algorithm is an essential approach of the Signal Protocol. It is used by two parties to exchange encrypted messages based on a shared secret key [52]. This shared secret key can be obtained by some key agreement protocol, e.g. the X3DH protocol described in the previous section 2.4.

Every party derives new keys for every Double Ratchet message from the shared secret key. Also, they send a Diffie-Hellman public values attached to the message. This Diffie-Hellman calculation is mixed into the derived keys [52]. Thus, the Double Ratchet provides a forward security and protection against deciphering old messages as well.

2.5.1 KDF Chains

The Key Derivation Function (KDF, see section 2.4.3) used in the Double Ratchet algorithm takes an *input* and the *KDF key* as the input keying material. If the KDF key is unknown to the attacker, the KDF output should be indistinguishable from a random string. If the KDF key is not secret and random, the KDF should still provide a secure cryptographic hash of its KDF key and the input data [52].

The KDF chain is created when the KDF output is used as the KDF key to another KDF:

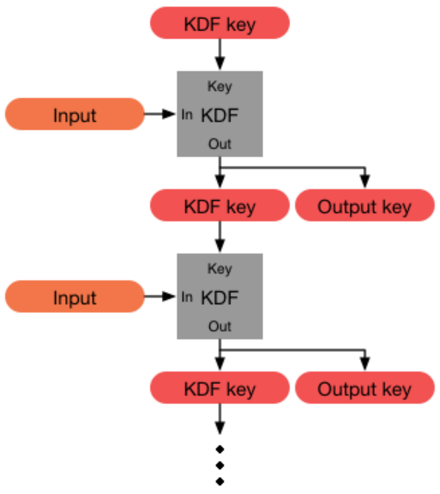

Figure 2.2: The KDF chain containing two ratchet steps [52].

The KDF chain has following properties:

- **Resilience** – If the KDF key is secret, the KDF output appears random.
- **Forward security** – If the KDF key is revealed at some point in time, the previous KDF outputs appear random.
- **Break-in recovery** – If the KDF key is revealed at some point in time, the future KDF keys appear random, provided the future KDF inputs have added a sufficient entropy.

Each party has three chains [52]:

- **Root chain**
- **Sending chain**
- **Receiving chain**

Alice's sending chain equals to Bob's receiving chain and vice versa. These chains are used for a symmetric cryptography – in the so called *symmetric-key ratchet* – deriving a key which is called a **message key**. This key is used for encrypting messages. The KDF keys are called the **chain keys**. Lastly, inputs for sending and receiving chains are **constants** and they do not provide the break-in recovery [52].

Following this terminology, we can modify the labels of the chain 2.2 to figure 2.3:

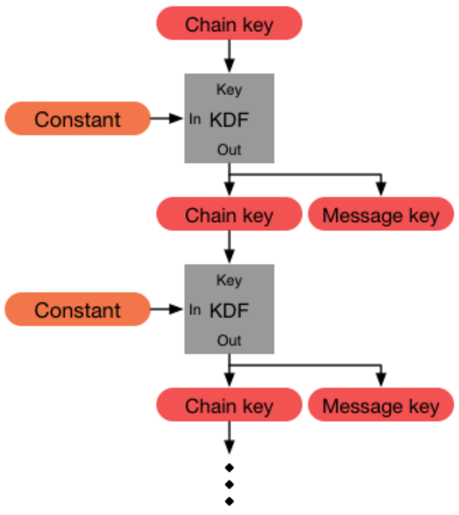

Figure 2.3: The symmetric-key ratchet – the KDF chain containing two ratchet steps. Every step derives new chain and message keys [52].

In other words, calculating the next chain key and the next message key for a given chain key is called a **ratchet step** in the symmetric-key ratchet.

2.5.2 Diffie-Hellman Ratchet

The KDF chains (section 2.5.1) do not provide the break-in recovery on their own. Thus, if an attacker could steal Alice's sending and receiving chain keys, she could read all the future incoming messages, because she can compute all the future message keys. To prevent this, additional information is mixed into the chain keys – Diffie-Hellman outputs.

Assume that Alice has a DH public key A and a DH private key a, Bob has a DH public key B and a DH private key b. Then, the Diffie-Hellman ratchet follows these steps:

1. Bob publishes his DH public key B to Alice[14].

[14]The public keys are sent in a header of every message which Alice and Bob sends.

2. Alice uses her DH private key a and Bob's DH public key B to compute a new DH output DH_A.
3. Alice publishes her DH public key A to Bob.
4. Bob uses his DH private key b and Alice's DH public key A to compute a new DH output DH_B.

Values DH_A and DH_B from the above approach are equal to each other. Thus, both Alice and Bob have the same shared secret, as in the standard Diffie-Hellman. Additionally, the following **DH ratchet** step is performed:

5. Bob generates a new DH key pair.
6. Bob computes a new DH_B from Alice's public key A and his new DH private key.
7. Bob sends his new DH public key to Alice.

These two processes can be seen in figure 2.4 below, numbered according to the steps above.

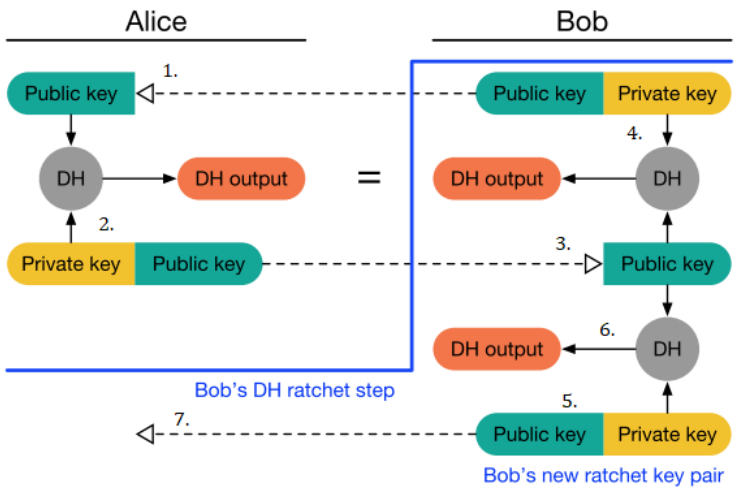

Figure 2.4: The Diffie-Hellman ratchet numbered according to the steps which both Alice and Bob have to perform [52].

After this, Alice can perform the same DH ratchet step as Bob did above.

If Trudy (an intruder) could somehow compromise Alice's (or Bob's) DH private key, it will be eventually replaced with an uncompromised one [52].

Now, if we look a little bit closer at figure 2.4, we can see that "DH outputs" are actually the chain keys described in section 2.5.1. So, to be precise, we can change labels to correspond the notation. The Diffie-Hellman ratchet with renamed labels is illustrated by figure 2.5.

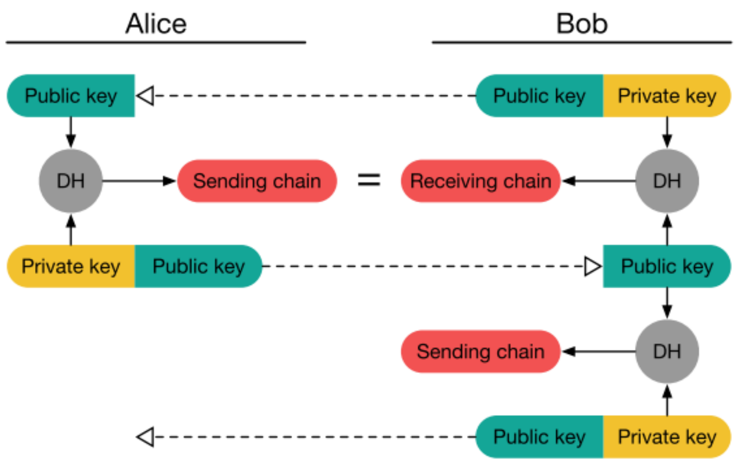

Figure 2.5: The Diffie-Hellman ratchet (figure 2.4) with renamed labels [52].

However, the figures above are a simplification. In the Diffie-Hellman ratchet, the chain keys are not used directly, but they are used as the KDF inputs to the *root chain*, and KDF outputs are used as the sending and the receiving chain keys. This improves the resilience and break-in recovery [52]. The (initial) root key is a shared secret between Alice and Bob, so it is the same for both parties. The X3DH can be used for the shared secret establishment (see section 2.4 for details). In the illustration 2.6 below, a single Diffie-Hellman ratchet step updates the root chain twice.

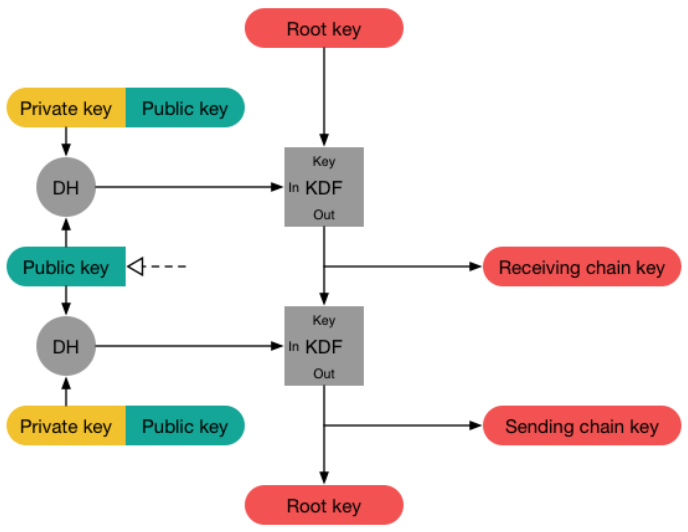

Figure 2.6: A single Diffie-Hellman ratchet step with the root chain [52].

