CRYPTOGRAPHIC PROTOCOLS OF SIGNAL AND SIGNAL BASED INSTANT MESSAGING APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HİLAL DİNÇER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

AUGUST 2022

Approval of the thesis:

CRYPTOGRAPHIC PROTOCOLS OF SIGNAL AND SIGNAL BASED INSTANT MESSAGING APPLICATIONS

submitted by **HİLAL DİNÇER** in partial fulfillment of the requirements for the degree of **Master of Science in Cryptography Department, Middle East Technical University** by,

Prof. Dr. Sevtap Kestel
Dean, Graduate School of **Applied Mathematics** _______________

Assoc. Prof. Dr. Oğuz Yayla
Head of Department, **Cryptography** _______________

Assoc. Prof. Dr. Ali Doğanaksoy
Supervisor, **Mathematics, METU** _______________

Dr. Pınar Gürkan Balıkçıoğlu _______________
Co-supervisor, **Cryptographer, Ankara**

Examining Committee Members:

Prof. Dr. Murat Cenk
Cryptography, IAM, METU _______________

Assoc. Prof. Dr. Ali Doğanaksoy
Mathematics, METU _______________

Assoc. Prof. Dr. Fatih Sulak
Mathematics, Atılım University _______________

Date:

ABSTRACT

CRYPTOGRAPHIC PROTOCOLS OF SIGNAL AND SIGNAL BASED INSTANT MESSAGING APPLICATIONS

DİNÇER, HİLAL
M.S., Department of Cryptography
Supervisor :Assoc. Prof. Dr. Ali Doğanaksoy
Co-Supervisor : Dr. Pınar Gürkan Balıkçıoğlu

August 2022, 64 pages

Instant messaging applications have replaced classical messaging in recent years. The fact that instant messaging applications transmit messages over the internet, therefore, being free and fast, played a major role in this rise. However, being internet-based has brought disadvantages as well as advantages. There are risks such as obtaining the message, changing the message, etc. by third parties. To avoid these risks, messages are encrypted, the sender is authenticated and their integrity is shown. However, with the developing quantum technology, it turned out that these algorithms will be broken in the near future. Now, studies are being made to make these algorithms resistant to post-quantum. In this thesis study, the key generation, key exchange, and encryption mechanisms used by the Signal Protocol in one-to-one communications, which is one of the most secure systems, are explained in detail. It is explained how open source Linphone, Xabber, Wire, and Element applications developed on the basis of Signal Protocol use Signal Protocol. In addition, in this thesis, the parameters used by Signal and Wire applications, but not specified in their documents, were obtained from open sources and added. Finally, the methods used to make the Signal Protocol quantum resistant are presented.

Keywords: Cryptography, Cryptographic protocols, Signal, Double Ratchet, Protocols of Instant Messaging Applications, Key Exchange, Encryption, Post-quantum, Quantum

ÖZ

SİGNAL VE SİGNAL TABANLI ANLIK MESAJLAŞMA UYGULAMALARINDAKİ KRİPTOGRAFİK PROTOKOLLER

DİNÇER, HİLAL
Yüksek Lisans, Kriptoloji Bölümü
Tez Yöneticisi : Doç. Dr. Ali Doğanaksoy
Ortak Tez Yöneticisi : Dr. Pınar Gürkan Balıkçıoğlu

Ağustos 2022 , 64 sayfa

Anlık mesajlaşma uygulamaları son yıllarda klasik mesajlaşmanın yerini almıştır. Bu yükselişte anlık mesajlaşma uygulamalarının mesajları internet üzerinden iletmesi, dolayısıyla ücretsiz ve hızlı olması büyük rol oynamıştır. Ancak internet tabanlı olması avantajların yanında dezavantajları da beraberinde getirmiştir. Mesajın üçüncü şahıslar tarafından alınması, mesajın değiştirilmesi vb. riskler vardır. Bu risklerden kaçınmak için mesajlar şifrelenir, gönderenin kimliği doğrulanır ve bütünlükleri gösterilir. Ancak gelişen kuantum teknolojisi ile yakın gelecekte bu algoritmaların kırılacağı ortaya çıktı. Günümüzde, bu algoritmaları post-kuantuma dayanıklı hale getirmek için çalışmalar yapılıyor. Bu tez çalışmasında, en güvenli sistemlerden biri olan Signal Protokol'ünün bire bir iletişimde kullandığı anahtar üretimi, anahtar değişimi ve şifreleme mekanizmaları detaylı olarak anlatılmıştır. Signal Protokol'ü temel alınarak geliştirilen açık kaynaklı Linphone, Xabber, Wire ve Element uygulamalarının Signal Protokol'ünü nasıl kullandığı açıklanmıştır. Ayrıca bu tezde Signal ve Wire uygulamalarının kullandığı ancak dökümantasyolarında belirtilmeyen parametreler açık kaynaklardan elde edilmiş ve eklenmiştir. Son olarak Signal Protokol'ünü kuantuma dayanıklı hale getirmek için kullanılan yöntemlerden bahsedilmiştir.

Anahtar Kelimeler: Kriptografi, Kriptografik Protokoller, Signal, Double Ratchet, Anlık Mesajlaşma Uygulamalarının Protokolleri, Anahtar Değişimi, Şifreleme, Kuantum, Kuantum Sonrası

To my family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Assoc. Dr. Ali Doğanaksoy, for providing guidance and feedback throughout this thesis. Furthermore, I would like to thank my co-supervisor Dr. Pınar Gürkan Balıkçıoğlu for her support and encouragement. I would not have made it through my master's degree without them.

I also would like to thank Prof. Dr. Murat Cenk, who did not spare his valuable opinions and help when I needed it.

I would like to express my very great appreciation to my parents and my sister for their love and prayers. Knowing that they are with me in every difficulty I face and feeling their endless support has always given me strength.

Finally, to my caring, loving and supportive husband, Muharrem, my deepest thanks for his patience throughout this process and for always encouraging me. My heartfelt thanks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

X3DH	Extended Triple Diffie Hellman
XEdDSA	The protocol to create and verify Edwards-curve Digital Signature Algorithm compatible signatures
VXEdDSA	Extends XEdDSA To Make It A Verifiable Random Function
EdDSA	Edwards Curve Digital Signature Algorithm
SHA	Secure Hash Algorithm
ECDH	Elliptic Curve Diffie Hellman
AD	Associated Data
AEAD	Authenticated Encryption with Associated Data
HKDF	Key Derivation Function based on HMAC Message Authentication Code
PRF	Pseudo Random Function
KDF	Key Derivation Function
MAC	Message Authentication Code
HMAC	Hash Based Message Authentication Code
SIV	Synthetic Initialization Vector
CBC	Cipher Block Chaining
AES	Advanced Encryption Standard
XMPP	Extensible Messaging and Presence Protocol
PGP	Pretty Good Privacy
OTR	Off-the-Record Messaging Protocol
IV	Initialization Vector
PKCS	Public Key Cryptography Standards
LIME	Linphone Instant Message Encryption

GCM	Galois/Counter Mode
IKM	Input Key Metarial
OKM	Output Key Metarial
KEM	Key Encapsulation Mechanism
SIDH	Supersingular Isogeny Diffie–Hellman Key Exchange
CSIDH	Commutative Supersingular Isogeny Diffie–Hellman Key Exchange

CHAPTER 1

INTRODUCTION

Although instant messaging is done over the internet today, the history of instant messaging is older than the internet and dates back to the 1960s. It was first used in operating systems with multiple users such as Compatible Time-Sharing System (CTSS) and Multiplexed Information and Computing Service (Multics)[24]. As so networks developed, protocols spread over the network. Some of the protocols are peer-to-peer protocols, while others are client to server protocols. As a result of the widespread use of the Internet, it has begun to reach more and more users. Both the increase in the number of users and the transmission of messages through public channels made it easier for third parties to intervene. Thus, problems such as privacy, message integrity, and authentication began to emerge.

The Signal Protocol was first published in 2010 as the Text Secure Protocol, whose predecessor was OTR (Off-the-record Messaging). Later, TextSecure v2 and TextSecure v3 were published and took the final form in 2016, taking the name Signal Protocol. Signal protocol provides authentication, confidentiality, asynchronicity, integrity, causality preservation, destination validation, participant consistency, forward and backward secrecy, message unlinkability, participation repudiation, and message repudiation[41]. Thanks to these features, it has shown that it is one of the most secure protocols by getting a full score of 7 out of 7 on the Electronic Frontier Foundation (EFF) secure messaging scorecard[6] in 2014. Today, applications such as Whatsapp [8], Facebook Messenger [2], Skype [7], which have the most users, also use the Signal Protocol.

1.1 Related Works

An analysis of the Signal Protocol was published by Ruhr Univercity Bochum researchers in 2014[25]. In this analysis, they focuson three main parts, key generation, key exchange and authenticated encryption, were analyzed and basic security claims were discussed. In addition, they have proven that the Signal TextSecure protocol can achieve its alleged security targets if key registration is assumed to be done securely. As a result, they found the Signal Protocol safe.

In October 2016, researchers from Oxford University, Queensland University of Technology and McMaster University published an formally the Signal protocol's analysis[19]. In this study, X3DH (Extended Triple Diffie-Hellman) key exchange and Double Ratchet Protocols were analyzed and it was concluded that the Signal protocol is cryptographically sound.

Of course, the studies we presented showed that the Signal Protocol is safe with calculations made with classical computers. However, in 1997 Shor's algorithm [39] showed that public key algorithms are unstable against quantum computers. Because of the cost, encryption operations are not done with public key algorithms, while key exchanges are made with public key algorithms. Therefore, first of all, studies have begun on quantum resistant algorithms that can replace public key algorithms. One of them is the NIST Post-Quantum Standardization Process[3], which the National Institute of Standards and Technology (NIST) initiated in 2016. NIST aims to standardize quantum resistant public key encryption and digital signature algorithms at the end of this process. While a total of 69 applications were selected for NIST for the round 1, the process is currently in the round 4. The first algorithm to standardize has been identified, and in round 4, four candidate will be considered for standardization.

1.2 Our Contributions

In this thesis study, Signal protocol and for those who want to use this protocol in their own applications in the future, how to choose the parameters are explained.

The algorithms used in key exchange and encryption of Linphone, Wire, Xabber and

Riot.im that developed using Signal Protocol are explained. Similar and different features in these applications are indicated. Thus, those who want to develop applications will have more than one sample in their hands for algorithm and parameter selection.

Signal and Wire protocols have specified the algorithms they use in their documentation, but not the parameters. These parameters were found from their open source codes and brought to this study.

By compiling the studies on post quantum algorithms developed for key exchange in the Signal protocol, we discussed the advantages and disadvantages of these algorithms. By comparing the Diffie-Hellman security features, we decided which one is more suitable for the Signal Protocol.

1.3 Organizations

The rest of this thesis is as follows:

In Chapter 2, all notations and some cryptographic primitives are used in protocols are described.

In Chapter 3, Signal protocol is reported and for those who want to use this protocol, how to choose parameters and algorithms is explained.

In Chapter 4, the algorithms and parameters used by the Signal protocol and the protocols of other applications based on the Signal protocol are examined.

In Chapter 5, the algorithms and parameters for key exchange in post quantum world are suggested for Signal and Signal based protocols are mentioned and algorithms compared for Diffie-Hellman security features.

In Chapter 6, the thesis is summarized and possible future works are mentioned.

CHAPTER 2

PRELIMINARIES

2.1 Notation

The list of common notations in this thesis is given below.

- DR: Double Ratchet
- SK: Shared secret key
- IK: Identity key
- SPK: Signed prekey
- OPK: One-time prekey
- EK: Ephemeral key
- AD: Associated Data
- RK: Root key
- CK: Chain key
- MK: Message key
- $ECDH_{out}$: The output of ECDH
- AES_KEY: The key is used in AES to encrypt the message
- $HMAC\_KEY$: The key is used in HMAC for authentication
- AES_IV: The initial value is used in AES when encrypt the message

In this thesis, the term *client* is represented as Alice or Bob. The client is the one side of the end-to-end encrypted communication.

2.2 Definitions

There are some definitions of cryptographic concepts used in this thesis.

2.2.1 Cryptographic Hash Functions

A hash function is a mathematical function which takes arbitrary-length input and returns the fix-length output. Let H be a hash function and A be a input message then $H : A \to B$, length of B is fix.

A hash function is a cryptographic hash function if it satisfies these three conditions.:

- **One-way function:** In practice, it is infeasible to reverse computation and getting input message.
- **Deterministic:** For given the same messages m_1 and m_2, their hashes are the same $H(m_1) = H(m_2)$.
- **Collision resistance:** For given different messages m_1 and m_2, it should be $H(m_1) \neq H(m_2)$.

Cryptographic hash functions are used in many areas of cryptography such as digital signature, authentication, and message integrity etc.

2.2.2 HMAC (Message Authentication Code Mechanism Based on Cryptographic Hash Function)

HMAC[29] is a Message Authentication Code (MAC) mechanism based on cryptographic hash function with a secret key. HMAC satisfies both data integrity and authentication. The length of HMAC output is depending on cryptographic hash function used in the system and it is fix. The definition of HMAC in [29] as follow;

$$\begin{aligned} HMAC(K,m) &= H\Big(K' \oplus opad || H\big((K' \oplus ipad)||m\big)\Big) \\ K' &= \begin{cases} H(K), & \text{K is larger than block size} \\ K, & \text{otherwise} \end{cases} \end{aligned} \tag{2.2.1}$$

In the definition;

- H is a hash function,
- m is the message,
- K is the secret key,
- K' is the derived key from secret key,
- *opad* is the block-sized outer padding, consisting of repeated bytes valued $0x5C$
- *ipad* is the block-sized inner padding, consisting of repeated bytes valued $0x36$

2.2.3 KDF (Key Derivation Function)

KDF is a cryptographic algorithm that derives cryptographically strong secret key from a secret value such as a main key or a password.

2.2.4 HKDF (HMAC-based Extract-and-Expand Key Derivation Function)

HKDF[30] is a simple key derivation function based on HMAC. HKDF, firstly, takes IKM (Input Key Material) and extract a fixed-length pseudorandom key. Then, it expands the key into several additional pseudorandom keys.

2.2.5 AEAD (Authenticated Encryption with Associated Data)

AEAD[33] is an authenticated encryption format. AEAD has the ability to check the integrity and authenticity of some Associated Data (AD) in addition to authenticated encryption.

2.3 Post-Quantum Cryptographic Algorithms

This section contains some definitions used to make the Signal protocol post-quantum robust.

