UNIVERSITY OF TRENTO

Advanced Programming of Cryptographic Methods

Project Report

The Rust Unique Secure Talk (T.R.U.S.T)

Matteo Bordignon, Alessandro Perez, Christian Sassi

June 4, 2025

Contents

Chapter 1

Introduction

This report presents the design, implementation, and evaluation of **T.R.U.S.T.** (**T**he **R**ust **U**nique **S**ecure **T**alk), a terminal user interface (TUI)-based secure chat application developed in Rust. The primary objective of this project was to explore secure communication principles and implement a reliable end-to-end encryption system.

To achieve this purpose we combined different cryptographic primitives to create a key agreement protocol based on **X3DH** [2] and later encrypt each message with a new key given by the key management scheme: **Double Ratchet** [1] first introduced by Signal.

The combination of the two gives resilience to both practical and complete leak of the encryption key during the same or different sessions as we will see in later chapters 4. More over thanks to the choice of **Rust** we are sure of not having memory issues or data race conditions, also when a variable is out-of scope we are sure it is set to zero before deleting it.

1.1 Organization of the Report

This document is organized as follows: Introduction 1 provides an overview of the application's purpose and core functionalities; Requirements 2 defines both the functional and security requirements; Technical Details 3 outlines the system's architecture, the main implementation decisions, and the code structure; Security Considerations 4 discusses cryptographic measures and potential threats; Known Limitations 5 highlights current constraints and possible improvements; and finally, Instructions for Installation and Execution 6 guides readers through setting up and running the application.

1.2 Overview of the Protocol

The first step to achieve end-to-end encryption is to have a secure method of establishing a shared secrete. There exists many algorithms mainly based on public key cryptography, we decided to implement **X3DH**, both for its security derived by the Diffie-Hellman algorithm and the use of signatures to prevent miss-bindings compared to the classical implementation of Diffie-Hellman.

1.2.1 Extended Triple Diffie-Hellman Overview

X3DH establishes a shared secret key between two parties who mutually authenticate each other based on public keys. X3DH provides forward secrecy and cryptographic deniability.

This choice was also due to the fact that X3DH is designed for asynchronous settings where one user ("Bob") is offline but has published some information to a server. Another user ("Alice") wants to use that information to send encrypted data to Bob, and also establish a shared secret key for future communication.

Before exchanging keys with other users we first need to establish a secure connection between the user and the server so that the server can distribute the correct keys to whoever asks for them. This is done using X3DH between each user and the server we know that this process is secure because the public key of the server is hard-coded and a simple hash of the program ensures integrity.

Once a shared key is established the user can request for any key bundle containing all the necessary keys to start a chat with another user. Now the user can start the establishment of a shared secrete with another user passing through the server. After the shared key is established the two parties will use the Double Ratchet to send and receive encrypted messages.

1.2.2 Double Ratchet Overview

The parties derive new keys for every message using **Double Ratchet**, so that earlier keys cannot be calculated from later ones. The parties also send Diffie-Hellman public values attached to their messages. The results of Diffie-Hellman calculations are mixed into the derived keys so that later keys cannot be calculated from earlier ones. These properties give some protection to earlier or later encrypted messages in case of a compromise of a party's keys.

Now that keys are derived we can send each of the messages encrypted using AES-GCM, an AEAD algorithm, which gives both CCA security and CI. Thanks to all of these algorithms we can have end-to-end encryption and ensure both confidentiality and integrity.

Chapter 2

Requirements

TRUST needs to follow a list of both **functional** requirements and **security** requirements, these properties are non negotiables, we implemented the application using these requirements as guidelines that helped us in choosing algorithms.

2.1 Functional Requirements

This section defines the functional requirements of the system, outlining the core features and expected behaviors necessary for its operation. For this project we can split them further into two sub sets **server functional requirements** and **client functional requirements**.

2.1.1 Server Functional Requirements

The server must fulfill the following functional requirements:

- **Manage user registration**
- **Distribute public keys of registered users**
- **Relay encrypted messages to their intended recipients**

The server cannot access the content of user messages; it can only identify the sender and recipient.

Client Functional Requirements

The client application, on the other hand, must meet the following requirements:

- **Offer a user-friendly TUI**
- **Allow users to register in the system**
- **Enable users to add others to their contact list**
- **Support message exchange between users**

This set of requirements should be enough to have a functional chat application, by adding the **security requirements** the application will also have: security of data in transit with end-to-end-encryption (E2EE).

2.2 Security Requirements

In addition to the functional requirements, we have established the following security requirements:

- **Server Authentication**: Clients must be able to verify the authenticity of the server.
- **Client-Server Confidentiality**: Clients must be able to establish a shared secret with the server.
- **End-to-End Confidentiality**: Clients must be able to establish a shared secret with other clients.
- **End-to-End Integrity**: Clients must use shared secrets to encrypt, decrypt, and verify the integrity of messages exchanged with both the server and other clients.
- **Forward Secrecy**: The compromise of a key must not affect the confidentiality of data exchanged previously.
- **Self Healing (Post Compromise Security)**: even if an adversary temporarily compromises a user's device and obtains current session keys, future messages will remain secure as long as the device regains control and resumes proper ratcheting.