2.5.3 The Double Ratchet

Combining the symmetric-key ratchet (section 2.5.1) and the Diffie-Hellman ratchet (section 2.5.2) gives us the Double Ratchet [52] with these rules:

- When a message is sent or received, a symmetric-key ratchet step is applied to the sending or receiving chain to derive the message key.
- When a new Diffie-Hellman ratchet public key is received from the other party, a DH ratchet step is performed prior to the symmetric-key ratchet to replace the chain keys.

Consider that Alice wants to send a message to Bob. Both parties established a shared secret beforehand, which is used as a root key for the root chain. Alice also obtained Bob's Diffie-Hellman ratchet public key.

Before Alice can send her first message, her communication needs to be initialized. The initialization is comprised of three steps:

1. Alice generates her own Diffie-Hellman ratchet key pair.
2. Alice performs a Diffie-Hellman calculation, using Bob's DH ratchet public key and her DH ratchet private key she just generated.
3. DH output is then used to calculate a new root key (RK) and a new sending chain key (CK). Now, Alice should remove her original root key for the forward secrecy.

Alice's initialization process can be seen in figure 2.7. Colors are preserved according to the previous illustrations:

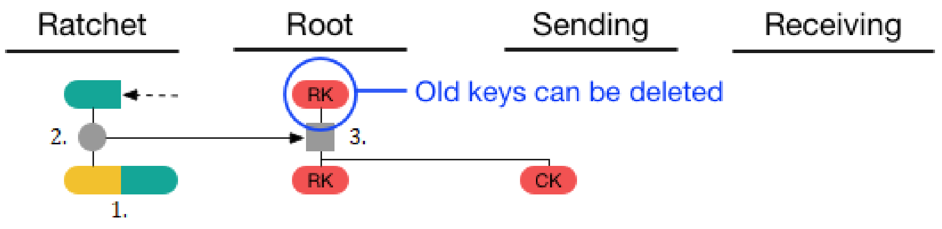

Figure 2.7: Alice's Double Ratchet initialization with colors preserved according to the previous illustrations. Alice can delete the old key RK immediately after it was used [52].

Now that Alice's communication is initialized, she can send a message $A1$ to Bob. To be able to do so, she needs to perform an additional calculation:

4. Alice performs a symmetric-key ratchet step using the CK calculated in the step 3. This will generate a new CK and a message key $A1$, which is used for encrypting the message $A1$.

Note that step 4 does not include the second parameter for the symmetric-key ratchet KDF. This input is only a constant, so for simplicity, we skip this value. Now, Alice should remove her old sending chain key for the forward secrecy as well. After the message was sent, she should also delete the $A1$ message key.

Sending a message can be illustrated by figure 2.8:

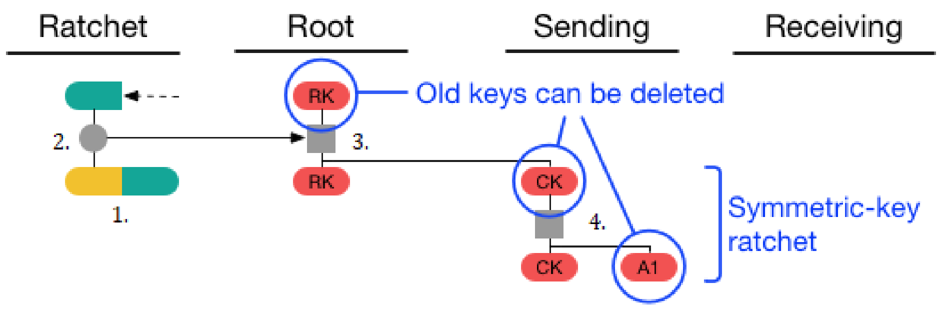

Figure 2.8: Alice sends the Double Ratchet message, deriving new keys during the process. She can delete old keys immediately after they were used [52].

When Alice receives the message $B1$ from Bob which carries a new Diffie-Hellman ratchet public key, she needs to perform these steps:

5. Alice calculates a Diffie-Hellman ratchet step with her DH ratchet private key (from the step 1.) and Bob's new DH ratchet public key $B1$. Bob performed this step himself, so this is a synchronization of both parties.

6. Output of the root chain from the step 5 is used in the receiving chain. The symmetric-key ratchet is performed over this chain key.

7. Alice generates a new Diffie-Hellman ratchet key pair.

8. Alice uses the new ratchet private key to calculate a new root key and a new sending chain key.

Note that the steps 5, 7 and 8 are performed simultaneously, creating a single Diffie-Hellman ratchet step. The separation is just for a better illustration. Receiving a message can be illustrated by figure 2.9, following the steps above.

If Alice next sends a message $A2$, receives a message $B2$ with Bob's **old** DH ratchet public key and then sends the messages $A3$ and $A4$, Alice's sending chain will ratchet three steps, and her receiving chain will ratchet once (following Alice sent three messages and received one message). Thus, Alice only performs the symmetric-key ratchets. This process is illustrated by figure 2.10.

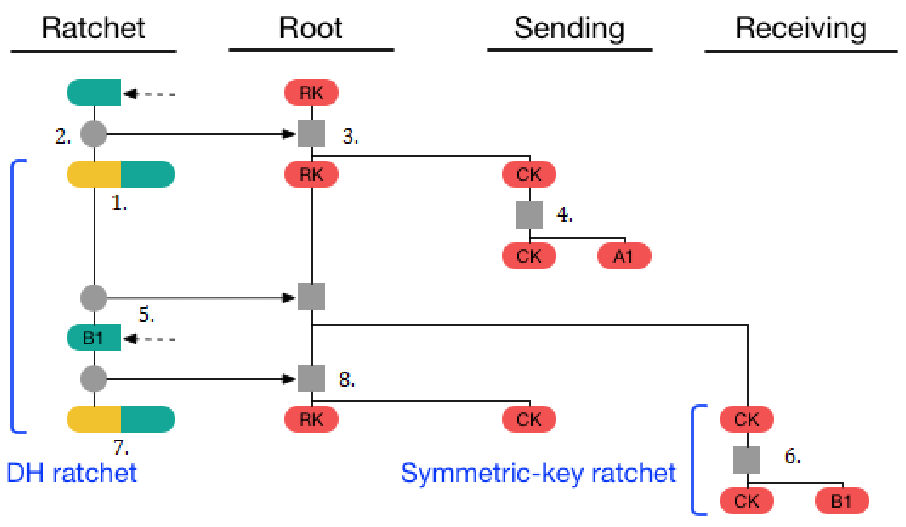

Figure 2.9: Alice receives the Double Ratchet message with a new DH ratchet public key. She performs both Diffie-Hellman and symmetric-key ratchets [52].

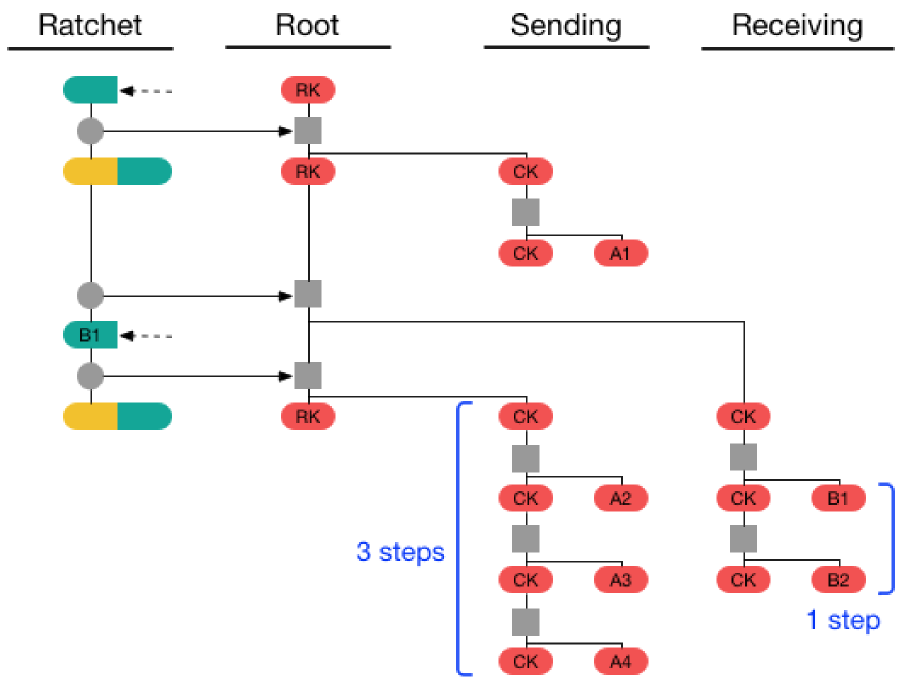

Figure 2.10: Alice sends three more messages and receives one message from Bob with the old DH ratchet public key. Alice's sending chain ratchets three steps, and her receiving chain ratchets once [52].

However, the situation is different when Bob sends a **new** DH ratchet public key with the messages $B3$ and $B4$. In this case, Alice has to perform the Diffie-Hellman ratchet step before the symmetric-key ratchet. DH ratchet step generates a new sending chain key and a new message key $A5$. Alice can use this key if she wants to send a new (fifth) message.