2.3.1 KEM(Key Encapsulation Mechanism)

KEM is used to send shared secret key. In the quantum world, the public key algorithms are not safe. Therefore, KEM is used instead of them. We can say that KEM consists of three algorithms in general:

- Generate public-private key pair
- Encapsulate takes the public key as input and outputs the shared secret and encapsulates this secret key
- Decapsulate takes the encapsulated secret key and the private key as input and outputs the shared secret key

2.3.2 SIDH(Supersingular Isogeny Diffie Hellman)

SIDH [28] is a post-quantum crytographic algorithm which is used for key exchange. There are many complex isogeny calculations in SIDH, but people who have a grasp of Diffie-Hellman will understand SIDH easily. Let, firstly, generate public parameters:

- A prime $p = {l_A}^{e_A} \cdot {l_B}^{e_B} \cdot f \pm 1$, where l_A and l_B are different small primes , e_A and e_B are large exponents and f is small cofactor

- A supersingular elliptic curve E over $\mathbb{F}_{p^2}$
- Fixed elliptic points P_A, Q_A, P_B, Q_B on E where P_A and Q_A are in order ${l_A}^{e_A}$, P_B and Q_B are in order ${l_B}^{e_B}$

In the key exchange, below steps are following:

- Alice,
  - Generates two random integers m_A, $n_A < (l_A)^{e_A}$,
  - Generates $R_A := m_A \cdot (P_A) + n_A \cdot (Q_A)$,
  - Creates an isogeny mapping $\phi_A : E \rightarrow E_A$ and curve E_A isogenous E using R_A and Velu's formulas [34],
  - Computes images $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$.
- Bob,
  - Generates two random integers m_B, $n_B < (l_B)^{e_B}$,
  - Generates $R_B := m_B \cdot (P_B) + n_B \cdot (Q_B)$,
  - Creates an isogeny mapping $\phi_B : E \rightarrow E_B$ and curve E_B isogenous E using R_B and Velu's formulas,
  - Computes images $\{\phi_B(P_A), \phi_B(Q_A)\} \subset E_B$.
- Alice sends to Bob E_A, $\phi_A(P_B)$ and $\phi_A(Q_B)$.
- Bob also sends to Alice E_B, $\phi_B(P_A)$ and $\phi_B(Q_A)$.
- Alice has m_A, n_A, $\phi_B(P_A)$, and $\phi_B(Q_A)$. Then,
  - Computes $S_{BA} := m_A(\phi_B(P_A)) + n_A(\phi_B(Q_A))$,
  - Creates an isogeny mapping ψ_{BA} using S_{BA} and Velu's formulas,
  - Creates an elliptic curve E_{BA} isogenous to E using ψ_{BA},
  - Finally computes $K := j - invariant(j_{BA})$ of the curve E_{BA}.
- Bob also has m_B, n_B, $\phi_A(P_B)$, and $\phi_A(Q_B)$. Then,

- Computes $S_{AB} := m_B(\phi_A(P_B)) + n_B(\phi_A(Q_B))$,
- Creates an isogeny mapping ψ_{AB} using S_{AB} and Velu's formulas,
- Creates an elliptic curve E_{AB} isogenous to E using ψ_{AB},
- Finally computes $K := j - invariant(j_{AB})$ of the curve E_{AB}.

A function of K is the secret key.

2.3.3 CSIDH(Commutative Supersingular Isogeny Diffie-Hellman)

CSIDH [18] is a cryptographic primitive that is used instead of Diffie-Hellman in the quantum world. CSIDH is a commutative action based on supersingular elliptic curve. Keys can be reused in CSIDH because it enables static-static key exchange.

CHAPTER 3

SIGNAL PROTOCOL

The Signal protocol is a cryptographic protocol used to provides end-to-end encryption in messaging applications. In the early instant messaging applications, end-to-end encryption was not provided, only the traffic between the server and the client was encrypted. Signal protocol is based on off-the-record messaging protocol [21].

Signal protocol has main four part:

- The X3DH Key Agreement Protocol [31]
- The XEdDSA and VXEdDSA Signature Schemes [37]
- The Double Ratchet Algorithm [38]
- The Sesame Algorithm: Session Management for Asynchronous Message Encryption [32]

The first part explains how to obtain a shared secret key between two parties. The second part explains how to create and verify EdDSA (Edwards-curve Digital Signature Algorithm) compliant signatures using public and private key. The third part explains how encrypted messages are exchanged based on a shared secret key. In fact, applications based on the signal protocol specifically use this part. The final section explains how to manage message encryption sessions in an asynchronous and multi-device environment. In this thesis we focus on key exchange, message encryption and exchange of encrypted messages.

3.1 Key Exchange

The X3DH protocol creates a shared secret key between the two parties, which is mutually authenticated based on the their public keys. The protocol contains three parties: Alice, Bob and a server. When Alice wants to send a message to Bob, Bob might be offline but he had sent his key information to the server. Alice uses this key information to generate a secret key and communicate with Bob.

The parameters used by this protocol are shown in Table 3.1.

Table 3.1: X3DH parameters recommended for protocols that will use the Signal protocol

Name	**Definitions**
curve	X25519 or X448
hash	A 256-bit or 512-bit hash function
info	An ASCII string

For example, an application can choose *X448*, *SHA-512*, and *"MyInfo"* as parameters.

X3DH protocol uses some elliptic curve public keys. For example Alice sends a message to Bob. The keys required in this case are;

- Identity keys IK_A for Alice and IK_B for Bob
- A signed prekey SPK_B for Bob
- A set of one-time prekeys OPK_B for Bob
- An ephemeral key EK_A for Alice

When the protocol ends, each party has a 32-byte secret key, SK, to communicate.

X3DH has three stages:

1. Bob's identity key and prekeys are uploaded to a server,
2. Alice receives a "prekey bundle" containing Bob's keys from the server and uses it to initiate the conversation.

3. Bob receives Alice's first message and processes it.

Firstly, Bob sends a set of public keys which contains:

- Bob's identity key IK_B,
- Bob's signed prekey SPK_B,
- Bob's prekey signature $Sig(IK_B, Encode(SPK_B))$,
- A set of Bob's one-time prekeys $(OPK_{B_1}, OPK_{B_2}, OPK_{B_3}, ...)$

to server.

Except for the identity key, Bob must again upload other keys several times. As the one-time prekeys are updated when decreases on the server, other keys are updated once a week or monthly.

To send a message to Bob, Alice first contact to the server and pull a prekey bundle which contains Bob's keys; IK_B, SPK_B, $Sig(IK_B, Encode(SPK_B))$, and optionally OPK_B. After a one-time prekey is used, the server deletes the one-time prekey. If any one-time prekey does not exist in the server the bundle is sent without a one-time prekey.

Alice verifies the prekey signature. If verification is failed Alice annul the protocol, otherwise generates an ephemeral key EK_A and calculate:

$$ECDH_1 = ECDH(IK_A, SPK_B) \tag{3.1.1}$$

$$ECDH_2 = ECDH(EK_A, IK_B) \tag{3.1.2}$$

$$ECDH_3 = ECDH(EK_A, SPK_B) \tag{3.1.3}$$

$$ECDH_4 = ECDH(EK_A, OPK_B) \tag{3.1.4}$$

$$SK = HKDF(ECDH_1||ECDH_2||ECDH_3||ECDH_4) \tag{3.1.5}$$

"ECDH" represents output of an Elliptic Curve Diffie-Hellman function, *"HKDF"* represents 32 bytes of output from *HMAC Key Derivation Function*.

If the bundle does not have a OPK_B then $ECDH_4$ is not calculated.

Alice deletes her ephemeral key and ECDH outputs after SK is calculated.

Lastly, Alice computes an associated data byte sequence AD that includes identity information for Alice and Bob:

$$AD = Encode(IK_A)||Encode(IK_B) \tag{3.1.6}$$

For calculation of AD, parties' certificates, usernames, or other identifying information can be used unlike identity keys.

After all the computation, Alice sends the first message to Bob. It contains;

- Alice's identity key IK_A,
- Alice's ephemeral key EK_A,
- Identifiers stating which of Bob's prekeys Alice used
- An initial ciphertext is encrypted with some ***AEAD*** using AD and using an encryption key which is either SK or the output from some cryptographic pseudo random function keyed by SK.

After the first message is sent, communication can continue through the SK or keys derived from SK.

After Bob receives the first message, Bob gets IK_A and EK_A with the message. Using his own private keys corresponding prekeys which Alice used, derive SK by repeating ECDH and HKDF algorithms. Bob also computes AD as reported and tries to decrypt Alice's initial message. If decryption is failed, Bob annul protocol. If the initial message is decrypted successfully then Bob deletes any one-time prekey which was used. This deletion provides forward secrecy.

3.1.1 Double Ratchet

The double ratchet algorithm is used to encrypt messages and send and receive these encrypted messages. Actually, double ratchet can be considered post-X3DH. Because the outputs of X3DH become inputs of the double ratchet.

- The SK becomes the input to initiate the double ratchet.
- The AD is used in double ratchet encryption and decryption as input.
- The SPK_B becomes Bob's first ratchet key pair to initiate double ratchet.

In every double ratchet, new keys are derived thus and so earlier keys can not be computed using later keys. Also, in every double ratchet, parties send Diffie-Hellman public keys. Parties use the result of Diffie-Hellman to derive new keys so that later keys can not be computed using earlier keys.

The double ratchet has mainly two parts: symmetric-key ratchet and Diffie-Hellman ratchet. Symmetric-key ratchet is used to obtain the message key when a message is sent or received. Diffie-Hellman ratchet is used to obtain new chain keys when a new ratchet public key is received. The core concept of these two parts is *Key Derivation Function, KDF*.

KDF is a cryptographic function which takes a secret and random KDF key and some input data then gives an output data. The term KDF chain is used when some part of output of a KDF is used as output key and the rest part of the output is used as KDF key with another input data for another KDF. Figure 3.1 shows a KDF chain.

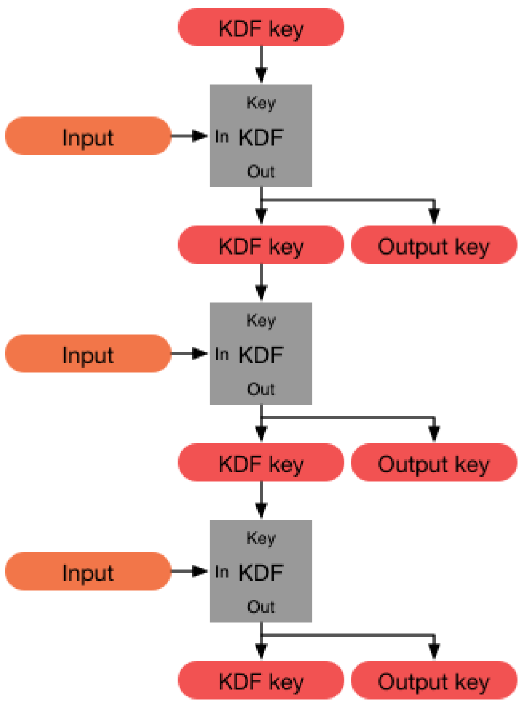


Figure 3.1: KDF Chain [38]

A KDF chain has resilience, forward security and break-in recovery [38].

- Resilience: Whether or not the third party knows about KDF keys, they see the output key as random.
- Forward security: The third party who learns the KDF key at any time sees the output keys from the past as random.
- Break-in recovery: The third party who learns the KDF key at any time will see the future output keys as random, provided that the future inputs have added enough entropy.

Symmetric-key Ratchet: A message key is required to encrypt each message. KDF chains' outputs are called message key, *MK*, and chain key, *CK*, which are the KDF keys for these chains. In symmetric-key ratchet KDF takes constant data as input data. These chains ensure that each message is encrypted with a unique key. After encryption, these keys can be deleted. The calculation of the next chain key and the message key is done in a single ratchet step as shown in the Figure 3.2. In the symmetric-key ratchet step HMAC is used to get the next 32-byte CK and 32-byte MK.

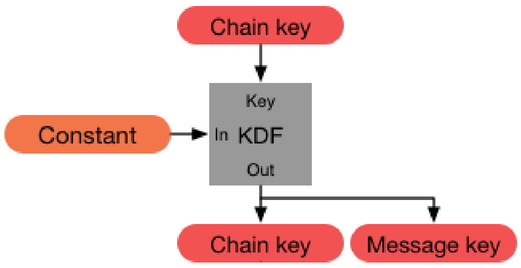


Figure 3.2: Single Ratchet [38]

The Signal protocol recommends to use *HMACSHA-256* or *HMACSHA-512* with CK as the HMAC key. If calculating the message key MK it takes $0x01$ as constant, if calculating the chain key CK it takes $0x02$ as constant.

If the chain key is stolen then all future message keys can be computed. Therefore symmetric-key ratchet and Diffie-Hellman ratchet are combined in the Double Ratchet algorithm.

Diffie-Hellman Ratchet: Diffie-Hellman ratchet obtains new chain keys using Diffie-Hellman outputs. In each ratchet so that, in case the chain key is stolen, the attacker can not calculate future message keys.

In the Diffie-Hellman ratchet algorithm, each party generates a ratchet key pair which is a *Diffie-Hellman key pair* in every ratchet. Every message has the sender's current ratchet public key in the header part. When Alice sends a new ratchet public key to Bob, a Diffie-Hellman ratchet step is applied and the current key pair is updated with a new one.

In this ratchet, one party initializes the ratchet with the other party's public key. For example, as shown in Figure 3.3, Alice takes Bob's ratchet public key, then calculates Diffie-Hellman outputs with her ratchet private key.

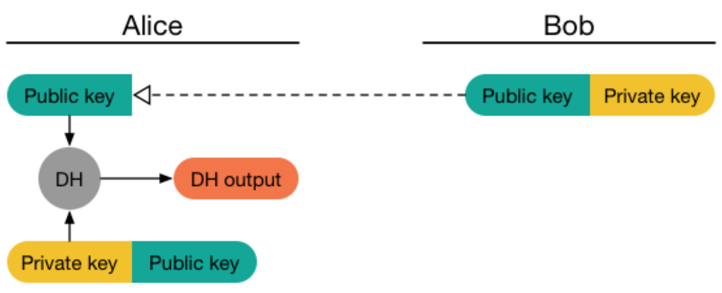


Figure 3.3: Diffie-Hellman Key Exchange I [38]

In Figure 3.4, Bob takes Alice's message, which contains Alice's ratchet public key in the header part, he performs Diffie-Hellman step with his ratchet private key and Alice's ratchet public key and gets the same Diffie-Hellman output as Alice gets. After that Bob generates a new ratchet key pair and calculates a new Diffie-Hellman output to send a message.

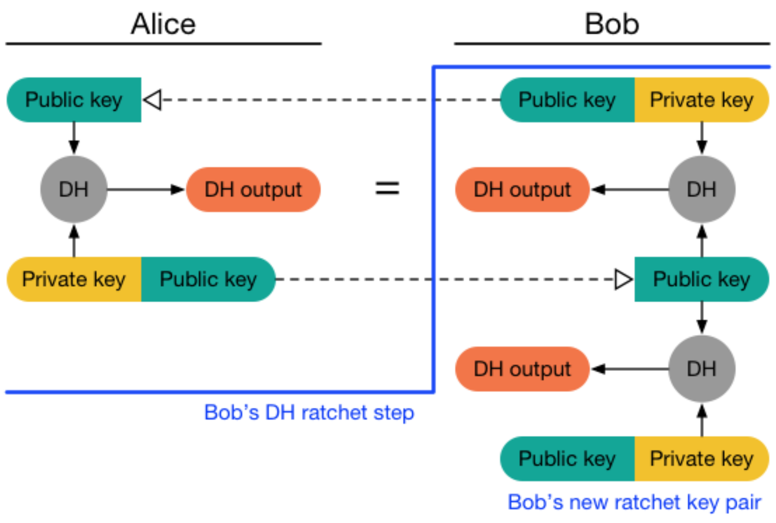


Figure 3.4: Diffie-Hellman Key Exchange II [38]

After Alice takes Bob's message, in Figure 3.5, she computes two Diffie-Hellman outputs like Bob. One of the outputs is exactly the same output with Bob's latest output and the other one is a new ratchet key pair. This process continues like pingpong behavior.

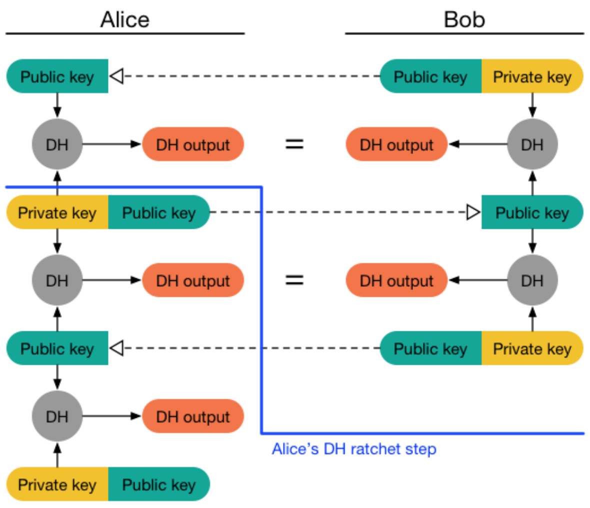


Figure 3.5: Diffie-Hellman Key Exchange III [38]

All computed Diffie-Hellman outputs are used to generate sending and receiving chain keys. The sending key of Alice is the same as the receiving key of Bob as shown in Figure 3.6.

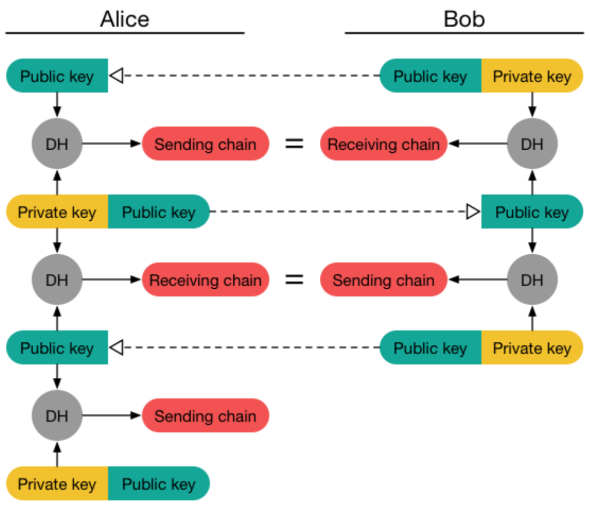


Figure 3.6: Sending and Receiving Chain Keys [38]

The above diagram is roughly. The sending and receiving keys are not taken as the same Diffie-Hellman outputs. After compute Diffie-Hellman output, this value is used as input data in a KDF with root key *RK* shown in Figure 3.7. The first 32-byte of KDF output is used as the root key for the next KDF and the rest 32-byte of KDF output is used as a sending or receiving chain key. In Diffie Hellman Ratchet, HKDF is used to produce the next root key RK and the sendinlg/receiving chain key CK.

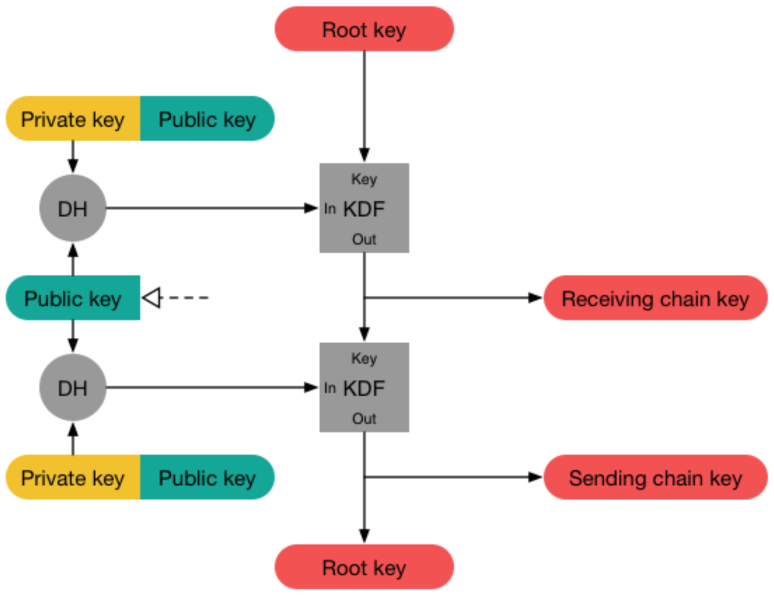


Figure 3.7: Diffie-Hellman Ratchet [38]

The Signal protocol recommends using *HKDF-SHA256* or *HKDF-SHA512* with inputs the root key RK, the output of Diffie-Hellman and info value.

Double ratchet is a combination of symmetric-key ratchet and Diffie-Hellman ratchet.

In Figure 3.8, Alice computes Diffie-Hellman output with her ratchet private key and Bob's ratchet public key and also generates her new ratchet key pair in the first ratchet. Then this Diffie-Hellman output is used as input data in a KDF with a secret root key so Alice obtains a new RK, and sending CK.

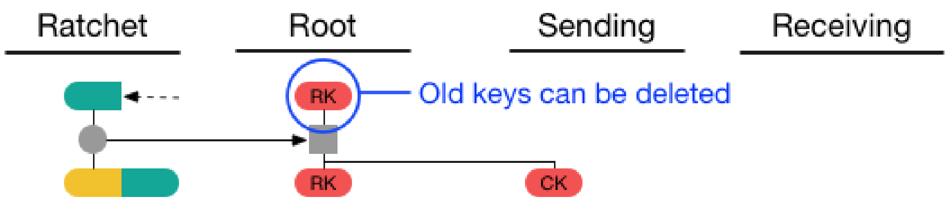


Figure 3.8: Old Keys Deletion [38]

After the next RK and CK, a symmetric-key ratchet is applied to the CK to send message A1, the message key $A1$ (will be labeled with the message) is produced in Figure 3.9. After the message A1 is sent, keeping the new chain key while deleting the old chain key and the message key.

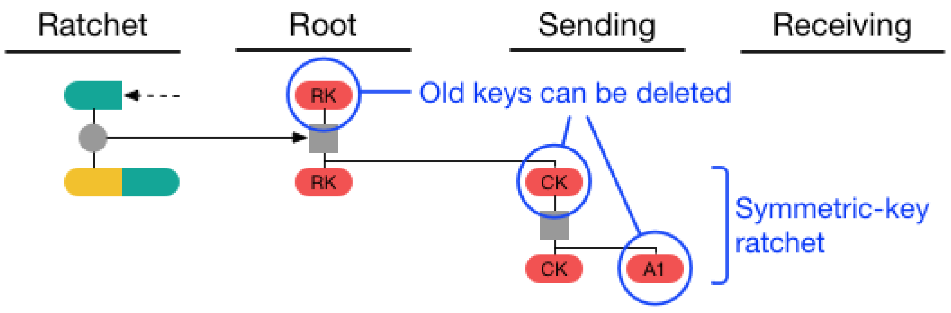


Figure 3.9: Symmetric Key Ratchet [38]

Until Alice takes response from Bob it continues. After Alice takes the message B1 from Bob, Alice again applies Diffie-Hellman ratchet because of taking a new ratchet public key from Bob (Bob's public keys are showed as the message name) and also again applies a symmetric-key ratchet to produce the next MK as shown in Figure 3.10.

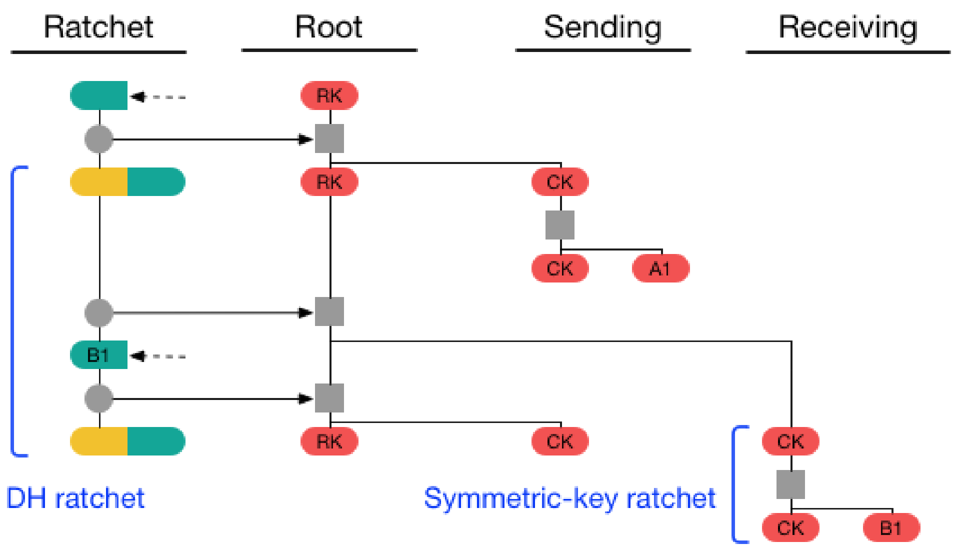


Figure 3.10: Key computation for the next message [38]

3.2 Encryption

In the Signal protocol, AEAD encryption scheme based on either SIV (Synthetic Initialization Vector) or a composition of CBC (Cipher Block Chaining) with HMAC is recommended. Below is a recommended encryption example based on CBC mode with HMAC:

1. To produce 80 bytes output use *HKDF-SHA256*. The inputs are the salt of HKDF is zero with length is hash's output length, the key is the message key MK, and info.
2. The first 32-byte of output of HKDF is encryption key, the second 32-byte is authentication key and last 16-byte is IV.
3. The message is encrypted with AES256 in CBC mode with PKCS#7 (Public-Key Cryptography Standards) padding.
4. Lastly, the output of *HMAC-SHA256* which is calculated with authentication key and AD, is appended to the ciphertext.

CHAPTER 4

SIGNAL AND SIGNAL BASED PROTOCOLS

4.1 SIGNAL

Signal protocol documentation provides an overview for those who will use the Signal protocol in their applications. However, it did not give any information about which parameters or which algorithms it used. Therefore, information similar to these has been obtained from Signal's source code.

4.1.1 Key Exchange

The parameters used by Signal protocol in X3DH are shown in Table 4.1.

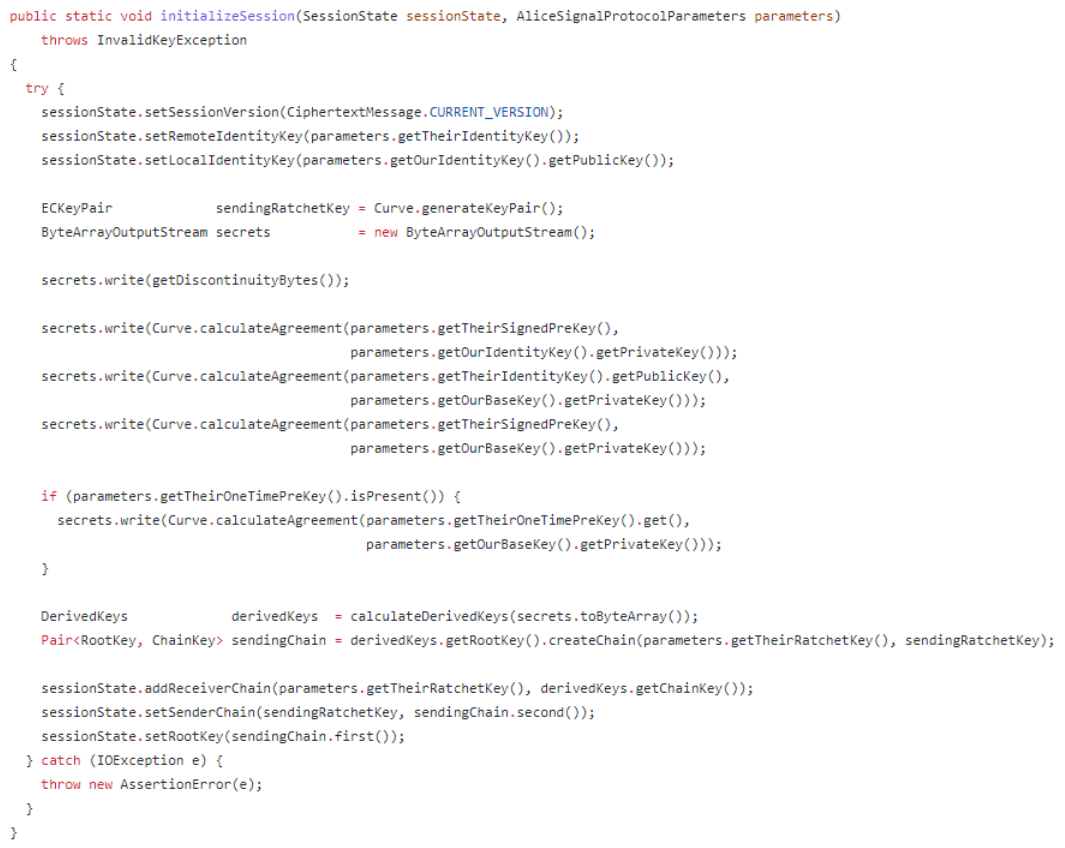
Table 4.1: X3DH parameters for Signal protocol

Name	**Definitions**
curve	CurveX25519
hash	SHA256
info	"WhisperText"

Firstly, SK is computed in the function *initializeSession*;

$$\begin{aligned} SK = HKDF(ECDH(IK_A, SPK_B)||ECDH(EK_A, IK_B)|| \\ ECDH(EK_A, SPK_B)||ECDH(EK_A, OPK_B)) \end{aligned} \tag{4.1.1}$$

The Figure 4.1 is taken from source code of Signal and shows *initializeSession* function.

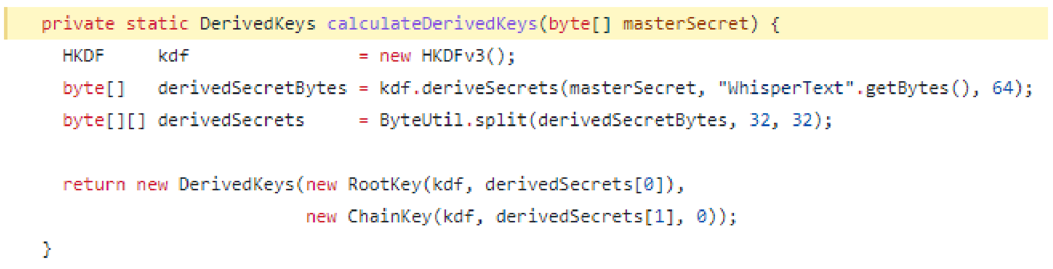


```java
public static void initializeSession(SessionState sessionState, AliceSignalProtocolParameters parameters)
    throws InvalidKeyException
{
  try {
    sessionState.setSessionVersion(CiphertextMessage.CURRENT_VERSION);
    sessionState.setRemoteIdentityKey(parameters.getTheirIdentityKey());
    sessionState.setLocalIdentityKey(parameters.getOurIdentityKey().getPublicKey());

    ECKeyPair             sendingRatchetKey = Curve.generateKeyPair();
    ByteArrayOutputStream secrets           = new ByteArrayOutputStream();

    secrets.write(getDiscontinuityBytes());

    secrets.write(Curve.calculateAgreement(parameters.getTheirSignedPreKey(),
                                           parameters.getOurIdentityKey().getPrivateKey()));
    secrets.write(Curve.calculateAgreement(parameters.getTheirIdentityKey().getPublicKey(),
                                           parameters.getOurBaseKey().getPrivateKey()));
    secrets.write(Curve.calculateAgreement(parameters.getTheirSignedPreKey(),
                                           parameters.getOurBaseKey().getPrivateKey()));

    if (parameters.getTheirOneTimePreKey().isPresent()) {
      secrets.write(Curve.calculateAgreement(parameters.getTheirOneTimePreKey().get(),
                                             parameters.getOurBaseKey().getPrivateKey()));
    }

    DerivedKeys             derivedKeys  = calculateDerivedKeys(secrets.toByteArray());
    Pair<RootKey, ChainKey> sendingChain = derivedKeys.getRootKey().createChain(parameters.getTheirRatchetKey(), sendingRatchetKey);

    sessionState.addReceiverChain(parameters.getTheirRatchetKey(), derivedKeys.getChainKey());
    sessionState.setSenderChain(sendingRatchetKey, sendingChain.second());
    sessionState.setRootKey(sendingChain.first());
  } catch (IOException e) {
    throw new AssertionError(e);
  }
}
```

Figure 4.1: Computation of shared secret key

Then the function, is called *calculateDerivedKeys* shown in Figure 4.2, takes SK and related parameters and gives the first root and chain keys;

$$RK_0||CK_{0,0} = HKDF(SK, \textit{"WhisperText"}, 64) \qquad (4.1.2)$$



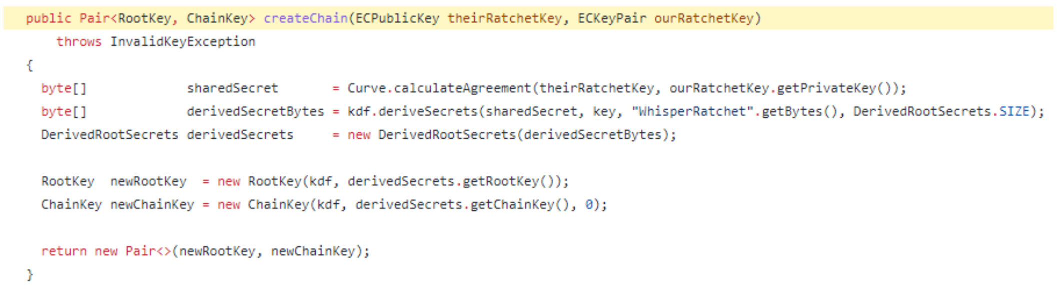
```java
private static DerivedKeys calculateDerivedKeys(byte[] masterSecret) {
  HKDF     kdf                = new HKDFv3();
  byte[]   derivedSecretBytes = kdf.deriveSecrets(masterSecret, "WhisperText".getBytes(), 64);
  byte[][] derivedSecrets     = ByteUtil.split(derivedSecretBytes, 32, 32);

  return new DerivedKeys(new RootKey(kdf, derivedSecrets[0]),
                         new ChainKey(kdf, derivedSecrets[1], 0));
}
```