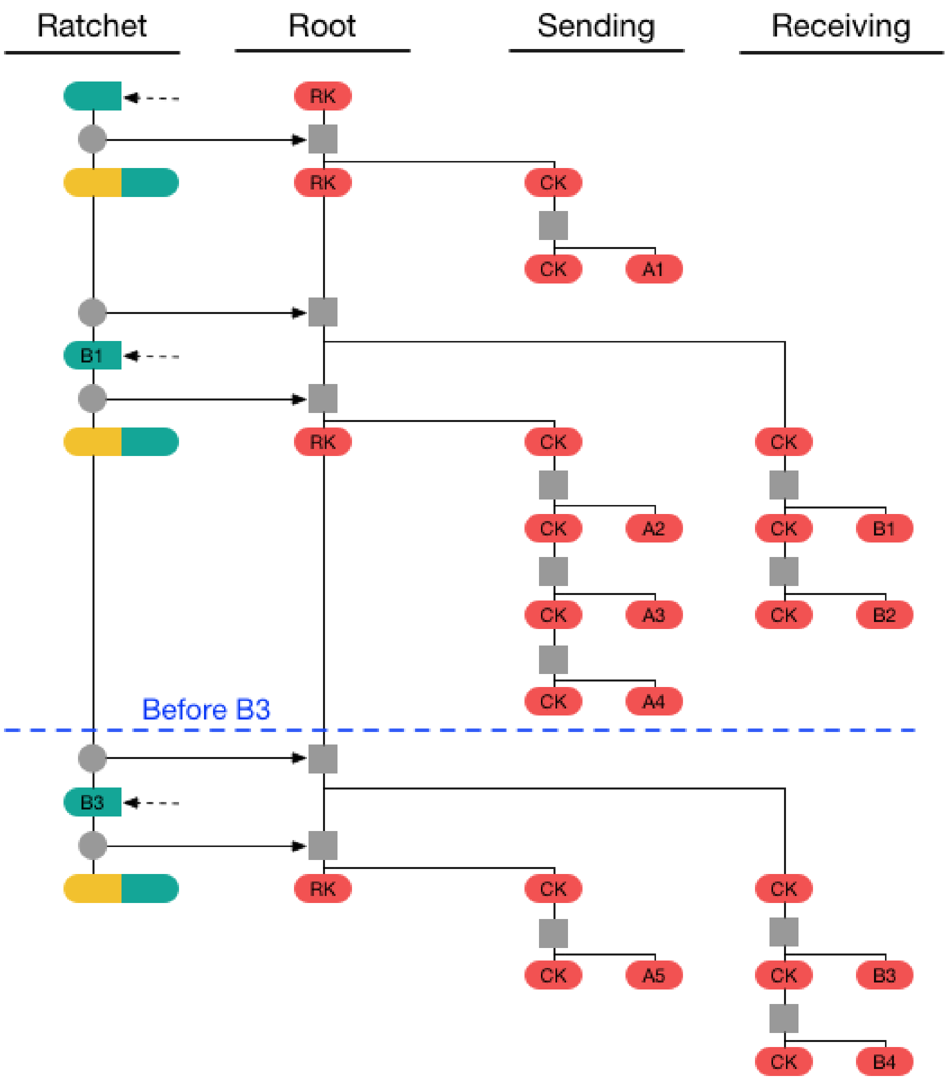

Figure 2.11: Alice performs the DH ratchet step with Bob's new DH ratchet public key before the symmetric-key ratchet [52].

Note that in figure 2.11, Alice has three sending chains and two receiving chains. New sending and receiving chain is created after every Diffie-Hellman ratchet step (except the first one).

2.5.4 Lost and Out-of-Order Messages

The Double Ratchet also handles lost or out-of-order messages [52] due to the additional information which is sent in the message header. When Alice sends a message, this additional information includes:

- Index of the message in the *current* sending chain N.
- Number of messages in the *previous* sending chain PN.

With this information, Bob can store the message keys of the skipped messages in case they arrive later by this approach[15]:

- If a new Alice's DH public ratchet key is included in the message header, number of skipped messages can be calculated as:

  $$S = PN_A - RC_B \tag{2.8}$$

  where RC_B is a number of messages in the Bob's current receiving chain. The received N_A is a number of skipped messages in Bob's new receiving chain (which he has to create due to the new Alice's DH ratchet public key).

- If Alice does not provide a new DH public ratchet key in the message header (i.e. the current one is provided), the number of skipped messages can be calculated as:

  $$S = N_A - RC_B \tag{2.9}$$

  where RC_B is a number of messages in the Bob's current receiving chain.

A constant identifying the maximum tolerated limit for skipped messages should be set. This constant should be high enough to reflect a common message loss or a traffic delay. However, it should also be low enough that a malicious sender cannot trigger excessive recipient computation [52] with the possibility of causing a denial-of-service attack.

Additionally, both parties should set a timeout for the skipped messages as well. The attacker could eavesdrop these messages (even though they did not reach the intended recipient). If she would later compromise one party's device, she would be able to read these skipped messages.

2.5.5 Header Encryption in the Double Ratchet

As described in the previous sections (2.5.3 and 2.5.4), every transmitted message has a header which contains a DH ratchet public key and integers N and PN. From these values, Eve could deduce some information about the

[15]Lower indexes determine the party from which the information comes from.

communicating parties – e.g. which message belongs to which session, or the ordering of messages [52].

To prevent this, the Double Ratchet allows to encrypt headers using two additional keys:

- Header key (*HK*)
- Next header key (*NHK*)

These keys are initialized as a shared secret between Alice and Bob (the same way the root chain key is established).

The header key is used for encrypting or decrypting the message header of the *current* sending or receiving chain. The next header key is used for encrypting or decrypting the message header from a *new* sending or receiving chain which is created because one party sent a new Diffie-Hellman public key (see section 2.5.3 for details) [52].

With this approach, figure 2.7 can be modified to contain the additional *HK* and *NHK* keys:

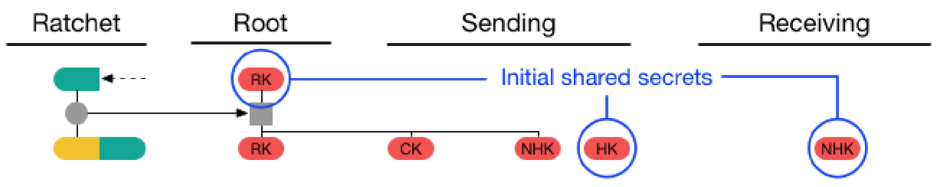

Figure 2.12: Alice's Double Ratchet initialization with the header keys. These keys are established as the shared secret between Alice and Bob [52].

Note that the sending chain *NHK* and the receiving chain *NHK* are **not** the same keys. The sending chain *NHK* is calculated from the root chain key and the DH values, while the receiving chain *NHK* is negotiated directly as a shared secret.

However, both Alice and Bob have to initialize these values following two rules:

- Alice's sending *HK* has to be equal to the Bob's receiving *NHK*, so that Alice's first message triggers a DH ratchet step for Bob
- Alice's receiving *NHK* has to be equal to the Bob's (initial) sending *NHK*, so that after Bob's first DH ratchet step, Bob's next message triggers a DH ratchet step for Alice

If Alice wants to send a new message with the encrypted header, she performs a symmetric-key ratchet (as in figure 2.8) and encrypts the header using the sending chain *HK* which is illustrated in figure 2.13.

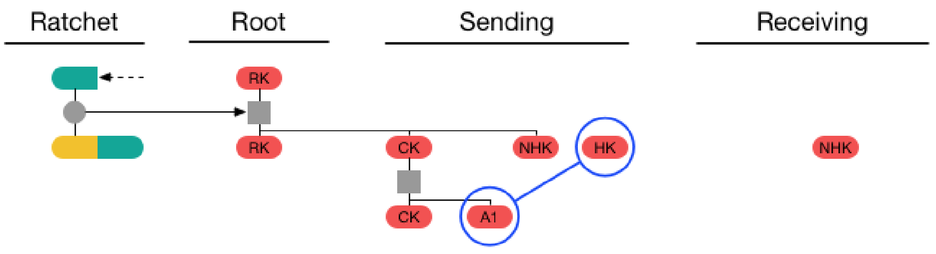

Figure 2.13: Alice sends the Double Ratchet message with the encrypted header. The *HK* key is used for the header encryption [52].

Now assume that Bob sent a message to Alice. Because the header of that message is encrypted, she does not know which header key she needs to use. Thus, she needs to try decrypting the header with every receiving header key at her disposal:

- Header key *HK* – in case Bob sent the old Diffie-Hellman ratchet public key
- New header key *NHK* – in case Bob sent a new Diffie-Hellman ratchet public key
- All stored header keys corresponding to previously skipped messages, if available

If the header was successfully decrypted using the new header key, Alice has to perform the Diffie-Hellman ratchet step. During this step, Alice has to replace the current header keys with the next header keys and the next header keys with the new header keys which were created during the DH ratchet step. This approach can be seen in figure 2.14.

If the header was successfully decrypted using a receiving header key, Alice only performs the symmetric-ratchet and decrypts with the current header key. If Bob did not send a new DH ratchet public key, Alice only performs the symmetric-ratchet with the current sending header key as well. This approach is illustrated by figure 2.15.

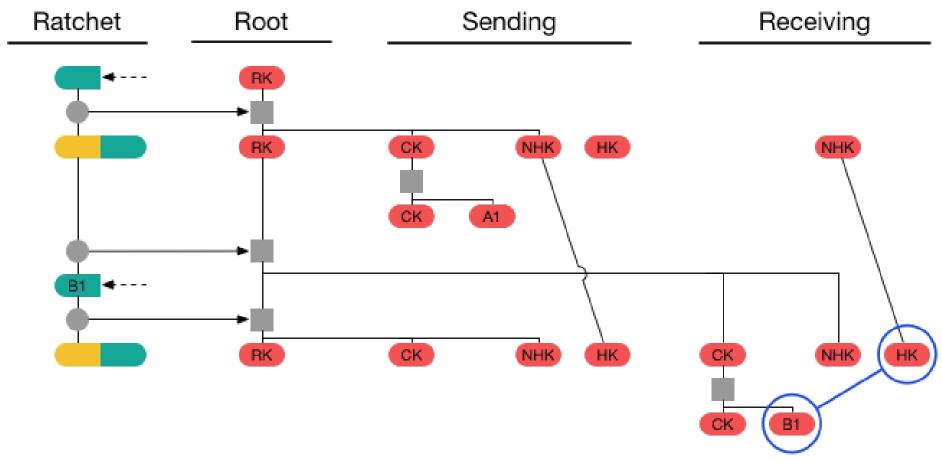

Figure 2.14: When Alice receives the message with the new header key, she performs the DH ratchet step which also derives new header keys [52].