Figure 4.2: Computation of the first root key and chain key

4.1.1.1 Double Ratchet

After computing the first root key and chain key, in each ratchet, the function *createChain* computes the new root keys and chain keys with info *"WhisperRatchet"* as shown in Figure 4.3.

$$RK_i||CK_{i,0} = HKDF(ECDH_{out}, RK_{i-1}, \text{"\textit{WhisperRatchet}"}, 64) \tag{4.1.3}$$



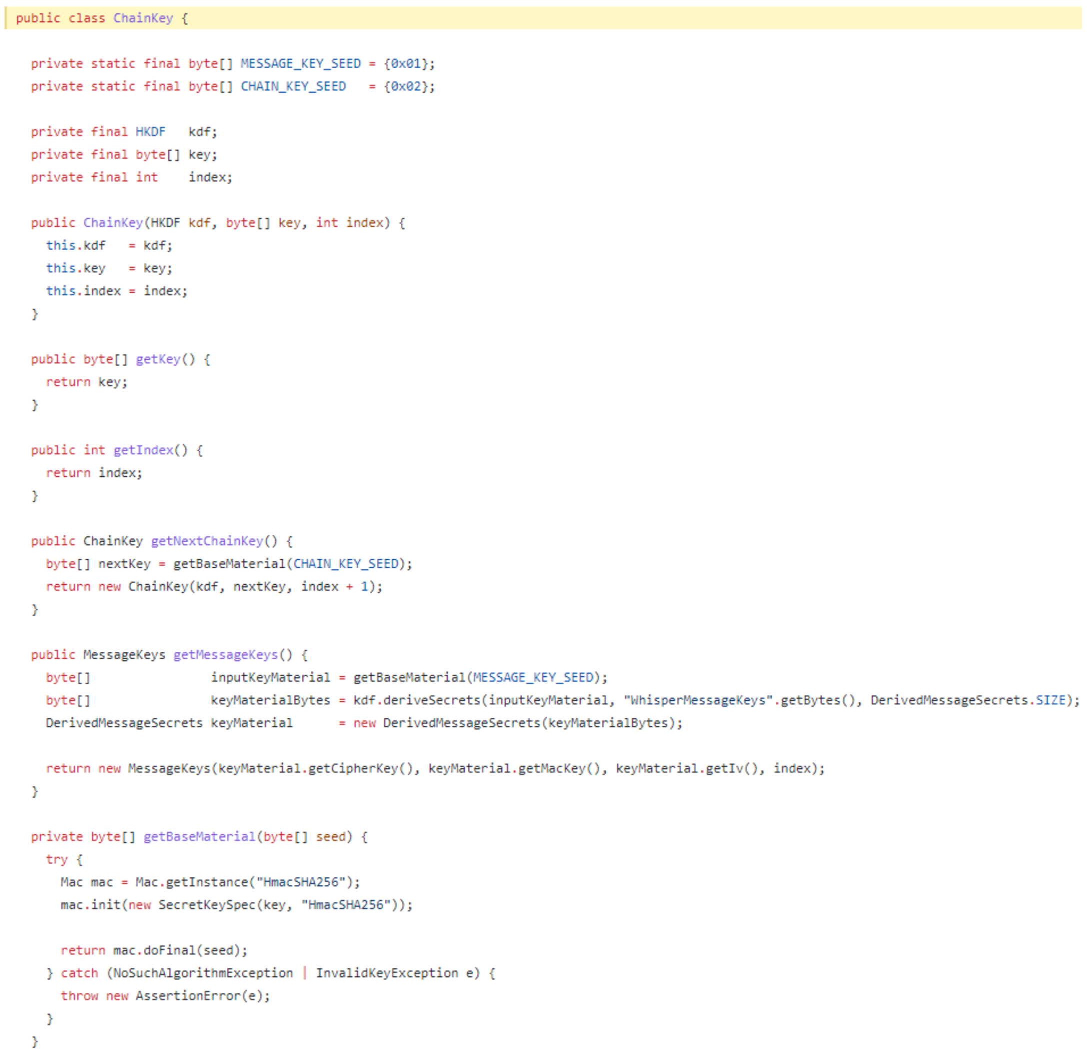
```java
public Pair<RootKey, ChainKey> createChain(ECPublicKey theirRatchetKey, ECKeyPair ourRatchetKey)
    throws InvalidKeyException
{
  byte[]              sharedSecret       = Curve.calculateAgreement(theirRatchetKey, ourRatchetKey.getPrivateKey());
  byte[]              derivedSecretBytes = kdf.deriveSecrets(sharedSecret, key, "WhisperRatchet".getBytes(), DerivedRootSecrets.SIZE);
  DerivedRootSecrets derivedSecrets     = new DerivedRootSecrets(derivedSecretBytes);

  RootKey  newRootKey  = new RootKey(kdf, derivedSecrets.getRootKey());
  ChainKey newChainKey = new ChainKey(kdf, derivedSecrets.getChainKey(), 0);

  return new Pair<>(newRootKey, newChainKey);
}
```

Figure 4.3: Computation of the next root key and chain key

Then the function *ChainKey* shown in Figurr 4.4, takes CK and computes the new chain key with constant $0x02$ and message key with constant $0x01$.

$$CK = HMAC(CK, 0x02) \tag{4.1.4}$$

$$MK = HMAC(CK, 0x01) \tag{4.1.5}$$



```
public class ChainKey {

  private static final byte[] MESSAGE_KEY_SEED = {0x01};
  private static final byte[] CHAIN_KEY_SEED   = {0x02};

  private final HKDF   kdf;
  private final byte[] key;
  private final int    index;

  public ChainKey(HKDF kdf, byte[] key, int index) {
    this.kdf   = kdf;
    this.key   = key;
    this.index = index;
  }

  public byte[] getKey() {
    return key;
  }

  public int getIndex() {
    return index;
  }

  public ChainKey getNextChainKey() {
    byte[] nextKey = getBaseMaterial(CHAIN_KEY_SEED);
    return new ChainKey(kdf, nextKey, index + 1);
  }

  public MessageKeys getMessageKeys() {
    byte[]                inputKeyMaterial = getBaseMaterial(MESSAGE_KEY_SEED);
    byte[]                keyMaterialBytes = kdf.deriveSecrets(inputKeyMaterial, "WhisperMessageKeys".getBytes(), DerivedMessageSecrets.SIZE);
    DerivedMessageSecrets keyMaterial      = new DerivedMessageSecrets(keyMaterialBytes);

    return new MessageKeys(keyMaterial.getCipherKey(), keyMaterial.getMacKey(), keyMaterial.getIv(), index);
  }

  private byte[] getBaseMaterial(byte[] seed) {
    try {
      Mac mac = Mac.getInstance("HmacSHA256");
      mac.init(new SecretKeySpec(key, "HmacSHA256"));

      return mac.doFinal(seed);
    } catch (NoSuchAlgorithmException | InvalidKeyException e) {
      throw new AssertionError(e);
    }
  }
```

Figure 4.4: Computation of the next chain key and the message key

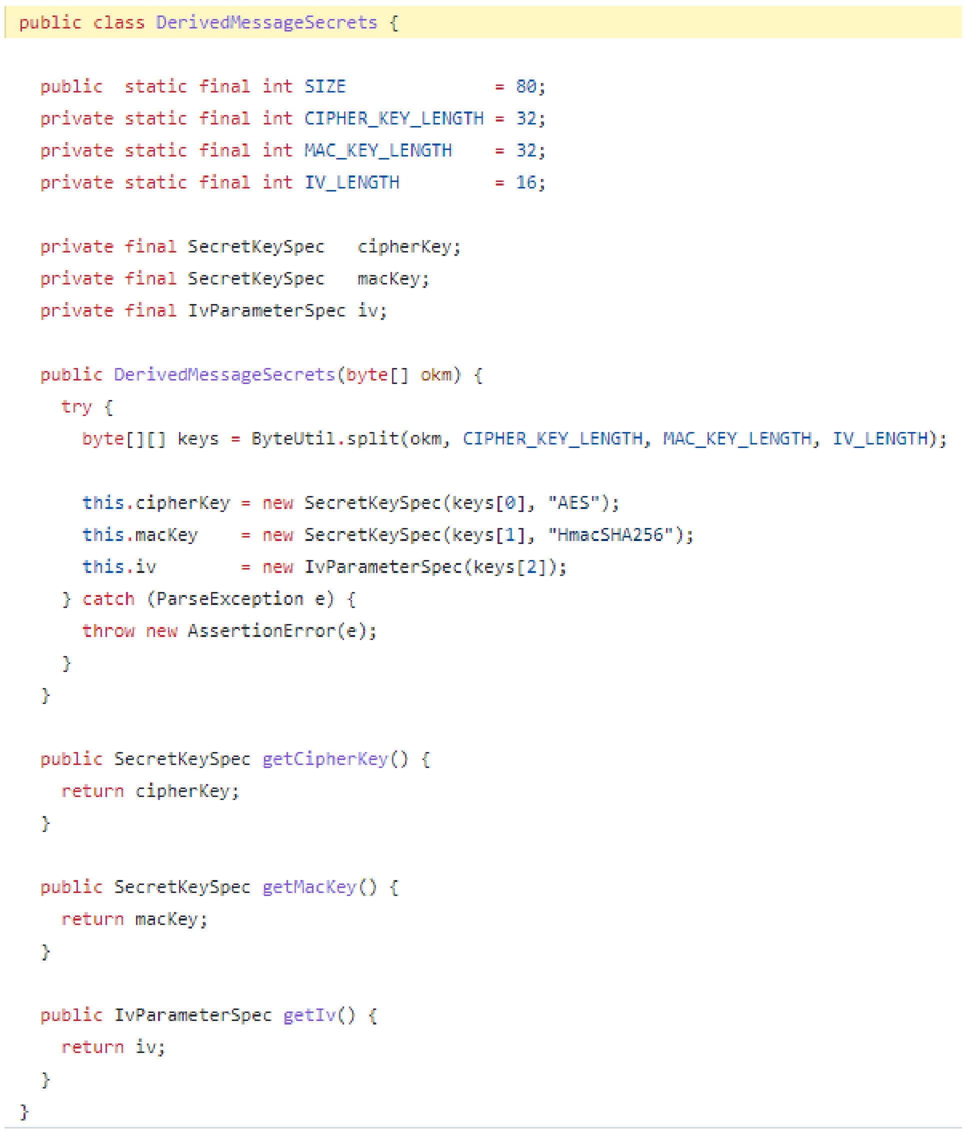
4.1.2 Encryption

As seen in the code block above, message key which is computed goes to *deriveSecrets* function with the info *"WhisperMessageKey"* as shown in Figure 4.4, then returns a byte array which has length 80 byte.

The first 32 byte of the array is mentioned as *CipherKey* in the source code is AES_KEY to use in AES, the second 32 byte is mentioned *MacKey* in the source code is $HMAC\_KEY$ to use in HMAC and the last 16 byte is mentioned *Iv* is AES_IV to use in AES.

$$
\begin{aligned}
&AES\_KEY_{i,j} || HMAC\_KEY_{i,j} || AES\_IV_{i,j} \\
&= HKDF(0, MK_{i,j}, \text{“\textit{WhisperMessageKey}"}, 80)
\end{aligned}
\tag{4.1.6}
$$

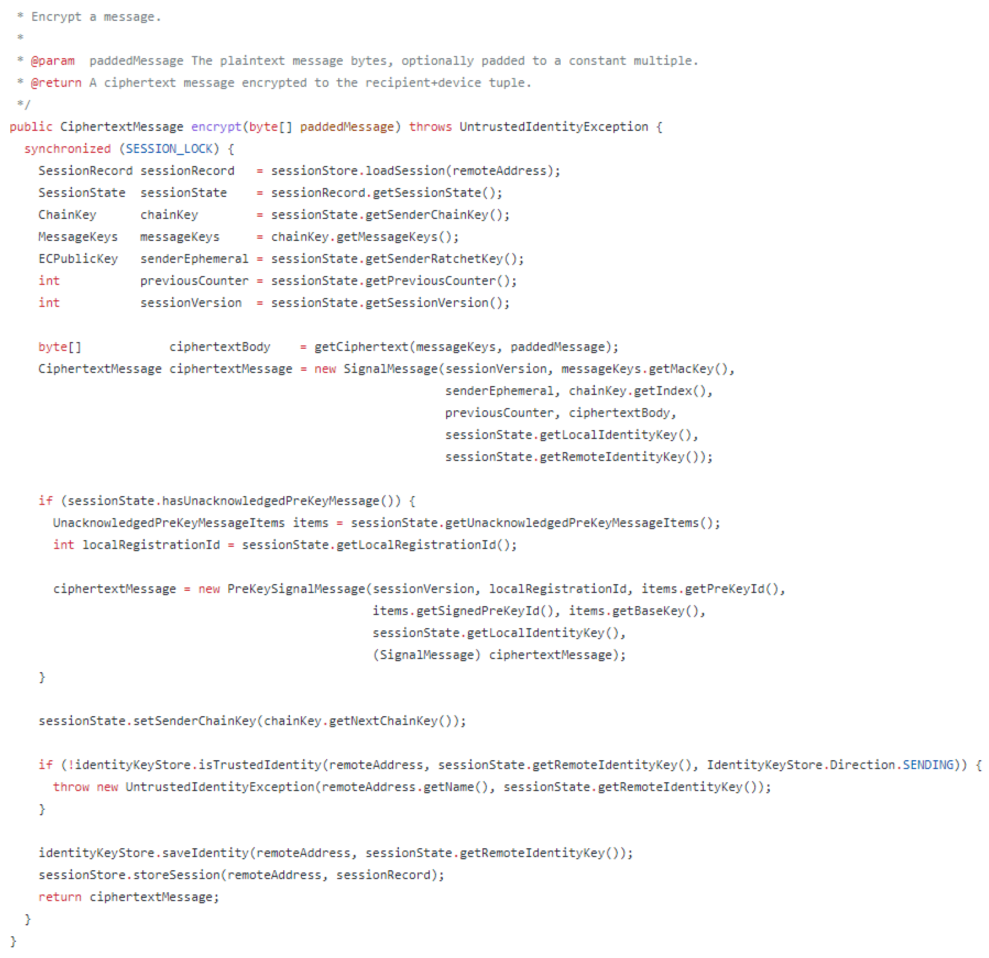
The *DerivedMessageSecrets* funcion in Figure 4.5 does this calculation.



```
public class DerivedMessageSecrets {

  public  static final int SIZE              = 80;
  private static final int CIPHER_KEY_LENGTH = 32;
  private static final int MAC_KEY_LENGTH    = 32;
  private static final int IV_LENGTH         = 16;

  private final SecretKeySpec   cipherKey;
  private final SecretKeySpec   macKey;
  private final IvParameterSpec iv;

  public DerivedMessageSecrets(byte[] okm) {
    try {
      byte[][] keys = ByteUtil.split(okm, CIPHER_KEY_LENGTH, MAC_KEY_LENGTH, IV_LENGTH);

      this.cipherKey = new SecretKeySpec(keys[0], "AES");
      this.macKey    = new SecretKeySpec(keys[1], "HmacSHA256");
      this.iv        = new IvParameterSpec(keys[2]);
    } catch (ParseException e) {
      throw new AssertionError(e);
    }
  }

  public SecretKeySpec getCipherKey() {
    return cipherKey;
  }

  public SecretKeySpec getMacKey() {
    return macKey;
  }

  public IvParameterSpec getIv() {
    return iv;
  }
}
```

Figure 4.5: Computation of AES_KEY, $HMAC\_KEY$ and IV

After all the keys are computed, *encrypt* function, shown in Figure 4.6, takes padded message and encrypts it.

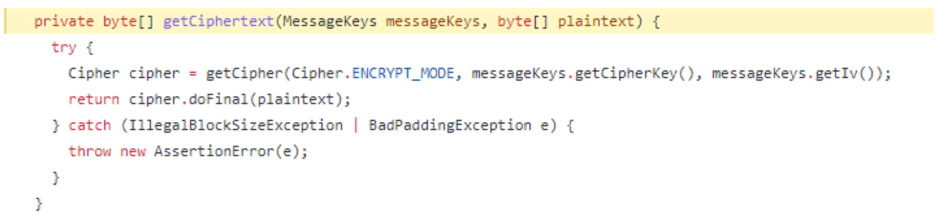


```
 * Encrypt a message.
 *
 * @param  paddedMessage The plaintext message bytes, optionally padded to a constant multiple.
 * @return A ciphertext message encrypted to the recipient+device tuple.
 */
public CiphertextMessage encrypt(byte[] paddedMessage) throws UntrustedIdentityException {
  synchronized (SESSION_LOCK) {
    SessionRecord sessionRecord   = sessionStore.loadSession(remoteAddress);
    SessionState  sessionState    = sessionRecord.getSessionState();
    ChainKey      chainKey        = sessionState.getSenderChainKey();
    MessageKeys   messageKeys     = chainKey.getMessageKeys();
    ECPublicKey   senderEphemeral = sessionState.getSenderRatchetKey();
    int           previousCounter = sessionState.getPreviousCounter();
    int           sessionVersion  = sessionState.getSessionVersion();

    byte[]            ciphertextBody    = getCiphertext(messageKeys, paddedMessage);
    CiphertextMessage ciphertextMessage = new SignalMessage(sessionVersion, messageKeys.getMacKey(),
                                                            senderEphemeral, chainKey.getIndex(),
                                                            previousCounter, ciphertextBody,
                                                            sessionState.getLocalIdentityKey(),
                                                            sessionState.getRemoteIdentityKey());

    if (sessionState.hasUnacknowledgedPreKeyMessage()) {
      UnacknowledgedPreKeyMessageItems items = sessionState.getUnacknowledgedPreKeyMessageItems();
      int localRegistrationId = sessionState.getLocalRegistrationId();

      ciphertextMessage = new PreKeySignalMessage(sessionVersion, localRegistrationId, items.getPreKeyId(),
                                                  items.getSignedPreKeyId(), items.getBaseKey(),
                                                  sessionState.getLocalIdentityKey(),
                                                  (SignalMessage) ciphertextMessage);
    }

    sessionState.setSenderChainKey(chainKey.getNextChainKey());

    if (!identityKeyStore.isTrustedIdentity(remoteAddress, sessionState.getRemoteIdentityKey(), IdentityKeyStore.Direction.SENDING)) {
      throw new UntrustedIdentityException(remoteAddress.getName(), sessionState.getRemoteIdentityKey());
    }

    identityKeyStore.saveIdentity(remoteAddress, sessionState.getRemoteIdentityKey());
    sessionStore.storeSession(remoteAddress, sessionRecord);
    return ciphertextMessage;
  }
}
```

Figure 4.6: Encryption and authentication

In the *encrypt* function, there are two function; *getCiphertext* and *SignalMessage*. *getCiphertext* function takes the padded plaintext and AES_KEY then obtain ciphertext, *SignalMessage* gets AD and computes HMAC of concatenated ciphertext and AD.

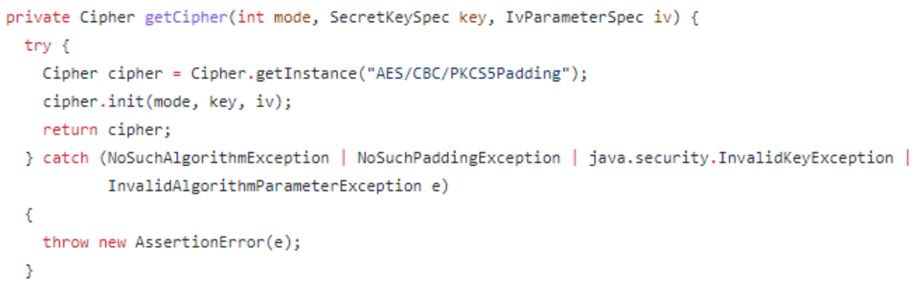
When look at the function *getCiphertext* in Figure 4.7, it calls *getCipher* function.



```
private byte[] getCiphertext(MessageKeys messageKeys, byte[] plaintext) {
  try {
    Cipher cipher = getCipher(Cipher.ENCRYPT_MODE, messageKeys.getCipherKey(), messageKeys.getIv());
    return cipher.doFinal(plaintext);
  } catch (IllegalBlockSizeException | BadPaddingException e) {
    throw new AssertionError(e);
  }
}
```

Figure 4.7: Encryption of plaintext

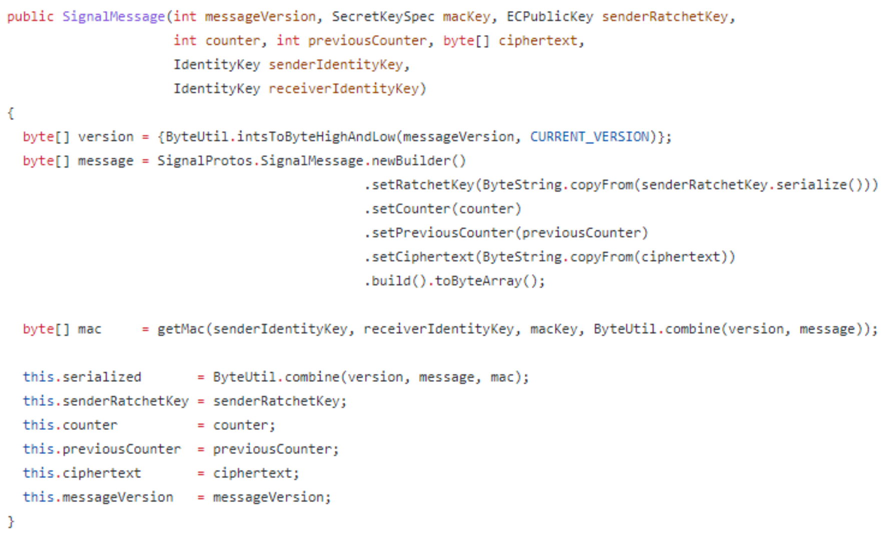
getCipher function shows that Signal use AES256 in CBC mode with PKCS#5 padding as shown in Figure 4.8.



```
private Cipher getCipher(int mode, SecretKeySpec key, IvParameterSpec iv) {
  try {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(mode, key, iv);
    return cipher;
  } catch (NoSuchAlgorithmException | NoSuchPaddingException | java.security.InvalidKeyException |
          InvalidAlgorithmParameterException e)
  {
    throw new AssertionError(e);
  }
```

Figure 4.8: AES in CBC mode with PKCS#5 padding

When look at the *SignalMessage* function shown in Figure 4.9, it takes parameters then return concatenated version, ciphertext and the HMAC of concatenated ciphertext and AD.



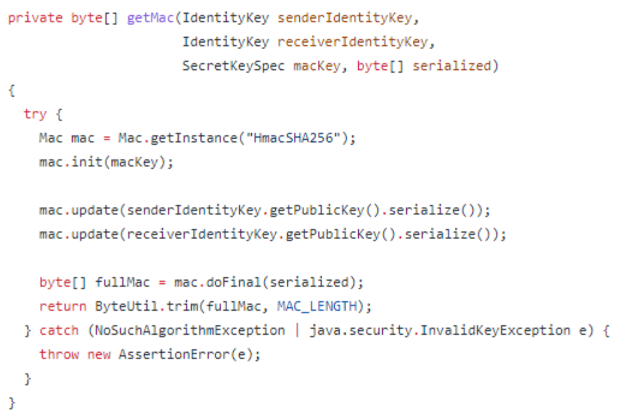
```
public SignalMessage(int messageVersion, SecretKeySpec macKey, ECPublicKey senderRatchetKey,
                     int counter, int previousCounter, byte[] ciphertext,
                     IdentityKey senderIdentityKey,
                     IdentityKey receiverIdentityKey)
{
  byte[] version = {ByteUtil.intsToByteHighAndLow(messageVersion, CURRENT_VERSION)};
  byte[] message = SignalProtos.SignalMessage.newBuilder()
                                             .setRatchetKey(ByteString.copyFrom(senderRatchetKey.serialize()))
                                             .setCounter(counter)
                                             .setPreviousCounter(previousCounter)
                                             .setCiphertext(ByteString.copyFrom(ciphertext))
                                             .build().toByteArray();

  byte[] mac     = getMac(senderIdentityKey, receiverIdentityKey, macKey, ByteUtil.combine(version, message));

  this.serialized       = ByteUtil.combine(version, message, mac);
  this.senderRatchetKey = senderRatchetKey;
  this.counter          = counter;
  this.previousCounter  = previousCounter;
  this.ciphertext       = ciphertext;
  this.messageVersion   = messageVersion;
}
```

Figure 4.9: Computation of AD and concatenation of ciphertext and HMAC output

HMAC is computed with function *getMac* as shown in Figure 4.10. *getMac* function takes IK_A, IK_B, $HMAC\_KEY$. First calculates AD.

$$AD = Encode(IK_A)||Encode(IK_B) \tag{4.1.7}$$

Then computes HMAC of concatenated the ciphertext and AD and gives the first 8 byte of HMAC output.



```
private byte[] getMac(IdentityKey senderIdentityKey,
                      IdentityKey receiverIdentityKey,
                      SecretKeySpec macKey, byte[] serialized)
{
  try {
    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(macKey);

    mac.update(senderIdentityKey.getPublicKey().serialize());
    mac.update(receiverIdentityKey.getPublicKey().serialize());

    byte[] fullMac = mac.doFinal(serialized);
    return ByteUtil.trim(fullMac, MAC_LENGTH);
  } catch (NoSuchAlgorithmException | java.security.InvalidKeyException e) {
    throw new AssertionError(e);
  }
}
```

Figure 4.10: Computation of associated data

Finally, the ciphertext and the first 8-byte of HMAC output are concatenated, then is sent to recipient.

4.2 LINPHONE

Linephone is a SIP based and an open source softphone for instant messaging. It uses Linphone Instant Message Encryption v2.0(LIME) Protocol [36].

LIME Protocol is a Signal based protocol with different parameters. Lime provides multiple devices per user and multiple users per device. Therefore LIME uses two different encryption mechanisms, one of them is optional. The first and main one is directly using Double Ratchet, the second and optional one is to encrypt the message with a random key and then encrypt the random key with Double Ratchet.

4.2.1 Key Exchange

LIME, different from Signal X3DH Protocol, generates, stores and transmits Identity Key in its EdDSA format and converts into X25519 or X448 format when an ECDH calculation is performed on it. All the other keys are stored in ECDH format.

Another discrepancy between LIME and Signal X3DH Protocol is HKDF. LIME uses HKDF which is based on SHA512. The size of output of HKDF is permissive and not subject to input or hash algorithm.

The parameters used by this protocol are shown in Table 4.2;

Table 4.2: X3DH parameters for LIME Protocol

Name	**Definitions**
curve	X25519 or X448
hash	SHA-512
info	"Lime"

To compute the shared secret key SK Lime Protocol uses X3DH protocol in the Signal Protocol with info *"Lime"* and salt is zero filled buffer in length of output of hash.

$$SK = HKDF(ZeroBuffer, F, ECDH_1||ECDH_2||ECDH_3||ECDH_4, \textit{Lime}) \tag{4.2.1}$$

In this equation F is a $0xFF$ filled buffer of length 32-byte or 57-byte depending on the curve25519 or curve448 respectively.

Lime also computes associated data AD using HKDF-SHA512. Firstly calculate AD_{input} with identity key IK and $deviceID$ is a unique string associated to a device, provided to Lime by Linphone.

$$AD_{input} = IK_B||IK_A||DeviceId_A||DeviceId_B \tag{4.2.2}$$

Then using HKDF-SHA512 with salt as zero filled buffer in length of output of hash,

AD_{input} and info *"X3DH Associated Data"*, AD is computed.

$$AD = HKDF(ZeroBuffer, AD_{input}, \textit{"X3DH Associated Data"}) \tag{4.2.3}$$

4.2.1.1 Double Ratchet

To calculate the first RK and the first CK, HKDF-SHA512 is used with the shared secret key SK,info *"DR Root Chain Key Derivation"*.

$$RK_0||CK_{0,0} = HKDF(SK, ECDH_{out}, \textit{"DR Root Chain Key Derivation"}) \tag{4.2.4}$$

After computation of the first RK and CK, in each double ratchet step compute the next RK and CK using the previous RK, the output of ECDH and the info *"DR Root Chain Key Derivation"*. The length of ECDH changes depend on curves. If curve25519 is used, the length of output of ECDH is 32 bytes, if curve448 is used then the length of output of ECDH is 36 bytes.

$$RK_i||CK_{i,0} = HKDF(RK_{i-1}, ECDH_{out}, \textit{"DR Root Chain Key Derivation"}) \tag{4.2.5}$$

To calculate the next CK and the message key MK, HMAC-SHA512 is used. The output length of the HMAC to generate MK is 48 byte, 32 bytes of 48 bytes is MK and the rest 16 bytes is AEAD nonce IV. If HMAC-SHA512 generates MK, it takes the previous CK and info $0x01$, if generates the next CK it takes the previous CK and info $0x01$.

$$CK = HMAC(CK, 0x02) \tag{4.2.6}$$

$$MK||IV = HMAC(CK, 0x01) \tag{4.2.7}$$

4.2.2 Encryption

Lime protocol, as presented above, uses two encryption mechanisms which one of them is optional.

The first one is that encrypt the message directly in the Double Ratchet. Figure 4.11 shows the encryption scheme.

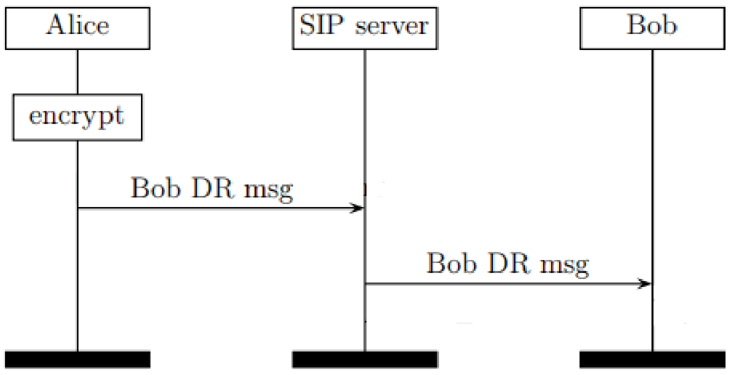


Figure 4.11: Double Ratchet Message encryption policy

For encryption, an AEAD encryption scheme based on SIV is used as recommended in Signal Protocol. It uses AES256 in GCM (Galois/Counter Mode) mode. This algorithm is considered reliable as old keys are deleted.

For encryption;

- Function takes MK, IV, plaintext and AD as inputs, then using AES-256 in GCM mode encrypts the plaintext.
- Also computes AD, using AD by computed X3DH Protocol. Let say $X3DH-AD$ instead of the first AD.

$$\begin{aligned} AD = RecipientUserId||SourcedeviceId||RecipientdeviceId \\ ||X3DH-AD<32bytes>||DRHeader \end{aligned} \tag{4.2.8}$$

- Finally concatenate the ciphertext and the HMAC output of concatenated ciphertext and $AD||X3DH-AD||header$ as AD, then it is sent to recipient.