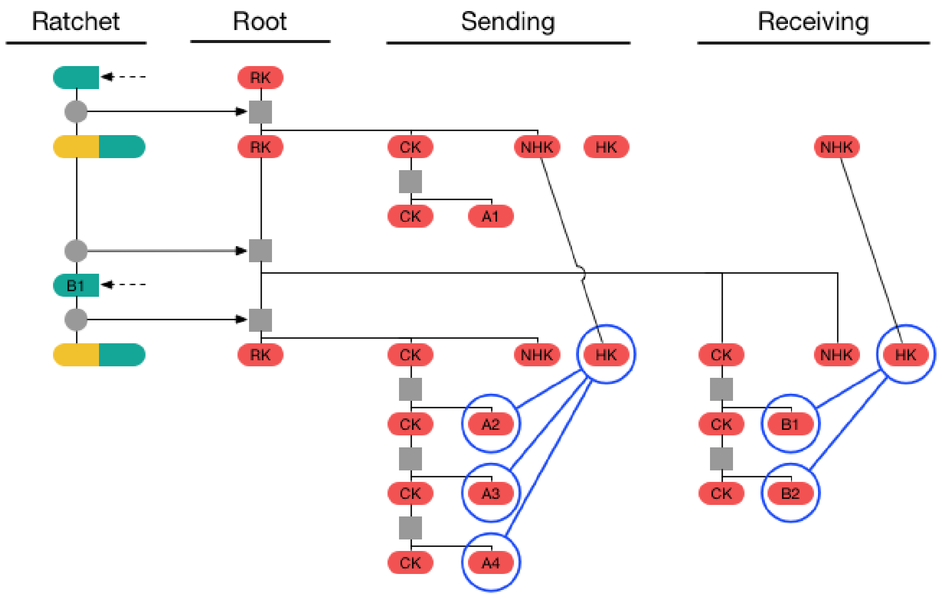

Figure 2.15: When Alice does not receive a new header key from Bob, she only performs the symmetric-key ratchets with the current sending and receiving header keys [52].

Chapter 3

Implementation Analysis

The Signal Protocol is implemented as an open-source code under the GNU General Public License (GPLv3). Three variants of the Signal Protocol library are available on GitHub [53] along with the Signal application for Android phones, iOS, desktops, and more. Anyone can contribute their own code and ideas if they are tested first and approved by the OWS team (via a pull request).

In the Signal Protocol, libraries are implemented in three programming languages – C [54], Java [55] and JavaScript [56], because the Signal Protocol is used on different platforms, e.g. Android mobile phones (Java) or the Electron desktop application (JavaScript), etc.

In this chapter, we will discuss our findings regarding to the Double Ratchet algorithm and several aspects which are closely related to it. The Double Ratchet algorithm is used for deriving new encrypting/decrypting keys from the shared secret. These keys are used for symmetric cryptography between the two parties. It is an essential approach of the Signal Protocol and it provides a better resilience, forward security and break-in recovery than other standard solutions. Theoretical details about the Double Ratchet can be found in section 2.5.

3.1 Code

In this analysis, we focus on the Java library which is used in the Signal Android application, specifically the commit `4f5e1ff299` which represents the most recent Signal Protocol version 2.6.2 (July 12, 2017).

To be able to analyze the Signal Protocol properly, we need to clone the `curve25519-java` [57] repository as well. It contains functions which provide generation of the elliptic curve public and private keys and other operations with the Curve25519 (see section 2.3.2 for details about this topic). In this analysis, we address the commit `0e7a6a1b2b` which represents version 0.4.1 of the Curve25519 library implementation (June 23, 2017).

The Signal Protocol libraries have some interfaces and callback functions which are supposed to be implemented on the client side of an application. For example, the storage functions which decide how and where exactly the client stores data are left unimplemented. The greatest emphasis is to store all private keys (and most importantly the Identity private key) somewhere durable and safe. A developer of the client application should take a great care about these particular parts, because underestimating these security aspects would most certainly lead to fatal security failures.

To cover some of these aspects, we decided to install the Android Studio [58] and compile our own Signal for Android application [59] (commit `9fb67b9f03`, version v4.17.0). *Gradle* [60] (which is used in this repository) ensures all the dependencies are downloaded properly and even though the client analysis is outside of the scope of this thesis, it provided a helpful insight about the interconnection between the Signal Protocol and the client application.

3.2 Keys

In the Signal Protocol, both parties use a set of elliptic curve key pairs for establishing a secured communication. The purpose of the particular keys can be found in sections 2.4 and 2.5.

3.2.1 Key Definitions and Key Generation

First of all, let's see how the keys are represented. Three java classes represent keys in the Signal Protocol Java library:

- `ECPrivateKey.java` – Contains a byte sequence representing the private key
- `ECPublicKey.java` – Contains a byte sequence representing the public key
- `ECKeyPair.java` – Contains both private and public key from the definitions above

However, other definitions of the keys and the cryptographic functions are available in the library `curve25519-java`. It has two sub-directories[16]. The first one is used for the key definitions and the second one is used for the key generators, hash function definitions, etc.

[16]

1. ./`curve25519-java/common/src/main/java/org/whispersystems/curve25519`
2. ./`curve25519-java/java/src/main/java/org/whispersystems/curve25519`

Now, let's focus on how the keys are created in the first place. The abstract class `BaseJavaCurve25519Provider.java` contains a few definitions of the security *providers*. These providers (`Sha512` and `SecureRandomProvider`) are only interfaces, so in this abstract class, we can only find the general approaches of how to generate a new key pair.

The private key (i.e. the byte sequence which is 32 bytes long) is first filled with random data. This is done in `JCESecureRandomProvider.java`. This provider imports standard java library `java.security.SecureRandom` and uses a SHA-1-based pseudo-random number generator (PRNG) to generate the byte sequence. As stated in the official Java documentation [61], PRNGs are statistically tested and they are considered cryptographically strong (see FIPS 140-2 [62] for details).

After the byte sequence has been generated, masks are used to modify the private key:

```
privateKey[0]   &= 248;
privateKey[31]  &= 127;
privateKey[31]  |= 64;
```

where `&=` is a bitwise *and* operator and `|=` is a bitwise *or* operator. The constants represent masks:

- $(248)_{10} = (1111\ 1000)_2$
- $(127)_{10} = (0111\ 1111)_2$
- $(64)_{10} = (0100\ 0000)_2$

These masks correspond to the official RFC 7748 documentation [35] and they allow to decode the byte sequence as an integer scalar. Without it, the private key would not have the proper format for calculations. However, this modification of the private key results in a lower security, because the resulting integer is of the form 2^{254} plus eight times a value between 0 and $2^{251} - 1$ (inclusive) [35].

Now, when the private key is created, the Signal Protocol calculates a corresponding public key as a twisted Edwards point in the Montgomery form (see section 2.3.1 for details). This calculation is performed in `curve_sigs.java` and it takes an Ed25519 base point (which is standard and commonly agreed to) and multiplies[17] it with the private key (which is a scalar represented as a byte sequence). If we mark the result as P, the conversion to the Montgomery x-coordinate is computed using this equation:

$$x = (P.y + 1)/(1 - P.y) \tag{3.1}$$

where $P.y$ is the y-coordinate of the point P. So, the resulting public key is represented as the x-coordinate, thus it is a byte sequence as well.

[17]Multiplication of the base point is fast, because Signal Protocol uses precalculated tables.

3.2.2 Key Security

As we could see in the previous section 3.2.1, the keys are 32 bytes long. Since the keys are used for the elliptic curve cryptography, we have to consider a possibility of attacks using the Baby-step giant-step (BSGS) [63] or the Pollard's Rho algorithm [64]. These algorithms effectively reduce the number of steps which a standard brute-force attack would have to calculate in attempts to solve the Discrete logarithm problem (DLP).

This effectively means that the overall security of the elliptic curve keys is actually the square root of all the possibilities:

$$O(\sqrt{2^{256}}) = O(2^{128}) \tag{3.2}$$

Furthermore, five bits of the private key are deterministically pre-defined, decreasing the overall security even further. Nevertheless, this key length is still sufficient for cryptographic purposes (if we omit the possibility of quantum computers).

3.3 Initialization

Before the actual communication begins, both parties have to be initialized. Two classes represent the initial parameters for Alice and Bob:

- `AliceSignalProtocolParameters.java`
- `BobSignalProtocolParameters.java`

The only difference between the two parties is who is trying to send the message first. If one party decides to send a message, she needs to be initialized beforehand by the second party (and thus she is marked as Alice) and vice versa. All initialization parameters can be found in the table 3.1:

Table 3.1: Initialization parameters of Alice and Bob

Alice	**Bob**
Alice's Identity key pair	Bob's Identity key pair
Alice's Base key pair	Bob's Signed prekey pair
Bob's Identity public key	Bob's Ratchet key
Bob's Signed public prekey	Bob's One-time prekey pair
Bob's Ratchet key	Alice's Identity public key
Bob's One-time public prekey	Alice's Base public key

Note that the one-time prekeys are optional and they do not have to be included during the initialization.

In addition, Alice's Base key pair is an undocumented name. We studied the usage of this key pair in some detail and it is actually the ephemeral key pair which is later used in the X3DH key agreement protocol (EKP_A, see table 2.4). In the old documentation of the TextSecure ProtocolV2 [65] (last edited on July 1, 2015) which contained so called Axolotl ratchet, we have found some mentions of another purpose of the Base key. There were actually two ways to determine which party is Alice or Bob – the first one was described in the paragraphs above, but the second one was based on the comparison of the ephemeral (Base) keys of both parties which they exchanged beforehand. Whoever had a *smaller* Base key would become Alice and the other party would become Bob. This was called the *KeyExchangeMessage* case. The approach which was described in the official documentations [32] was previously called the *PreKeyWhisperMessage* case.