The second one is that encrypt the message with a random key and then encrypt the random key in Double Ratchet shown in Figure 4.12. In Figure 4.12, "Bob DR msg" shows concatenated encrypted message and HMAC output, cipherMessage shows concatenated encrypted random seed and HMAC output.

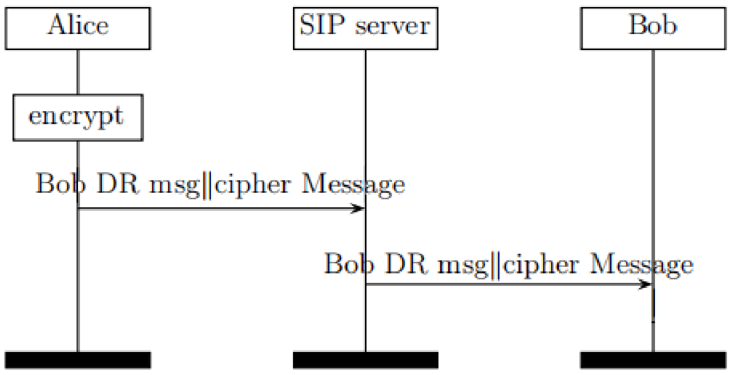


Figure 4.12: Cipher Message encryption policy

For encryption;

- First of all generate a 32-byte random seed.

- HKDF-SHA512 takes this random seed and “DR Message Key Derivation” as info then produces 32-byte encryption key AES_KEY and 16 bytes IV.

$$AES\_KEY||IV = HKDF(randomseed, \text{“\textit{DR Message Key Derivation}”}) \tag{4.2.9}$$

- The message is encrypted using AES256 in GCM mode with the AES_KEY and corresponding IV.

- Calculate HMAC of concatenated ciphertext and $sourceDeviceId||recipientUserId$ as AD, then the output is concatenated to the ciphertext.

- Also the random seed is encrypted using AES256 in GCM mode with the MK and corresponding IV.

- Compute AD,

$$AD = MessageTag < 16bytes > ||SourcedeviceId||RecipientdeviceId|| \\ X3DH - AD < 32bytes > ||DRHeader \tag{4.2.10}$$

- Calculate HMAC of concatenated ciphertext and AD, then the output is concatenated to the ciphertext of encryption random seed.

- Lastly, *Bob DR Msg* and *cipherMessage* are concatenated then it is sent to the recipient.

4.3 XABBER

Xabber is an open source XMPP client for Android system and Web. It uses OMEMO Protocol [40], which is an extension of XMPP, to encrypt communication between two clients.

In XMPP, there are two main end-to-end encryption schemes; Open PGP and Off-the-record Messaging. However, these schemes have some problems. For instance; OTR does not supply asynchronous messaging while Open PGP does not provide forward secrecy. For these reasons, OMEMO was obtained by developing the Double ratchet with X3DH key agreement protocol. The following section gives technical information about the protocol.

4.3.1 Key Exchange

OMEMO Protocol uses modified X3DH protocol to key agreement part. The parameters used by this protocol are shown in Table 4.3.

Table 4.3: X3DH parameters for OMEMO Protocol

Name	Definitions
curve	X25519 or X448
hash	SHA-256
info	"OMEMO X3DH"

OMEMO uses exactly the same X3DH key agreement protocol as presented above with its parameters.

$$SK = HKDF(ECDH_1||ECDH_2||ECDH_3||ECDH_4) \tag{4.3.1}$$

Alice also compute $AD = Encode(IK_A)||Encode(IK_B)$ as associated data.

4.3.1.1 Double Ratchet

OMEMO Protocol uses HKDF-SHA256 to calculate root keys. After computing the SK, it goes HKDF as input with info *"OMEMO X3DH"*, then the first RK and the first CK is obtained.

$$RK_0||CK_{0,0} = HKDF(SK, \textit{"OMEMO X3DH"}, 64) \tag{4.3.2}$$

After computing the first root key, it takes the previous root key RK, $ECDH_{out}$ as the output of ECDH and *"OMEMO Root Chain"* as info as inputs in each double ratchet.

$$RK_i||CK_{i,0} = HKDF(RK_{i-1}, ECDH_{out}, \textit{"OMEMO Root Chain"}, 64) \tag{4.3.3}$$

Then compute the next CK and MK using HMAC-SHA256. To obtain MK it takes the previous CK and constant $0x01$.

$$MK = HMAC(CK, 0x01) \tag{4.3.4}$$

To obtain the next CK it takes the previous CK and constant $0x02$.

$$CK = HMAC(CK, 0x02) \tag{4.3.5}$$

4.3.2 Encryption

For encryption, OMEMO protocol uses AES256 in CBC mode with HMAC-SHA256. However there is a difference between OMEMO protocol and Signal protocol. While, in the Signal protocol, message is taken as plaintext on the other hand, in OMEMO protocol, the message is encrypted then 32-byte encryption key and 32-byte HMAC key are taken as plaintext.

To encrypt the message OMEMO protocol usesAES256 in CBC mode and HMAC-SHA256.

- Generate a random encryption key RK which is crytographically secure random data [1].
- HKDF-SHA256 takes this encryption key as input, 256 zero-bits and HKDF info *"OMEMO Payload"* then gives output which has length 80-bytes.
- The first 32-byte of HKDF output is AES_KEY, the second 32-byte is $HMAC\_KEY$ and the last 16 byte is IV.

$$\begin{aligned} &AES\_KEY||HMAC\_KEY||AES\_IV \\ &= HKDF(0, RK, \textit{``OLM Payload"}, 80) \end{aligned} \tag{4.3.6}$$

- Using AES256 in CBC mode with padding PKCS#7 and AES_KEY and IV, the message is encrypted.
- Calculate HMAC-SHA256 takes ciphertext and $HMAC\_KEY$ as input and then the first 16-byte of output is concatenated to the ciphertext.

After encryption of the message, the concatenated encryption key and HMAC key is encrypted using double ratchet. OMEMO uses AES256 in CBC mode and HMAC-SHA256 in the Double Ratchet.

- After generation of MK it goes to HKDF with other inputs 256 zero-bit as salt and *"OMEMO Message Key Material"* as info and it gives output which has length 80-bytes.
- The first 32-byte of the output is AES_KEY, the second 32-byte is $HMAC\_KEY$ and last 16-byte is IV.

$$\begin{aligned} &AES\_KEY_{i,j}||HMAC\_KEY_{i,j}||AES\_IV_{i,j} \\ &= HKDF(0, MK_{i,j}, \text{“\textit{OLM Message Key Material}"}, 80) \end{aligned} \tag{4.3.7}$$

- Then using AES256 in CBC mode with PKCS#7 padding, the concatenated encryption key and HMAC key which are used to encrypt the message are encrypted with the computed keys and IV in the equation 4.3.7.
- The output and AD are concatenated then become input for HMAC.
- HMAC-SHA256 is computed with inputs, is taken the first 16-bytes and is appended to the output of AES and it is sent to recipient.

4.4 WIRE

Wire is an instant messaging application. It uses Proteus protocol [5] to encrypt text. Proteus protocol is a Signal based protocol with different parameters. Proteus protocol is open source and does not have a documentation. Therefore, source code has been analyzed.

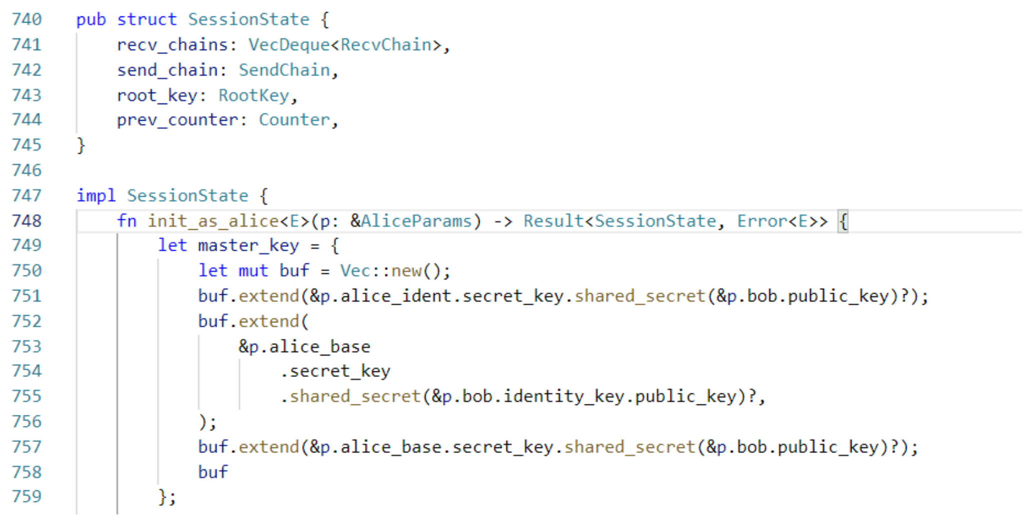
4.4.1 Key Exchange

Proteus Protocol uses X3DH protocol with the parameters shown in Table 4.4.

Table 4.4: X3DH parameters for Proteus protocol

Name	**Definitions**
curve	CurveX25519
hash	SHA-256
info	"handshake"

Firstly, the shared secret key which is named as *master_key* is computed using ECDH with curve25519 and inputs IK_A, IK_B, SPK_B and EK_A in function *SessionState* as shown in Figure 4.13.

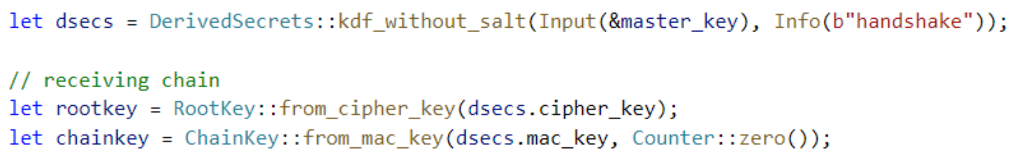


```
pub struct SessionState {
    recv_chains: VecDeque<RecvChain>,
    send_chain: SendChain,
    root_key: RootKey,
    prev_counter: Counter,
}

impl SessionState {
    fn init_as_alice<E>(p: &AliceParams) -> Result<SessionState, Error<E>> {
        let master_key = {
            let mut buf = Vec::new();
            buf.extend(&p.alice_ident.secret_key.shared_secret(&p.bob.public_key)?);
            buf.extend(
                &p.alice_base
                    .secret_key
                    .shared_secret(&p.bob.identity_key.public_key)?,
            );
            buf.extend(&p.alice_base.secret_key.shared_secret(&p.bob.public_key)?);
            buf
        };
```

Figure 4.13: Computation of shared secret key

Afterwards, *DerivedSecrets* function, shown in Figure 4.14, takes $master\_key$ and info *"handshake"* and gives the first RK and the first CK.

$$RK_0||CK_{0,0} = HKDF(SK, \text{"handshake"}, 64) \tag{4.4.1}$$



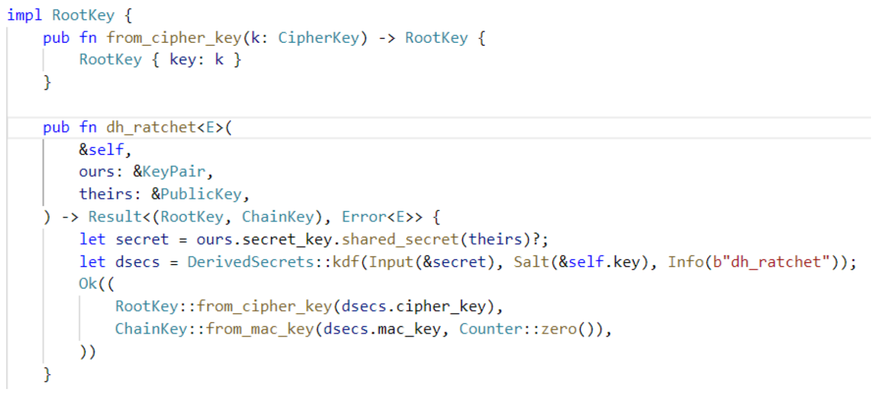
```
let dsecs = DerivedSecrets::kdf_without_salt(Input(&master_key), Info(b"handshake"));

// receiving chain
let rootkey = RootKey::from_cipher_key(dsecs.cipher_key);
let chainkey = ChainKey::from_mac_key(dsecs.mac_key, Counter::zero());
```

Figure 4.14: Computation of the first root and chain key

4.4.1.1 Double Ratchet

In the double ratchet procedure, in each ratchet, the new RK and the new CK are derived. To get the new RK, *DerivedSecret* function is called again with info *"dh_ratchet"* as shown in Figure 4.15.

$$RK_i||CK_{i,0} = HKDF(RK_{i-1}, \text{"dh\_ratchet"}, 64) \tag{4.4.2}$$



```
impl RootKey {
    pub fn from_cipher_key(k: CipherKey) -> RootKey {
        RootKey { key: k }
    }

    pub fn dh_ratchet<E>(
        &self,
        ours: &KeyPair,
        theirs: &PublicKey,
    ) -> Result<(RootKey, ChainKey), Error<E>> {
        let secret = ours.secret_key.shared_secret(theirs)?;
        let dsecs = DerivedSecrets::kdf(Input(&secret), Salt(&self.key), Info(b"dh_ratchet"));
        Ok((
            RootKey::from_cipher_key(dsecs.cipher_key),
            ChainKey::from_mac_key(dsecs.mac_key, Counter::zero()),
        ))
    }
```

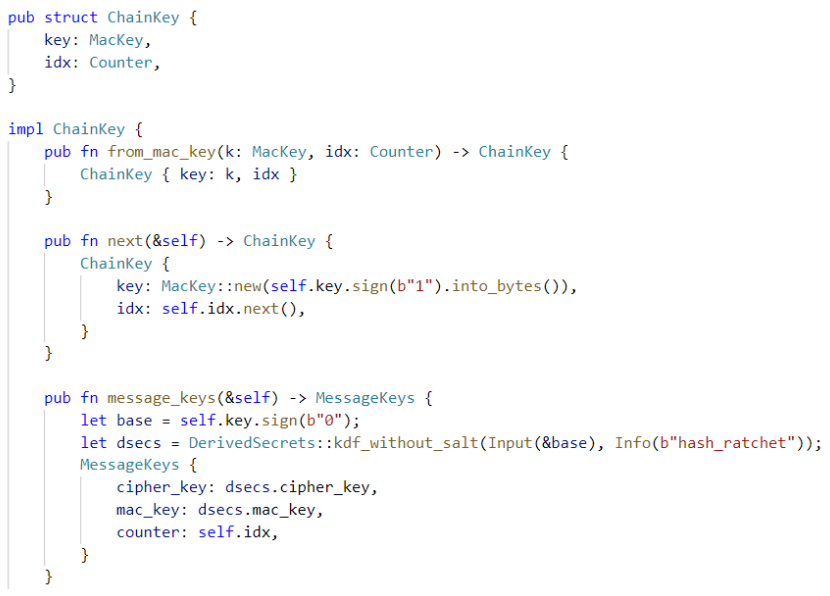
Figure 4.15: Computation of the next root and chain key

After obtaining RK and CK, CK, which is mentioned as *MacKey* in Figure 4.16, goes to function *next* then the new chain key is obtained with constant $0x01$.

$$CK = HMAC(CK, 0x02) \tag{4.4.3}$$

$$MK = HMAC(CK, 0x00) \tag{4.4.4}$$

Then CK goes to function *DerivedSecrets* with info *"hash_ratchet"* and constant $0x00$ then message key is obtained as shown in Figure 4.16.