According to the Java library README [55], every party has to generate these key pairs at the installation time:

- Identity key pair
- Prekeys
- Signed prekeys

Alongside these keys, one additional value has to be generated as well:

- Registration ID

The registration ID is a random number generated by the SHA1-PRNG contained in the `KeyHelper.java` file. The actual purpose of this value is undocummented, but it is most likely used as an additional information about the client. The server stores this value which is bound to the device ID (used in the Sesame algorithm [33]) as well. The analysis of the server side is outside of the scope of this thesis. However, the open-source repository of the Signal-Server implementation can be found here [66].

It is also advised to generate more prekeys at the initialization time. These prekeys are sent to the server afterwards. In the Signal Protocol Java library [55], the count 100 is stated as the recommended value. However, the decision of how many prekeys will be generated is left only to the developer who creates the client application and it is not restricted in any way.

3.4 Ratcheting

As stated in section 2.5, the Double Ratchet algorithm contains two types of ratchets:

- Symmetric-ratchet
- Diffie-Hellman Ratchet

Both of these principles used together form the Double Ratchet. However, to explain the ratchet implementation in detail, we have to inspect some other aspects as well.

3.4.1 HKDF

HMAC-based Key Derivation Function (HKDF) is a function which derives one or more secret keys from the secret values given as an input. This function is used for deriving new cryptography keys in the Double Ratchet algorithm. For more detailed information, see section 2.4.3 and section 2.5.1.

In the Signal Protocol Java library, HKDF implementation can be found in the `HKDF.java` file. It uses a standard HMAC-SHA256 algorithm from the `javax.crypto.Mac` library.

Two methods are present in the HKDF implementation as well:

- Extract
- Expand

This corresponds to the HKDF "extract-then-expand" paradigm (see section 2.4.3) [49].

The extract phase takes a salt and an input keying material as parameters. Standard HMAC-SHA256 from the `javax.crypto.Mac` library is used for generating the PRK (a pseudo-random key [49]) as a KDF output.

The PRK is then used as an input to the expand phase, along with an additional information *info* (e.g. the protocol number, algorithm identifier, etc.) and a desired output length L. Additionally, the constant HL (hash-length) is defined which denotes the length of the hash function output in octets [49]. In the Signal Protocol, this constant is equal to 32 bytes. Furthermore, according to the RFC specification, the expand phase should perform these calculations in an attempt to generate the output material OKM:

```
N = ⌈ L/HL ⌉
T = T(1) || T(2) || ... || T(N)
OKM = first L octets of T
```

The T function returns a HMAC output as:

```
T(0) = empty string with zero length
T(1) = HMAC(PRK, T(0) || info || 0x01)
T(2) = HMAC(PRK, T(1) || info || 0x02)
...
T(N) = HMAC(PRK, T(N-1) || info || 0xN)
```

where || is a concatenation of strings. Thus, in the expand phase, the PRK is used as the "salt" and additional information is mixed in directly as the input keying material, along with the previous hash outputs and an incremental byte counter.

In the Signal Protocol, HKDF is implemented exactly according to this specification, with a single exception – it allows to additionally set an offset to the byte counter. Thus, the $T(1)$ function can start with a different arbitrary value. Other values are then incremented as expected.

Both of these phases are encapsulated by the `deriveSecrets` function, which takes the input keying material, additional information and output length as parameters. In this definition, the salt is an optional value and by default it is a zero byte sequence of the length of HL. Even though the salt is an optional value and by default it does not have to be provided[18], it is highly recommended.

As we can later see in the chain key implementation (section 3.4.2) and the root key implementation (section 3.4.3), the salt is used while creating the root key. However, no salt is used while creating the message key – only seed and the previous chain key is used while creating a new input keying material, but no salt is provided to the HKDF.

The output length of the HKDF can be set separately. This allows different byte sequences for different purposes. For example, the HKDF output while creating a new chain key and root chain key (section 3.4.2 and section 3.4.3 below) is 64 bytes long. However, the actual output is split into two 32 byte sequences. The reason for doing so is to achieve a better performance while preserving the security.

3.4.2 Chains and Chain Keys

Chain keys are defined in the file called `ChainKey.java`. Every chain key contains several attributes:

- Message key seed – Equals to a byte `0x01` and it is a constant.
- Chain key seed – Equals to a byte `0x02` and it is a constant.
- KDF – Key derivation function which was definded in section 3.4.1.
- Key – Current chain key represented as a byte sequence.
- Index – Index of the current chain key N. After deriving a new chain key, this number is increased by 1. This information helps to deal with lost or out-of-order messages (see section 2.5.4 for details) and in cases when an excessive number of messages is pending from the sender (messages to-be-received), e.g. due to a possible denial-of-service attack[19].

The `ChainKey.java` also has three methods:

1. `getBaseMaterial()`

[18]The zero byte sequence is explicitly allowed by the specification [49].

[19]In the Signal Protocol, the limit is set to 2000 messages (`SessionCipher.java`).

2. `getNextChainKey()`

3. `getMessageKey()`

The first method takes a seed (message key seed or chain key seed) as a constant and performs a KDF (HMAC-SHA256) over a current chain key. This process will create a new input keying material. Note that this (H)KDF is simpler than the one mentioned in section 3.4.1, because it only takes a current key and a seed as the inputs and generates the output which is used either as a new key or as an input keying material (see paragraphs below).

The second method uses the first method. It derives a new chain key and replaces the current one. The chain key seed is used for the KDF.

The third method also uses the first method. It uses the message key seed to generate the input keying material for a second additional HKDF. This second HKDF is the same as the function described in section 3.4.1. However, no salt is used in this step, so this HKDF takes a zero byte sequence instead. Even though the salt is optional, it is highly recommended to use it. How much security would the additional salting bring to the overall security while generating the message keys should be inspected closer in future work.

The additional information which is passed to the HKDF as well is a string `WhisperMessageKeys` represented as a byte sequence.

The output of the (second) HKDF is then used to create the *Message secrets* (`MessageKeys.java` and `DerivedMessageSecrets.java`):

- Cipher key (32 bytes)
- MAC key (32 bytes)
- Initialization vector (16 bytes)

Thus, the HKDF output has 80 bytes and is split into three subsequences.

Cipher key is a 32 bytes long AES key (a standard `SecretKeySpec` library). MAC key is a standard 32 bytes long HMAC-SHA256 key. The initialization vector is 16 bytes long (`IvParameterSpec` library) which supports a CBC mode.

To clarify the difference between the chain keys (CK) and the message keys (MK), see figure 3.1 below. For even more details about this topic, see section 2.5.3.

3.4.3 Root Chain and X3DH Agreement

As we can see in figure 3.1, the only thing we do not know how to generate is the root key (RK). The root key is covered by the `RootKey.java` file and is very similar to the chain key. However, the root key does not have the index number and it does not use the predefined seeds.

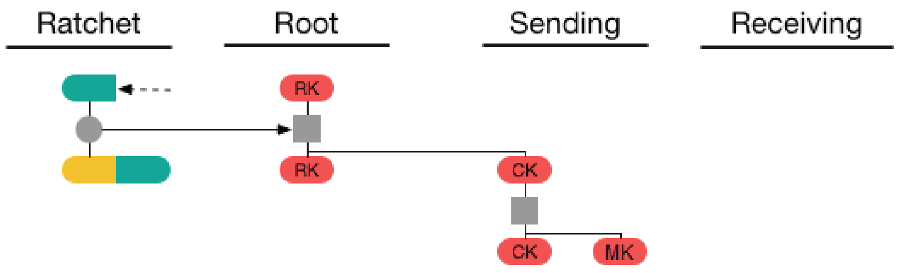

Figure 3.1: An illustration of the difference between the chain keys and the message keys [52].

However, the root chain has to be initialized first. Thus, a shared secret has to be established between Alice and Bob to initialize the root chain as the first step. The X3DH protocol (see section 2.4 for details) is the preferred choice and exactly this algorithm can be found in the `RatchetSession.java` file.

In this class, the `initalizeSession()` method calculates the shared secret for Alice by three or (optionally) four calculations:

- DH calculation between $privIK_A$ and SPK_B
- DH calculation between $privBK_A$ and IK_B
- DH calculation between $privBK_A$ and SPK_B
- DH calculation between $privBK_A$ and OPK_B (optional)

where SPK is the Signed prekey, IK is the Identity key, OPK is the One-time prekey and the BK is the Base key which is actually the ephemeral key (see section 3.3). The $priv$ denotes the private key, otherwise we mean the public key (see sections 2.4.1 and 2.4.5 for more details).

Similarly, the `initializeSession()` method calculates the shared secret for Bob:

- DH calculation between $privSPK_B$ and IK_A
- DH calculation between $privIK_B$ and BK_A
- DH calculation between $privSPK_B$ and BK_A
- DH calculation between $privOPK_B$ and BK_A (optional)

As we can see, this perfectly corresponds to the shared secret establishment in the X3DH agreement protocol which was described in section 2.4.

The DH calculation from the lists above is a simple Diffie-Hellman calculation, i.e. a multiplication of Alice's private key (scalar) and Bob's public key (elliptic curve point) and vice versa. This calculation can be found in the `BaseJavaCurve25519Provider.java` file using the `calculateAgreement()` method.

The shared secret is 64 bytes long. After it is established, it is then split into two 32 byte sequences. The first one is used as an initial root key and the second one is used as an initial receiving chain key which is used to decrypt the very first initial message.

After the initial root key is created, we can derive a new sending chain key by a simple additional DH calculation and a HKDF calculation (see figure 2.7). The algorithm takes the other party's DH ratchet public key and multiplies it with our party's private key. This shared secret is then used as the input keying material for the HKDF (section 3.4.1). Unlike the chain keys (section 3.4.2), this HKDF uses a salt – the current root key. The additional information is a string `WhisperRatchet` represented as a bytes sequence. The output of this HKDF (64 bytes) is then split and used as a new root key and a new sending chain key (both 32 bytes).

3.5 Encryption and Decryption of Messages

In the Signal Protocol implementation, encrypting and decrypting messages is based on working with sessions. Even though a session management and the Sesame algorithm [33] are outside of the scope of this thesis, we can still inspect some cryptographic practices.

The `javax.crypto.Cipher` is a standard library which allows to encrypt and decrypt messages with a usage of the standard cryptography algorithms. It is used in the `SessionCipher.java` class.

3.5.1 Encryption

Before the encryption or the decryption starts, a cipher must be created. In the Signal Protocol, the standard AES in CBC mode with PKCS5 padding is used. According to the official Java documentation [67], this standard AES uses 128-bit keys. However, Java allows to use the Java Cryptography Extension (JCE) which adds the possibility to have longer keys and use more secure variants of the standard algorithms, e.g. AES with a support for 256 bits key length. Thus, the AES-256 is used for the symmetric encryption.

The cipher has to be initialized with a message key and an initialization vector, too. These values are known from the chain key ratchet (see section 3.4.2 for details).

When the cipher is ready, plaintext can be encrypted using freshly derived message keys. After every encryption, a chain key ratchet is performed. Thus, with another encryption, new message keys will be derived from a new chain key.

3.5.2 MAC Calculation

While encrypting the message, a MAC has to be calculated as well. While deriving new message secrets from the chain key, a MAC key is created along with a message key and an initialization vector (see section 3.4.2 for details).

The `SignalMessage.java` file contains both the MAC calculation method and the MAC verification method. They both use a standard HMAC-SHA256.

In the MAC calculation method, HMAC takes four values as an input. All of these values are already available – they are either contained in the derived secrets, or they are provided by the server, or they are present on the client side:

- MAC key (32 bytes)
- Sender's Identity public key (32 bytes)
- Receiver's Identity private key (32 bytes)
- Ciphertext message and a message version number as a concatenated byte sequence

From these values, an 8 byte MAC sequence is generated[20].

3.5.3 Decryption

The decryption is a little more complicated. It differs depending on the state in which the algorithm currently is, i.e. if the party is initializing the communication with the first message, or the session is in an ongoing communication.

In the Signal Protocol, a message can take up two forms, represented by two files:

- `PreKeySignalMessage.java`
- `SignalMessage.java`

The first form contains all the necessary information for establishing the secured communication, e.g. the registration ID, the prekeys and the signed prekeys, the identity public key, the base key, etc. So, this is the initial message which also carries a ciphertext along with the parameters.

The second form only contains a Diffie-Hellman ratchet public key, ciphertext and counters (the numbers N and PN from the section 2.5.4).

These parameters are stored into a session. This session can be later recreated, stored and modified according to the incoming values.

Before the message can be decrypted, new message keys and chain keys have to be created (derived) first. The `SessionCipher.java` file contains two methods for deriving the new keys if necessary. If the received Signal message

[20]Only first 8 bytes are taken from the HKDF output.

carries a new DH public ratchet key, root chain ratchet is performed on the recipient side as well. Otherwise, the recipient uses her current receiving chain. In either case, the recipient has to derive new message keys or she uses some of the keys from the previously skipped messages.

Note that the DH ratchet step creates a new DH ratchet key pair on the recipient side as well, resulting in sending a new DH ratchet public key to the original sender, if the recipient decides to send her own message.

3.5.3.1 Skipped Messages

The client must check all of the previously stored (skipped) chain keys, if there are any available, because the received message could be delayed in transit. This is done by finding the correct receiving chain (using the sender's DH ratchet public key) and performing a maximum of N iterations over the stored message keys, comparing their indexes with the value N which arrived with the incoming message. If the correct message key is found, it is then used for decryption and deleted.

Note that the index N denotes a position in the current sending chain. This number is the same for both parties, i.e. for Alice's receiving chain and Bob's sending chain and vice versa.

A maximum of 2000 skipped messages (message keys) is set as a limit on the message loss. It also serves as a precaution against possible denial-of-service attacks.

A greater number N can arrive with the message as well. To be able to decrypt this message, a client has to perform $(N - RC)$ derivations of the new message keys, storing them in the process; RC is an index of the last message key in the receiving chain.

3.5.3.2 Unsuccessful Decryption and Archived Sessions

Unsuccessful decryption can be caused by several factors. For example, the message can be somehow damaged (i.e. incomplete), or the MAC verification is not valid (see section 3.5.4 below for details). To identify all the causes of the unsuccessful decryption is outside of the scope of this thesis.

In addition, the Signal Protocol counts a number of *archived* sessions (a supplement to the active session) with the maximum threshold of 40 sessions (`SessionRecord.java`). If the number of stored sessions is exceeded, the last used session is deleted and a new one is created (and marked as active).

3.5.4 MAC Verification

Before the decryption of the ciphertext, a MAC verification must be performed. To be able to do so, the chain key and the message key which were derived during the decryption process are used. MAC verification then takes the message (ciphertext and message version) and calculates the MAC using the approach which was described in section 3.5.2. If the calculated MAC is equal to the one that arrived with the message, the Signal Protocol marks the message as verified and the message can be decrypted. If not, the Double Ratchet terminates the message and reverts all changes.

3.6 Header Encryption

In the official documentation regarding the Double Ratchet algorithm [52] (section 2.5.5), a header encryption variant of the Double Ratchet can be found. However, we did not find any trace of this algorithm in the code. We have tried to contact the developers from the OWS team regarding the implementation of this variant, but unfortunately, without success.

We suspect that this more secured variant of the Double Ratchet is not implemented yet. Thus, we would expect a disclaimer in the official documentation.

3.7 Cleanup

In the section 2.5.3, we have described which keys should be deleted in the Double Ratchet algorithm for the forward secrecy. Keys should be deleted when they are not used anymore and when they will not be required in the future.

Furthermore, as mentioned in the Code section 3.1, storing (and cleaning) data is left to the developer of the particular application which uses the Signal Protocol. For example, in the Signal for Android client application, a *protobuf* by Google is used as a solution for storing the keys and other particular data. However, the exact functionality of the so called "Protocol Buffers" is outside of the scope of this thesis and we discuss the implemented general security principles instead.

Due to the importance of storing the data, we would expect some guidelines regarding this topic in the protocol documentation. However, we couldn't find any. A developer of the client application should use caution when implementing these parts.

3.7.1 Root Key Deletion

When a new DH ratchet public key is received with the incoming message, a DH ratchet step is performed. This step will derive a new root key. The previous root key will not be used ever again, thus it is obsolete and should be deleted.

Effectively, only one root key is saved at any given time, effectively replacing the old key with a new one after every DH ratchet step.

3.7.2 Chain Key Deletion

As was described before, the client uses two chains for communication – the sending chain and the receiving chain.

Similarly to the root key, only one sending chain key is stored at any given time. This means the old sending chain key is replaced by a new one after every key derivation.

On the other hand, the receiving chain key is deleted after every successful decryption. The cases of a successful decryption were discussed in section 3.5.3.

3.7.3 Message Key Deletion

All message keys are derived from the chain keys, either from the sending chain or the receiving chain.

The sending message keys are only used to encrypt an outgoing message and to calculate the MAC. Thus, they are not stored in any way and they are discarded immediately after they were used. This means they are only stored in-memory for a very short period of time.

As previously mentioned, when a new message is received, the client has to iterate over all the skipped message keys in her receiving chain. She stores this information in the sessions by keeping the (non-empty) receiving chains with relevant message keys.

The client firstly finds the correct receiving chain (using the sender's DH ratchet public key) and then it iterates through all the message keys in this chain, until the index N is found. If the correct message key is found and the received message is successfully decrypted, the message key is deleted. Note that the client's receiving chains have that many message keys equal to the number of skipped messages in that particular receiving chain.

Chapter 4

Evaluation of Results

The Signal Protocol uses only well known cryptographic standards which are continuously review by the community and security researchers. This ensures the protocol is based on standard and recommended cryptographic solutions and we did not find any discrepancies regarding this matter.

The standard AES-256 is used for the message encryption. Asymmetric cryptography is based on the elliptic curve calculations. The standard Curve25519 is used in the Signal Protocol. The OWS team also proposes the possibility to use the Curve448 (Ed448-Goldilocks) which sacrifices some performance over security. However, a quantum computer will be able to crack both of these curves. Thus, the Curve25519 is recommended and is used in the Signal Protocol for the X3DH agreement protocol and the (V)XEdDSA signature scheme.

Because of both the BSGS and the Pollard's Rho algorithms, the overall security of the elliptic curve keys is the square root of all the possibilities. Thus, only 128 bits key length limits the elliptic curve security (instead of 256 bits). Nevertheless, this key length is still sufficient for cryptography used in the classic computers.

Two forms of elliptic curves are defined for the calculations and storing data – the Edwards curve and the Montgomery curve. These forms allow to store only one coordinate of the curve's point resulting in a point compression along with the resistance against side-channel attacks. This is useful for the mobile devices (for which the protocol is originally designed) because they often lack of a large storage space and they could be under a wide range of (side-channel) attacks.

The cryptographic standards are combined into a very robust cryptosystem with XEdDSA, X3DH, and Double Ratchet algorithms at its heart. These algorithms can be also used separately. The XEdDSA and the X3DH algorithms provide a well secured and digitally signed establishment of the shared secret. The shared secret is derived by the Double Ratchet to new symmetric keys forming the perfect forward secrecy.

If an attacker could sacrifice enough resources for cracking the symmetric key, she would only obtain information about one message. The cracked key does not contain any information about previous and future keys because new Diffie-Hellman values are exchanged continuously during the communication and the derivation of keys ensures random-looking outputs. This could be praised as “as good as it gets” for the cryptographic security, because the attacker would have to expend a tremendous effort in order to decrypt the whole communication.