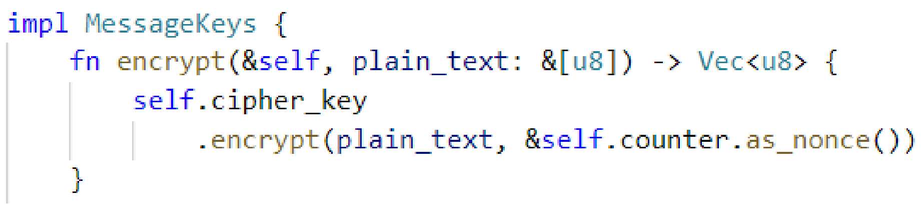
```rust
pub struct ChainKey {
    key: MacKey,
    idx: Counter,
}

impl ChainKey {
    pub fn from_mac_key(k: MacKey, idx: Counter) -> ChainKey {
        ChainKey { key: k, idx }
    }

    pub fn next(&self) -> ChainKey {
        ChainKey {
            key: MacKey::new(self.key.sign(b"1").into_bytes()),
            idx: self.idx.next(),
        }
    }

    pub fn message_keys(&self) -> MessageKeys {
        let base = self.key.sign(b"0");
        let dsecs = DerivedSecrets::kdf_without_salt(Input(&base), Info(b"hash_ratchet"));
        MessageKeys {
            cipher_key: dsecs.cipher_key,
            mac_key: dsecs.mac_key,
            counter: self.idx,
        }
    }
```

Figure 4.16: Computation of the next chain key and the message key

4.4.2 Encryption

Proteus protocol uses Chacha20 stream cipher for encryption. The first 32 byte of message key is *cipher_key* to use in encryption, the second 32 byte is *mac_key* to use in HMAC and the last 8 byte is the nonce value to use in encryption.

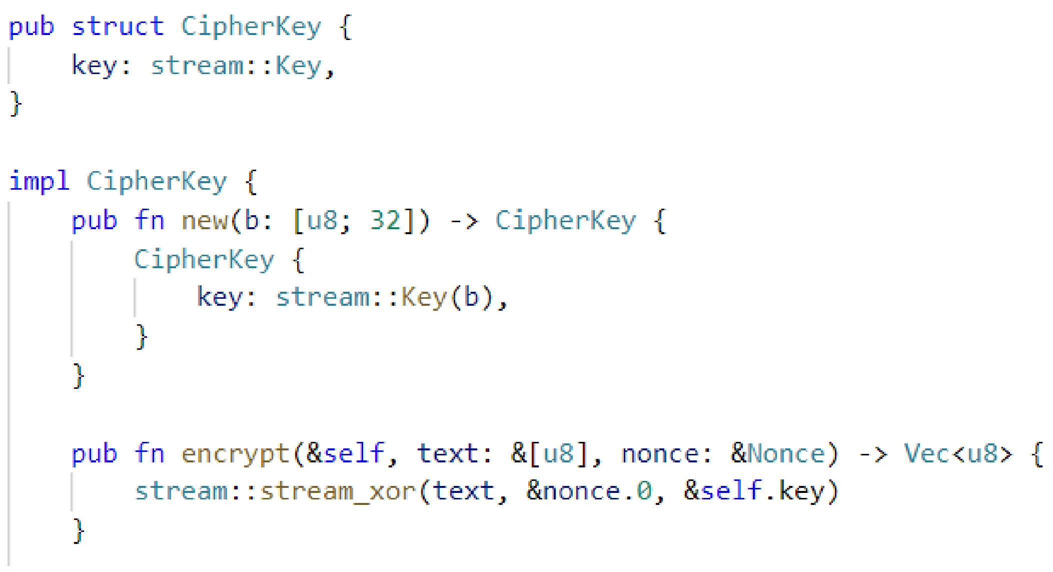
The *encrypt* function takes *cipher_key* as encryption key and plain text as shown in Figure 4.17.



```rust
impl MessageKeys {
    fn encrypt(&self, plain_text: &[u8]) -> Vec<u8> {
        self.cipher_key
            .encrypt(plain_text, &self.counter.as_nonce())
    }
```

Figure 4.17: Encryption I

In this function also another *encrypt* function from structure *CipherKey* is called. The Figure 4.18 shows that the function takes plaintext, *cipher_key* and nonce value and gives the ciphertext.



```
pub struct CipherKey {
    key: stream::Key,
}

impl CipherKey {
    pub fn new(b: [u8; 32]) -> CipherKey {
        CipherKey {
            key: stream::Key(b),
        }
    }

    pub fn encrypt(&self, text: &[u8], nonce: &Nonce) -> Vec<u8> {
        stream::stream_xor(text, &nonce.0, &self.key)
    }
```

Figure 4.18: Encryption II

4.5 ELEMENT (RIOT.IM)

Riot is an open source messaging application. It uses Matrix protocol for encryption. Matrix has two libraries to encrypt messages; Megolm library and Olm library [4]. Actually, Megolm library is an expansion of Olm library for rooms. Olm library is used for encryption in one-to-one communications.

4.5.1 Key Exchange

Olm library uses modified X3DH protocol to key agreement part with these parameters shown in Table 4.5.

Table 4.5: X3DH parameters for Olm Protocol

Name	**Definitions**
curve	X25519
hash	SHA-256
info	"OLM_ROOT"

It takes identity keys, IK_A and IK_B, and ephemeral keys, E_A and E_B, which are generated by using curve25519 then using X3DH protocol generates secret shared key SK. Different from the Signal Protocol, in Olm Protocol X3DH takes Bob's ephemeral key instead of Bob's signed prekey and there is no one time prekey in Bob's keys bundle.

$$SK = ECDH(IK_A, E_B)||ECDH(E_A, IK_B)||ECDH(E_A, E_B) \tag{4.5.1}$$

After compute the SK, it uses this key to compute the first RK.

4.5.1.1 Double Ratchet

As in the Signal Protocol, Matrix Protocol uses SK to generate the first RK and CK with info *"OLM_ROOT"*.

$$RK_0||CK_{0,0} = HKDF(0, SK, \textit{"OLM\_ROOT"}, 64) \tag{4.5.2}$$

After the first RK and CK generation, in each double ratchet step, HKDF-SHA256 will be applied again. It takes the previous RK, the result of ECDH where one of the inputs is the public key of one side and the other input is the private key of the other side and info *"OLM_RATCHET"* to get the next RK.

$$R_i||C_{i,0} = HKDF(R_{i-1}, ECDH_{out}, \textit{"OLM\_RATCHET"}, 64) \tag{4.5.3}$$

HMAC-SHA256 will be applied to obtain the next CK with inputs the previous CK and constant $0x02$.

$$C_{i,j} = HMAC(C_{i,j-1}, 0x02) \tag{4.5.4}$$

To generate message key MK, HMAC-SHA256 will be applied with inputs current CK and constant $0x01$.

$$M_{i,j} = HMAC(C_{i,j}, 0x01) \tag{4.5.5}$$

4.5.2 Encryption

After computation of the message key, using HKDF-SHA256 with input message key MK and info *"OLM_KEYS"*, 256 bit AES_KEY, 256 bit $HMAC\_KEY$ and 128 bit AES_IV are computed.

$$\begin{aligned} &AES\_KEY_{i,j}||HMAC\_KEY_{i,j}||AES\_IV_{i,j} \\ &= HKDF(0, MK_{i,j}, \textit{"OLM\_KEYS"}, 80) \end{aligned} \tag{4.5.6}$$

Afterward, using AES-256 in CBC mode with padding PKCS#7 with inputs which are the plaintext, the key $AES\_KEY_{i,j}$ and $AES\_IV_{i,j}$, the ciphertext $X_{i,j}$ is obtained.

Additionally HMAC-SHA256 is used for authentication. The complete message is sent to HMAC-SHA256 and the first 8 bytes of output are appended to the message and it is sent to recipient.

CHAPTER 5

POST-QUANTUM ALGORITHMS FOR SIGNAL AND SIGNAL BASED ALGORITHMS

Throughout the thesis, we explained the key exchange and encryption algorithms of the Signal Protocol and four different protocols developed based on the Signal Protocol. While public key algorithms are used for key exchange, symmetric algorithms are used for encryption. With today's technology, there is no algorithm that can efficiently break public key algorithms. In 1997, however, Shor's algorithm [39] showed that integer factorization problem and discrete logarithm problem can be broken with quantum computers in polynomial time. Therefore, post-quantum algorithms should replace classical public key algorithms. Symmetric key algorithms, on the other hand, can be made post-quantum resistant by strengthening their parameters. Therefore, first of all, it should be focused on the public-key algorithms used in the Signal Protocol.

In their study published in 2020, Alwen et al. [12] offered the answer to the question of how to make the Double Ratchet mechanism quantum-proof, assuming a quantum-proof key exchange is done. This study, in which KEM algorithms are used, is the first study for Double Ratchet. Since the public key algorithms in Double Ratchet have become quantum resistant with this work, only the public key algorithms in the X3DH protocol are included in this section.

In this section, we will focus on which post-quantum algorithm can be substituted for the key exchange algorithm in X3DH. All protocols presented in Chapter 4 uses the same key exchange algorithm with different parameters, therefore, the algorithms presented below are offered as suggestions for all protocols.

Today, many studies are carried out on post-quantum. For example, the 4th round has been reached in the work carried out by NIST to standardize post quantum algorithms [3]. As stated in the Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process [9], the first standardized algorithm is CRYSTALS–KYBER [14]. Also in round 4, BIKE, Classic McEliece, HQC, and SIKE will be considered for standardization. It is considered that these studies will also benefit to the Signal Protocol. As a matter of fact, Wire application has started post quantum studies by applying the NewHope [10] algorithm, one of the NIST tour 2 candidates, to its protocol.

Some work is being done to make the Signal Protocol resistant to quantum. It is possible to collect the examined studies under three headings; Key exchange protocols based on SIDH[1], CSIDH or KEM. In this section, we will review the early studies on SIDH, CSIDH, and KEM and compare these concepts and decide which one is more useful for the Signal Protocol.

The first study, The Post-Quantum Signal Protocol: Secure Chat in a Quantum World [23], was published in 2019. In this study, Duits et al. implement 10 different KEMs and 45 different version of these KEMs to key exchange in the Double Ratchet and SIDH to key exchange in the X3DH. They also evaluate them in terms of CPU usage, storage space, bandwidth and energy efficiency.Although ECDH is used on both Double Ratchet and X3DH, SIDH is preferred over KEM on X3DH as KEM does not enable key reuse. KEM algorithms used in the study are Big Quake, Frodo, BIKE, Leda, Crystal-Kyber, New Hope, Lima, SIKE, Saber, and Titanium, SIDH algorithms used are SIDH503 and SIDH751. However, because of Alwen et al.'s study [12] Double Ratchet is out of the scope. Key sizes used by SIDH are shown in Table 5.1.

[1] Castryck and Decru announced that they broke SIDH in their study published in July 2022[17]. However, due to the publication of the relevant article shortly before the publication of our study, we did not change this part of our study, but we would like to point out that SDIH is not suitable for post-quantum use.

Table 5.1: Key sizes in Curve25519, SIDH503 and SIDH751 [23]

	Curve25519	**SIDH503**	**SIDH751**
Public Key	32	378	564
Private Key	32	32	48
Security Level	0	1	3

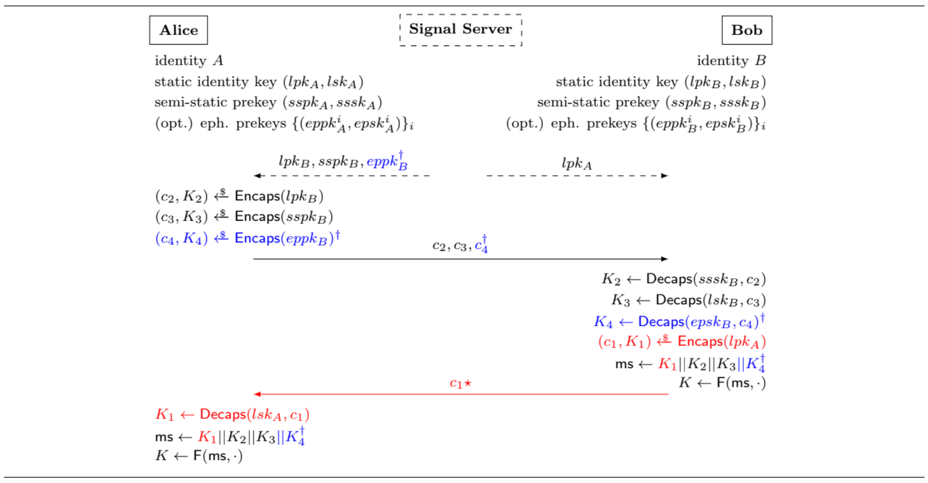
After all evaluations they decide to use both kyber512 and SIDH512 algorithms. Because in this situation per message delay only 0.02 seconds. However, to implement KEM to Signal protocol there are some changes in the protocol. On the other hand, to implement SIDH is more easy and in that situation delay is 0.03 seconds per message.

In another study conducted in 2019, Alvila et al. apply CSIDH to Signal and compare it with the original Signal and evaluate the results in A Performance Evaluation of Post-Quantum Cryptography in the Signal Protocol [11]. Alvila et al. choose CSIDH because all PQCRYPT and NIST's algorithms use key encapsulation mechanism. These algorithms, because they are KEM, cannot be used as static-static key exchange like ECDH in Signal however CSIDH is provided as a post-quantum algorithm. In addition to this, it is necessary to make changes to the Signal Protocol to implement KEM. In the study, thread times in milliseconds were compared when sending the first message, sending consecutive messages, and replying to the incoming message using Curve25519, CSIDH-512 and CSIDH-1024. The speed of CSIDH is practical. It is also stated in the article that the key lengths of CSIDH are short as shown in Table 5.2.

Table 5.2: Key sizes in Curve25519, CSIDH-512 and CSIDH-1024 [11]

	Curve25519	**CSIDH-512**	**CSIDH-1024**
Public Key	32	64	128
Private Key	32	32	64
Security Level	0	1	3

In 2020, Brendel et al. published Towards Post-Quantum Security for Signal's X3DH Handshake [16]. They say that SIDH and CSIDH satisfies key reuse but SIDH can be attacked when keys are reused, while parameter selection of CSIDH is doubtful and it has high cost. Therefore they came up with a new notions; split Key Encapsulation Mechanism or split KEM. Split KEM is actually a KEM that allows both parties to contribute to the encapsulation with either a one-time key or a static key. Key generation which is on the encapsulator side is separated from the encapsulation algorithm in split KEM. This enables key reuse as in the Diffie-Hellman algorithm. As seen in Figure 5.1, required for key sharing containing Alice's identity key, the last stream which is red, breaks X3DH's asynchronicity.



Alice
Signal Server
Bob
identity A
static identity key (lpk_A, lsk_A)
semi-static prekey $(sspk_A, sssk_A)$
(opt.) eph. prekeys $\{(eppk_A^i, epsk_A^i)\}_i$
identity B
static identity key (lpk_B, lsk_B)
semi-static prekey $(sspk_B, sssk_B)$
(opt.) eph. prekeys $\{(eppk_B^i, epsk_B^i)\}_i$
$lpk_B, sspk_B, eppk_B^\dagger$
lpk_A
$(c_2, K_2) \xleftarrow{\$} \mathsf{Encaps}(lpk_B)$
$(c_3, K_3) \xleftarrow{\$} \mathsf{Encaps}(sspk_B)$
$(c_4, K_4) \xleftarrow{\$} \mathsf{Encaps}(eppk_B)^\dagger$
$c_2, c_3, c_4^\dagger$
$K_2 \leftarrow \mathsf{Decaps}(sssk_B, c_2)$
$K_3 \leftarrow \mathsf{Decaps}(lsk_B, c_3)$
$K_4 \leftarrow \mathsf{Decaps}(epsk_B, c_4)^\dagger$
$(c_1, K_1) \xleftarrow{\$} \mathsf{Encaps}(lpk_A)$
$\mathsf{ms} \leftarrow K_1 || K_2 || K_3 || K_4^\dagger$
$K \leftarrow \mathsf{F}(\mathsf{ms}, \cdot)$
$c_1 \star$
$K_1 \leftarrow \mathsf{Decaps}(lsk_A, c_1)$
$\mathsf{ms} \leftarrow K_1 || K_2 || K_3 || K_4^\dagger$
$K \leftarrow \mathsf{F}(\mathsf{ms}, \cdot)$

Figure 5.1: Signal's X3DH key exchange with KEMs replacing the Diffie-Hellman operations. The optional ephemeral prekey (combination) shown in blue[16].