The Signal Protocol is completely open-source which allows anyone to perform a security analysis. In general, we highly recommend to use the open-source solutions for security systems like this. First of all, it shows that the authors have “nothing to hide”. Of course, it does not provide any information about the security of such a solution, but it mitigates the phenomenon of *security by obscurity* which should have no place in cryptography.

4.1 Security Considerations

We conclude that the Signal Protocol exhibits strong security features. However, we found a few discrepancies and undocumented specifications which the reader should carefully consider along with a few other security concerns.

The stated remarks do not affect security of the Signal Protocol as it exists now. However, they could be significant in the future and/or the alternative implementations could be affected.

4.1.1 Header Encryption

The header encryption variant of the Double Ratchet algorithm represents a more secure approach of distributing an additional information along with the message. The header information is encrypted using a set of additional *header keys.*

This process is described in the official Double Ratchet documentation [52] and section 2.5.5. However, we did not find any trace of this variant in the code. We have tried to contact the developers of the OWS team regarding this issue, but unfortunately, without success.

We suspect that this more secured variant of the Double Ratchet algorithm is not implemented. Thus, we would expect a disclaimer in the official documentation and/or the OWS team should provide more information regarding this topic.

4.1.2 Storing Data

The protocol is used on several platforms and countless devices. Thus, it is understandable that some implementation details are left to the developer of the particular application which uses the Signal Protocol.

In our thesis, we covered only the Java implementation of the protocol. This implementation uses a *protobuf* by Google in the official Signal for Android application as a solution for storing the keys and other particular data. However, we found only few general suggestions in the official protocol repository [55] regarding how the developer should implement the data storage in her own client application.

Because of the importance of storing the sensitive data (such as the user's private keys), we would expect to see further information or guidelines regarding this topic.

4.1.3 Authentication

The Signal Protocol does not provide any cryptographic guarantee of the authenticity of communicating parties. Public keys should be verified out-of-band using another communication channel (e.g. in person with a fingerprint written on a piece of paper). Without it, the communication could already be intercepted using a man-in-the-middle attack.

In the Signal for Android application, users can compare their fingerprints in person (e.g. in the form of the QR code). This procedure is highly recommended and the developer who wants to use the Signal Protocol in her own application should also implement a similar solution.

4.1.4 Post-Quantum Cryptography

As mentioned, the design of the Signal Protocol is heavily secured against attacks using the classic computers. However, another concern is the possible uprising of the quantum computers.

From the security perspective, we can only assume that all our communication is being recorded. Even though the communication is (or should be) encrypted and it cannot be decrypted at the moment, the quantum computers will be able to crack all commonly used asymmetric systems like RSA, Diffie-Hellman, and ECDH – i.e. including the cryptography used in the Signal Protocol.

Solution for securing the future messages is to use a post-quantum cryptography which could be deployed into the existing cryptosystems. However, the already recorded messages would still be decrypted.

Conclusion

In this thesis we described a selection of the currently used instant messaging protocols with a particular focus on their security. In the selection, we included utterly unsecured protocols (the IRC in its original design) as well as very robust security solutions (the Signal Protocol).

We analyzed the Signal Protocol in detail. In this thesis, mathematical aspects of the protocol's functionality were outlined and we explained how these preliminaries improve the security of the cryptographic operations. Moreover, we covered the essential approaches of the Signal Protocol such as the Double Ratchet algorithm, the Extended Triple Diffie-Hellman key agreement protocol or the XEdDSA and the VXEdDSA signature schemes.

Furthermore, we analyzed the Java implementation of the open-source Signal libraries which are currently used in the official Signal for Android application. We validated the theoretical security considerations against the particular solution from the Open Whisper Systems team. Hence we conclude that the Signal Protocol exhibits strong security features.

Moreover, we clarified the security aspects regarding the cryptographic operations which are used in the Signal Protocol implementation, such as the HKDF, the elliptic curve calculations, the key derivation (ratcheting), and more.

We also found a few discrepancies and undocumented operations in the protocol's implementation compared to the official documentations, such as the header encryption variant of the Double Ratchet algorithm. We evaluated such concerns and raised several recommendations.

Finally, we brought up many security considerations which emerged from the protocol's design, such as the server trust or the need for separated authentication channels. Even though a majority of them cannot be simply avoided by the protocol's implementation, we described the security recommendations that a user should take if she would wish to include the Signal Protocol in her own application.

Our work is focused mainly on the protocol's functionality, design and its Java implementation. However, there are still many aspects which might be addressed by additional research in the future. This research might focus on the Sesame algorithm and a session management between multiple devices. Since this thesis only analyzed the Java implementation of the Signal Protocol libraries, other research might target the C and JavaScript implementations, too.

Bibliography

[1] XMPP. *XMPP Instant Messaging* [online]. [accessed 2018-04-21]. Available from: `https://xmpp.org/uses/instant-messaging.html`

[2] Oikarinen, J.; Reed, D. *Internet Relay Chat Protocol*. RFC 1459, RFC Editor, May 1993, [accessed 2018-04-05]. Available from: `https://tools.ietf.org/html/rfc1459`

[3] Kalt, C. *Internet Relay Chat: Architecture*. RFC 2810, RFC Editor, April 2000, [accessed 2018-04-05]. Available from: `https://tools.ietf.org/html/rfc2810`

[4] Kalt, C. *Internet Relay Chat: Client Protocol*. RFC 2812, RFC Editor, April 2000, [accessed 2018-04-05]. Available from: `https://tools.ietf.org/html/rfc2812`

[5] Lo, J. *Denial of Service or "Nuke" Attacks* [online]. March 2005, [accessed 2018-04-05]. Available from: `http://www.irchelp.org/nuke/`

[6] Hartmann, R. *Default Port for Internet Relay Chat (IRC) via TLS/SSL*. RFC 7194, RFC Editor, August 2014, [accessed 2018-04-05]. Available from: `https://tools.ietf.org/html/rfc7194`

[7] Borisov, N.; Goldberg, I.; et al. *Off-the-Record Communication, or, Why Not To Use PGP* [online]. October 2004, [accessed 2018-04-07]. Available from: `https://otr.cypherpunks.ca/otr-wpes.pdf`

[8] Jabber.org. *Frequently Asked Questions* [online]. [accessed 2018-04-09]. Available from: `https://www.jabber.org/faq.html`

[9] XMPP. *An Overview of XMPP* [online]. [accessed 2018-04-21]. Available from: `https://xmpp.org/about/technology-overview.html`

[10] Waher, P.; Doi, Y. *XEP-0322: Efficient XML Interchange (EXI) Format* [online]. Last updated: January 2018, [accessed 2018-04-22]. Available from: `https://xmpp.org/extensions/xep-0322.html`

[11] Jabber.org. *Service Policy* [online]. August 2014, [accessed 2018-04-22]. Available from: `https://www.jabber.org/policy.html`

[12] Saint-Andre, P. *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP).* RFC 3923, RFC Editor, October 2004, [accessed 2018-04-22]. Available from: `https://tools.ietf.org/html/rfc3923`

[13] Ramsdell, B. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification.* RFC 3851, RFC Editor, July 2004, [accessed 2018-04-23]. Available from: `https://tools.ietf.org/html/rfc3851`

[14] Let's Encrypt. *Let's Encrypt homepage* [online]. [accessed 2018-04-22]. Available from: `https://letsencrypt.org/`

[15] Telegram. *FAQ for the Technically Inclined* [online]. [accessed 2018-04-07]. Available from: `https://core.telegram.org/techfaq`

[16] Telegram. *Telegram FAQ* [online]. [accessed 2018-04-06]. Available from: `https://telegram.org/faq`

[17] Jakobsen, J. B. *A practical cryptanalysis of the Telegram messaging protocol* [online]. September 2015, [accessed 2018-04-07]. Available from: `http://cs.au.dk/~jakjak/master-thesis.pdf`

[18] Laurie, B. *OpenSSL's Implementation of Infinite Garble Extension* [online]. August 2006, [accessed 2018-04-07]. Available from: `http://www.links.org/files/openssl-ige.pdf`

[19] Telegram. *Mobile Protocol: Detailed Description* [online]. [accessed 2018-04-06]. Available from: `https://core.telegram.org/mtproto/description`

[20] Dunn, J. E.; Macaulay, T.; et al. *Best secure mobile messaging apps* [online]. February 2018, [accessed 2018-04-27]. Available from: `https://www.techworld.com/security/best-secure-mobile-messaging-apps-3629914/`

[21] McCormick, R. *Edward Snowden's favorite encrypted chat app is now on Android* [online]. The Verge, November 2015, [accessed 2018-04-08]. Available from: `https://www.theverge.com/2015/11/3/9662724/signal-encrypted-chat-app-android-edward-snowden`

[22] Chen, B. X. *Worried About the Privacy of Your Messages? Download Signal* [online]. The New York Times, December 2016, [accessed 2018-04-08]. Available from: `https://www.nytimes.com/2016/12/07/technology/personaltech/worried-about-the-privacy-of-your-messages-download-signal.html`

[23] Marlinspike, M. *Advanced cryptographic ratcheting* [online]. Open Whisper Systems, November 2013, [accessed 2018-04-08]. Available from: `https://signal.org/blog/advanced-ratcheting/`

[24] Frosch, T.; Mainka, C.; et al. *How Secure is TextSecure?* [online]. Cryptology ePrint Archive, Report 2014/904, October 2014, [accessed 2018-04-08]. Available from: `https://eprint.iacr.org/2014/904.pdf`

[25] Marlinspike, M. *Signal on the outside, Signal on the inside* [online]. Open Whisper Systems, March 2016, [accessed 2018-04-08]. Available from: `https://signal.org/blog/signal-inside-and-out/`

[26] Marlinspike, M. *Open Whisper Systems partners with WhatsApp to provide end-to-end encryption* [online]. Open Whisper Systems, November 2014, [accessed 2018-04-08]. Available from: `https://signal.org/blog/whatsapp/`