In split KEM, as seen in Figure 5.2, both Alice and Bob can reuse their key pairs and contribute encapsulation. Further, split KEM ensures asynchronicity as there is no additional message flow from Bob to Alice. The NIST submitted KEMs which are passively-secure, particularly lattice-based, could be used as the split KEM format because the encapsulation mechanisms can be divided into a key generation and a shared key computing part.

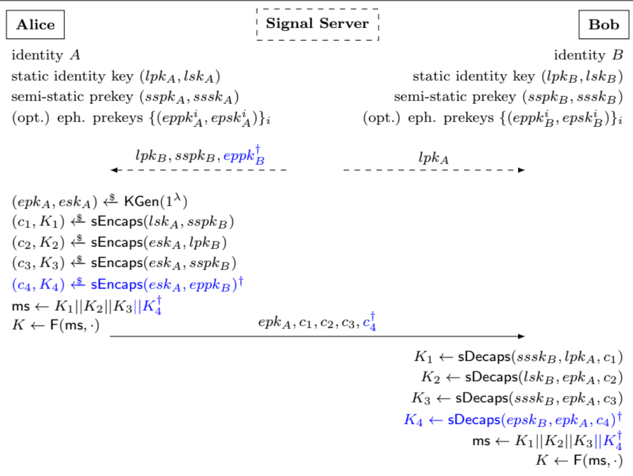

Alice
Signal Server
Bob
identity A
static identity key (lpk_A, lsk_A)
semi-static prekey $(sspk_A, sssk_A)$
(opt.) eph. prekeys $\{(eppk_A^i, epsk_A^i)\}_i$
identity B
static identity key (lpk_B, lsk_B)
semi-static prekey $(sspk_B, sssk_B)$
(opt.) eph. prekeys $\{(eppk_B^i, epsk_B^i)\}_i$
$lpk_B, sspk_B, eppk_B^\dagger$
lpk_A
$(epk_A, esk_A) \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$
$(c_1, K_1) \xleftarrow{\$} \mathsf{sEncaps}(lsk_A, sspk_B)$
$(c_2, K_2) \xleftarrow{\$} \mathsf{sEncaps}(esk_A, lpk_B)$
$(c_3, K_3) \xleftarrow{\$} \mathsf{sEncaps}(esk_A, sspk_B)$
$(c_4, K_4) \xleftarrow{\$} \mathsf{sEncaps}(esk_A, eppk_B)^\dagger$
$\mathsf{ms} \leftarrow K_1 || K_2 || K_3 || K_4^\dagger$
$K \leftarrow \mathsf{F}(\mathsf{ms}, \cdot)$
$epk_A, c_1, c_2, c_3, c_4^\dagger$
$K_1 \leftarrow \mathsf{sDecaps}(sssk_B, lpk_A, c_1)$
$K_2 \leftarrow \mathsf{sDecaps}(lsk_B, epk_A, c_2)$
$K_3 \leftarrow \mathsf{sDecaps}(sssk_B, epk_A, c_3)$
$K_4 \leftarrow \mathsf{sDecaps}(epsk_B, epk_A, c_4)^\dagger$
$\mathsf{ms} \leftarrow K_1 || K_2 || K_3 || K_4^\dagger$
$K \leftarrow \mathsf{F}(\mathsf{ms}, \cdot)$

Figure 5.2: Signal's X3DH key exchange with split KEMs replacing the Diffie-Hellman operations. The optional ephemeral prekey (combination) shown in blue [16].

It has been previously stated that Diffie-Hellman security properties cannot be achieved using KEM alone, two studies conducted in 2021 are also on this. The first one is An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable [26] and the second one is Post-quantum Asynchronous Deniable Key Exchange and the Signal Handshake [15]. While Hashimato et al. use KEM and signature scheme in [26], Brendel et al. use KEM and Designated Verifier Signature (DVS) in [15]. The signature algorithms are out of our scope. But we can say that both studies provided Diffie-Hellman security features such as asynchronicity, deniability, forward secrecy and authenticity.

Table 5.3 shows the comparison of the studies reviewed in terms of Diffie-Hellman safety features.

Table 5.3: Comparison of studies in terms of Diffie-Hellman security properties

	Asynchronicity	**Deniability**	**Forward secrecy**	**Authenticity**
CSIDH	✓	✓	✓(weak)	✓
SIDH		✓	✓	✓
Split KEM	✓			✓
KEM and Signature	✓	✓	✓	✓
KEM and DVS	✓	✓	✓	✓

As a result, each study to make the Signal Protocol resistant to quantum has its own advantages and disadvantages. Moreover the study which is published in July 2022 described breaking of SIDH[17]. Although it is easy to apply to the SIDH and CSIDH Signal Protocol and various changes are needed in the protocol to implement KEM, KEM is the area that needs work in our opinion. Because it provides all the security features of Diffie-Hellman, asynchronicity, deniability, forward secrecy and authenticity, by being supported by signatures, and since all of the NIST Post Quantum Cryptography Standardization process candidates are KEM, it will be standardized in the near future.

CHAPTER 6

CONCLUSION

In this thesis, firstly, we give a cryptographic background. Cryptographic primitives, used key exchange algorithms and used encryption algorithms are describe shortly. After that, the Signal Protocol which is the main protocol of the thesis is explained and give general perspective for choosing parameters.

Next, we presented Signal based and open source instant messaging applications; Linphone, Xabber, Wire and Element. We touched on which algorithms they use for key exchange and encryption in one-to-one communication, what the inputs of these algorithms are, and how the AD they use for authentication is obtained. We also documented the information that is not included in the documentation of Wire and Signal applications by examining the source codes.

Finally, we examined the post-quantum algorithms that can replace ECDH in the key exchange protocol of the Signal Protocol, discussed their advantages and disadvantages, and tried to decide on the most useful one for the Signal Protocol.

For the future works; those who want to develop applications using the Signal Protocol can look at how other applications implement the Signal Protocol in the light of this thesis. For example, the Wire application used a different encryption algorithm than the algorithms suggested by Signal. By making changes like this, new end-to-end encrypted instant messaging applications can be developed. Furhermore, post-quantum is a very new and developing field. As described in this thesis, none of the post-quantum algorithms presented is perfect for the Signal Protocol. An algorithm that can be easily implemented to Signal and can provide all the security properties of Diffie-Hellman such as asynchronicity, deniability can be developed.

REFERENCES

[1] Crytographically secure random data. `https://datatracker.ietf.org/doc/html/rfc4086`, accessed: 2022-08-20.

[2] Facebook messenger. `https://www.messenger.com/`, accessed: 2022-08-20.

[3] Nist post-quantum standardization protocol. `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization`, accessed: 2022-08-20.

[4] Olm protocol. `https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md`, accessed: 2022-08-20.

[5] Proteus protocol. `https://github.com/wireapp/proteus`, accessed: 2022-08-20.

[6] Secure messaging scorecard. which apps and tools actually keep your messages safe? `https://www.eff.org/node/101713/`, accessed: 2022-08-20.

[7] Skype. `https://www.skype.com/`, accessed: 2022-08-20.

[8] Whatsapp. `https://whatsapp.com`, accessed: 2022-08-20.

[9] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, R. Peralta, et al. Status report on the third round of the nist post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology Gaithersburg, MD, 2022.

[10] E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. De La Piedra, P. S. T. Pöppelmann, and D. Stebila. Newhope. *Submission to the NIST Post-Quantum Cryptography standardization project, Round*, 2, 2019.

[11] M. Alvila. A performance evaluation of post-quantum cryptography in the signal protocol, 2019.

[12] J. Alwen, S. Coretti, and Y. Dodis. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–158. Springer, 2019.

[13] D. J. Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[14] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[15] J. Brendel, R. Fiedler, F. Günther, C. Janson, and D. Stebila. Post-quantum asynchronous deniable key exchange and the signal handshake. In *IACR International Conference on Public-Key Cryptography*, pages 3–34. Springer, 2022.

[16] J. Brendel, M. Fischlin, F. Günther, C. Janson, and D. Stebila. Towards post-quantum security for signal's x3dh handshake. In *International Conference on Selected Areas in Cryptography*, pages 404–430. Springer, 2020.

[17] W. Castryck and T. Decru. An efficient key recovery attack on sidh (preliminary version). *Cryptology ePrint Archive*, 2022.

[18] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes. Csidh: an efficient post-quantum commutative group action. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 395–427. Springer, 2018.

[19] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, 2020.

[20] J. Daemen and V. Rijmen. Aes proposal: Rijndael. 1999.

[21] M. Di Raimondo, R. Gennaro, and H. Krawczyk. Secure off-the-record messaging. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages 81–89, 2005.

[22] W. Diffie and M. E. Hellman. New directions in cryptography. In *Secure communications and asymmetric cryptosystems*, pages 143–180. Routledge, 2019.

[23] I. Duits. The post-quantum signal protocol: Secure chat in a quantum world. Master's thesis, University of Twente, 2019.

[24] M. Fetter. *New Concepts for Presence and Availability in Ubiquitous and Mobile Computing: Enabling Selective Availability through Stream-Based Active Learning*, volume 33. University of Bamberg Press, 2019.

[25] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is textsecure? In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 457–472. IEEE, 2016.

[26] K. Hashimoto, S. Katsumata, K. Kwiatkowski, and T. Prest. An efficient and generic construction for signal's handshake (x3dh): post-quantum, state leakage secure, and deniable. *Journal of Cryptology*, 35(3):1–78, 2022.

[27] K. Igoe, D. McGrew, and M. Salter. Fundamental Elliptic Curve Cryptography Algorithms. RFC 6090, Feb. 2011.

[28] D. Jao and L. D. Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[29] D. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Feb. 1997.

[30] D. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[31] M. Marlinspike and T. Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283, 2016.

[32] M. Marlinspike and T. Perrin. The sesame algorithm: session management for asynchronous message encryption. *Revision*, 2:2017–04, 2017.

[33] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, Jan. 2008.

[34] D. Moody and D. Shumow. Analogues of vélu's formulas for isogenies on alternate models of elliptic curves. *Mathematics of Computation*, 85(300):1929–1951, 2016.

[35] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[36] J. Pascal. Linphone instant message encryption v2. 0 (lime v2. 0). 2018.

[37] T. Perrin. The xeddsa and vxeddsa signature schemes. *Specification. Oct*, 2016.

[38] T. Perrin and M. Marlinspike. The double ratchet algorithm. *GitHub wiki*, 2016.

[39] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[40] A. Straub, D. Gultsch, T. Henkes, K. Herberth, P. Schaub, and M. WiBfeld. Xep-0384: Omemo encryption. *XMPP Extension Protocol, XEP*, 384, 2009.

[41] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249. IEEE, 2015.

Appendix A

DIFFIE-HELLMAN KEY EXCHANGE

Diffie-Hellman Key Exchange [22] is a public-key protocol that allows two parties to share the secret key over an public channel. In this protocol, firstly, both of the parties agree on a finite cyclic group G in $\mod p$ where p is prime and generator g in G. Then each other of parties selects a random number between $[0, p-1]$, these random numbers are their private keys. Let a is random number of Alice and b is random number of Bob. They compute their public keys $A = g^a \mod p$ and $B = g^b \mod p$ then send them to each other. Alice takes $B \mod p$ and computes;

$$B^a = (g^b \mod p)^a \mod p = g^{ab} \mod p$$

Similarly, Bob takes $A \mod p$ and computes

$$A^b = (g^a \mod p)^b \mod p = g^{ab} \mod p$$

Computed $g^{ab} \mod p$ is the shared secret key for parties.

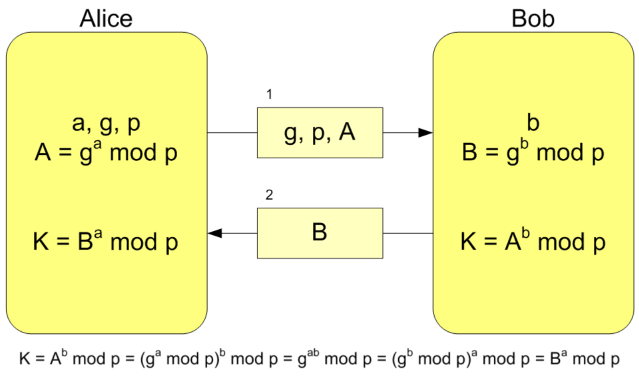

Alice
Bob
a, g, p
A = g^a mod p
K = B^a mod p
1
g, p, A
2
B
b
B = g^b mod p
K = A^b mod p
K = A^b mod p = (g^a mod p)^b mod p = g^ab mod p = (g^b mod p)^a mod p = B^a mod p

Figure A.1: Diffie-Hellman Key Exchange

Appendix B

ECDH KEY EXCHANGE

ECDH[27] is a key exchange protocol that allows the private key to be shared between two parties over a common open channel. It is a type of Diffie-Hellman protocol that provides key exchange using elliptic curve cryptography.

In ECDH, like Diffie-Hellman protocol firstly, both of the parties agree on elliptic curve over a finite field $\mathbb{F}_p$ with generator point G and order p. Then each other of parties selects a random number between $[0, p-1]$, these random numbers are their private keys. Let a is random number of Alice and b is random number of Bob. They compute their public keys $A = a \cdot G$ and $B = b \cdot G$ then send them to each other. Alice takes B and computes the point

$$(x, y) = B \cdot a = (b \cdot G) \cdot a$$

Similarly Bob takes A and computes the point

$$(x, y) = A \cdot b = (a \cdot G) \cdot b$$

The x coordinate of the point (x, y) is the shared secret key for parties.

Appendix C

AES

AES[20] is a symmetric block cipher encryption algorithm. While AES takes a fixed-length 128-bit block, it can take keys of three different lengths; 128, 192 and 256 bits. Depending on the key length respectively, it has 10, 12 and 14 rounds. In each round there are 4 steps;

- `SubBytes`: In this step, each byte is replaced with another byte using a substutioin box.
- `ShiftRows`: It is a transposition step. The last three rows are shifted a certain number of steps cyclically.
- `MixColumns`: It provides linear mixing by combining the four bytes in each column.
- `AddRoundKey`: This step implement bitwise XOR to each byte of the state with corresponding byte of the round key.

Only `AddRoundKey` is applied before the first round, `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` are applied respectively in all rounds except the last round. In the last round `MixColumns` is not applied, `SubBytes`, `ShiftRows` and `AddRoundKey` steps are applied respectively.

Appendix D

CHACHA20

Chacha [13] is a modification of stream cipher Salsa20. The Chacha's difference from Salsa is that it uses a new round function to increase diffusion and performance. Chacha takes 256-bit keys, 32-bit initial counter, 96-bit nonce and 128-bit constant. All of these values are divided into 32 bit and placed in a matrix like this.

$$\begin{bmatrix} constant & constant & constant & constant \\ key & key & key & key \\ key & key & key & key \\ counter & nonce & nonce & nonce \end{bmatrix}$$

Then, each round "Quarter-round(QR)" is applied. Quarter-round is a function as defined;

$$QR(a,b,c,d) = \begin{cases} a+=b, & d\oplus=a, & d <<<= 16 \\ c+=d, & b\oplus=c, & b <<<= 12 \\ a+=b, & d\oplus=a, & d <<<= 8 \\ c+=d, & b\oplus=c, & b <<<= 7 \end{cases} \tag{D.0.1}$$

In the odd number round QR is applied to columns, in the even number round QR is applied diagonals. If we index the matrix as follow;

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

QR is applied in odd number round as follow;

$$\begin{aligned}&QR(0,4,8,12)\\&QR(1,5,9,13)\\&QR(2,6,10,14)\\&QR(3,7,11,15)\end{aligned}$$

in even number round as follow;

$$\begin{aligned}&QR(0,5,10,15)\\&QR(1,6,11,12)\\&QR(2,7,8,13)\\&QR(3,4,9,14)\end{aligned}$$

Chacha20 [35] is a instance of Chacha where 20 rounds are used. After obtaining the key, the key and plaintex are added bitwise XOR to get ciphertext.