[27] Marlinspike, M. *Open Whisper Systems partners with Google on end-to-end encryption for Allo* [online]. Open Whisper Systems, May 2016, [accessed 2018-04-08]. Available from: `https://signal.org/blog/allo/`

[28] Marlinspike, M. *Facebook Messenger deploys Signal Protocol for end-to-end encryption* [online]. Open Whisper Systems, July 2016, [accessed 2018-04-08]. Available from: `https://signal.org/blog/facebook-messenger/`

[29] Lund, J. *Signal partners with Microsoft to bring end-to-end encryption to Skype* [online]. Open Whisper Systems, January 2018, [accessed 2018-04-08]. Available from: `https://signal.org/blog/skype-partnership/`

[30] Wire Swiss GmbH. *Wire Security Whitepaper* [online]. September 2017, [accessed 2018-04-08]. Available from: `https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf`

[31] Marlinspike, M. *Signal Foundation* [online]. Open Whisper Systems, February 2018, [accessed 2018-03-25]. Available from: `https://signal.org/blog/signal-foundation/`

[32] Open Whisper Systems. *Signal Specifications* [online]. 2013-2018, [accessed 2017-11-21]. Available from: `https://signal.org/docs/`

[33] Marlinspike, M.; Perrin, T. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption* [online]. Open Whisper Systems, April 2017, [accessed 2017-11-22]. Available from: `https://signal.org/docs/specifications/sesame/`

[34] Perrin, T. *The XEdDSA and VXEdDSA Signature Schemes* [online]. Open Whisper Systems, October 2016, [accessed 2017-11-22]. Available from: `https://signal.org/docs/specifications/xeddsa/`

[35] Langley, A.; Hamburg, M.; et al. *Elliptic Curves for Security*. RFC 7748, RFC Editor, January 2016, [accessed 2018-01-22]. Available from: `https://tools.ietf.org/html/rfc7748`

[36] Josefsson, S.; Liusvaara, I. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032, RFC Editor, January 2017, [accessed 2018-01-23]. Available from: `https://tools.ietf.org/html/rfc8032`

[37] Micali, S.; Rabin, M.; et al. *Verifiable Random Functions* [online]. 1999, [accessed 2017-12-12]. Available from: `https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Pseudo%20Randomness/Verifiable_Random_Functions.pdf`

[38] Bernstein, D. J.; Birkner, P.; et al. *Twisted Edwards Curves* [online]. Cryptology ePrint Archive, Report 2008/013, March 2008, [accessed 2018-01-22]. Available from: `https://eprint.iacr.org/2008/013.pdf`

[39] Bernstein, D. J.; Lange, T. *Faster addition and doubling on elliptic curves* [online]. Cryptology ePrint Archive, Report 2007/286, September 2007, [accessed 2018-01-24]. Available from: `https://eprint.iacr.org/2007/286.pdf`

[40] Hisil, H.; Wong, K. K.-H.; et al. *Twisted Edwards Curves Revisited* [online]. International Association for Cryptologic Research, 2008, [accessed 2018-01-22]. Available from: `https://iacr.org/archive/asiacrypt2008/53500329/53500329.pdf`

[41] Edwards, H. M. *A Normal Form For Elliptic Curves* [online]. American Mathematical Society, April 2007, [accessed 2018-01-22]. Available from: `http://www.ams.org/journals/bull/2007-44-03/S0273-0979-07-01153-6/S0273-0979-07-01153-6.pdf`

[42] Castryck, W.; Galbraith, S.; et al. *Efficient arithmetic on elliptic curves using a mixed Edwards–Montgomery representation* [online]. Cryptology ePrint Archive, Report 2008/218, 2008, [accessed 2018-02-13]. Available from: `https://eprint.iacr.org/2008/218.pdf`

[43] Bernstein, D. J.; Duif, N.; et al. *High-speed high-security signatures* [online]. Journal of Cryptographic Engineering, July 2011, [accessed 2018-02-14]. Available from: `https://ed25519.cr.yp.to/ed25519-20110705.pdf`

[44] Jivsov, A. *Compact representation of an elliptic curve point* [online]. Network Working Group, March 2014, [accessed 2018-02-14]. Available from: `https://tools.ietf.org/html/draft-jivsov-ecc-compact-05`

[45] Bernstein, D. J.; Lange, T. *SafeCurves: choosing safe curves for elliptic-curve cryptography* [online]. [accessed 2018-02-15]. Available from: `https://safecurves.cr.yp.to/`

[46] Hamburg, M. *Ed448-Goldilocks, a new elliptic curve* [online]. Cryptology ePrint Archive, Report 2015/625, 2015, [accessed 2018-02-16]. Available from: `https://eprint.iacr.org/2015/625.pdf`

[47] Marlinspike, M.; Perrin, T. *The X3DH Key Agreement Protocol* [online]. Open Whisper Systems, November 2016, [accessed 2017-11-22]. Available from: `https://signal.org/docs/specifications/x3dh/`

[48] Bernstein, D. J. *Curve25519: new Diffie-Hellman speed records* [online]. February 2006, [accessed 2018-02-27]. Available from: `https://www.iacr.org/cryptodb/archive/2006/PKC/3351/3351.pdf`

[49] Krawczyk, H.; Eronen, P. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869, RFC Editor, May 2010, [accessed 2018-02-26]. Available from: `https://tools.ietf.org/html/rfc5869`

[50] Rogaway, P. *Authenticated-Encryption with Associated-Data* [online]. September 2002, [accessed 2018-02-27]. Available from: `http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf`

[51] Krovetz, T.; Rogaway, P. *The OCB Authenticated-Encryption Algorithm*. RFC 7253, RFC Editor, May 2014, [accessed 2018-02-28]. Available from: `https://tools.ietf.org/html/rfc7253`

[52] Marlinspike, M.; Perrin, T. *The Double Ratchet Algorithm* [online]. Open Whisper Systems, November 2016, [accessed 2017-11-22]. Available from: `https://signal.org/docs/specifications/doubleratchet/`

[53] Open Whisper Systems. *Signal repositories* [online]. [accessed 2018-03-09]. Available from: `https://github.com/signalapp/`

[54] Open Whisper Systems. *Signal Protocol C Library* [online]. [accessed 2018-03-09]. Available from: `https://github.com/signalapp/libsignal-protocol-c`

[55] Open Whisper Systems. *Signal Protocol Java Library* [online]. [accessed 2018-03-09]. Available from: `https://github.com/signalapp/libsignal-protocol-java`

[56] Open Whisper Systems. *Signal Protocol JavaScript Library* [online]. [accessed 2018-03-09]. Available from: `https://github.com/signalapp/libsignal-protocol-javascript`

[57] Open Whisper Systems. *Backed Curve25519 implementation* [online]. [accessed 2018-03-14]. Available from: `https://github.com/signalapp/curve25519-java`

[58] Google Inc. *Android Studio* [online]. [accessed 2018-03-20]. Available from: `https://developer.android.com/studio/index.html`

[59] Open Whisper Systems. *Signal for Android* [online]. [accessed 2018-03-16]. Available from: `https://github.com/signalapp/Signal-Android`

[60] Gradle Inc. *Gradle Build Tool homepage* [online]. [accessed 2018-03-20]. Available from: `https://gradle.org/`

[61] Oracle and/or its affiliates. *Class SecureRandom* [online]. [accessed 2018-03-16]. Available from: `https://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html`

[62] NIST. *FIPS PUB 140-2: Security Requirements for Cryptographic Modules* [online]. May 2001, [accessed 2018-03-16]. Available from: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf`

[63] Blackburn, S. R.; Teske, E. *Baby-Step Giant-Step Algorithms for Non-uniform Distributions*. Springer, 2000. In: Bosma W. (eds) Algorithmic Number Theory. ANTS 2000. Lecture Notes in Computer Science, vol 1838. Springer, Berlin, Heidelberg.

[64] Pollard, J. M. *A monte carlo method for factorization* [online]. Springer, 1975, [accessed 2018-03-25]. Available from: `https://doi.org/10.1007/BF01933667`

[65] Open Whisper Systems. *ProtocolV2* [online]. [accessed 2018-03-16]. Available from: `https://github.com/JavaJens/TextSecure/wiki/ProtocolV2`

[66] Open Whisper Systems. *Signal Server* [online]. [accessed 2018-03-16]. Available from: `https://github.com/signalapp/Signal-Server`

[67] Oracle and/or its affiliates. *Class Cipher* [online]. [accessed 2018-03-30]. Available from: `https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html`

Appendix A

Acronyms

AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
AOL	America Online
BSGS	Baby-Step Giant-Step
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
FIPS	Federal Information Processing Standard
HKDF	HMAC-based Key Derivation Function
HMAC	Hash-based Message Authentication Code
IETF	Internet Engineering Task Force
IGE	Infinite Garble Extension
IM	Instant Messaging
IRC	Internet Relay Chat
JCE	Java Cryptography Extension
KDF	Key Derivation Function
NIST	National Institute of Standards and Technology

OTR	Off-the-Record
OWS	Open Whisper Systems
PRNG	Pseudo-random Number Generator
RFC	Requests for Comments
RSA	Rivest–Shamir–Adleman
SHA	Secure Hash Algorithm
SMTP	Simple Mail Transfer Protocol
SSL	Secure Sockets Layer
XMPP	Extensible Messaging and Presence Protocol

Appendix B

Contents of the enclosed CD

```
│ readme.txt ....................... the file with CD contents description
├─ repositories.zip ................... the copy of the Signal repositories
│  ├─ curve25519-java .................... the curve25519-java repository
│  ├─ libsignal-protocol-java ..... the libsignal-protocol-java repository
│  └─ Signal-Android ...................... the Signal-Android repository
├─ src ..................................... the directory of source codes
│  └─ thesis .............. the directory of LaTeX source codes of the thesis
└─ text ........................................ the thesis text directory
   └─ thesis.pdf .......................... the thesis text in PDF format
```