Chapter 3

Technical Details

This section provides an in-depth overview of the architecture, implementation, and key components of TRUST. It covers the cryptographic protocols used, system architecture, core libraries, and design decisions that ensure security and usability.

The application is built using Rust's asynchronous runtime (`tokio`) and the framework for networking is `tokio-tungstenite`, while the user interface is implemented with `ratatui`. X3DH and Double Ratchet were implemented using primitives provided by the `dalek-cryptography` library, and AES-GCM using primitives provided by the `aes_gcm` library, ensuring end-to-end encryption for user messages.

3.1 Architecture

The application follows a client-server architecture with some modifications to enhance flexibility. By leveraging **WebSockets**, which provide bidirectional TCP communication, both the client and server can initiate requests. This approach simplifies development by enabling real-time message exchange without requiring traditional request-response cycles, making the system more efficient and responsive. To better understand the application's functionality, we outline four key scenarios that define its core operations:

1. **Establishment of a Secure Connection**: The client and server perform a handshake to initiate a secure, encrypted communication channel.

2. **User Registration**: New users send their public keys and register with the server while ensuring their identity remains protected.

3. **Adding a Friend**: Users exchange public keys and establish a secure communication channel for encrypted messaging.

4. **Sending Messages**: Messages are encrypted, transmitted over WebSockets, and decrypted on the recipient's side, ensuring end-to-end security.

Each of these steps plays a crucial role in maintaining the privacy and integrity of user interactions. The following sections describe these scenarios in detail.

3.1.1 Establishment of a Secure Connection

The secure connection process begins with a standard **WebSocket handshake** (Figure 3.1) between the client and server. Once the bidirectional channel is established, the client transmits its **Pre-Key Bundle**, which the server processes to:

- Derive the shared key used to encrypt communications.
- Generate the **Server Initial Message (SIM)**, which is then sent back to the client.

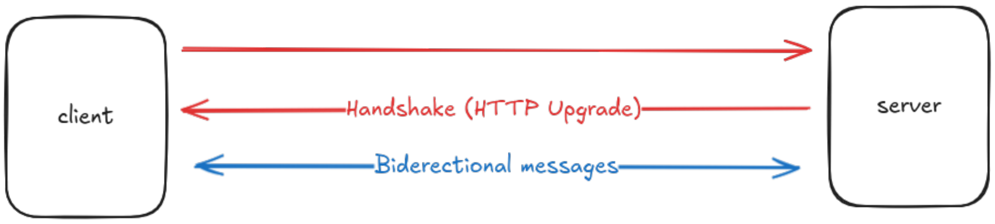


Figure 3.1: WebSocket Handshake

Upon receiving the SIM, the client verifies the server's identity by comparing the known **server public identity key** with the one provided in the message. This prevents **Man-in-the-Middle (MitM) attacks**. If authentication is successful, the client processes the SIM and derives the shared key. From this point forward, all communication between the client and server will be encrypted.

3.1.2 User Registration

Once a secure connection with the server is established, the user can register by choosing a **username** and sending a **registration request** to the server. The server then verifies whether the chosen username is unique.

- If the username is **available**, the server sends a **confirmation response** to the client.

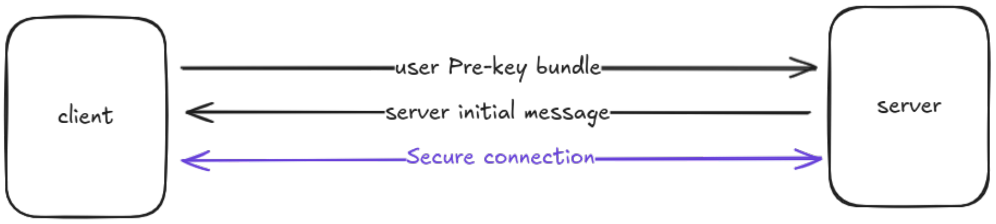


Figure 3.2: Secure connection establishment

- If the username is **already taken**, the server responds with a `"Conflict"` status code and the message `"User Already Exists"`, prompting the user to choose a different username.

```
{
    "action":"register",
    "username":"xyz",
    "bundle:"dGhpcyBpcyB0aGUgdXNlciBwcmVrZXkgYnVuZGxluIGJ2U2
      NA=="
}
```

Listing 3.1: Example of a Registration request

3.1.3 Adding a Friend

Once the user is registered in the system, they can add a new friend by submitting a **get user pre-key bundle request** to the server, specifying the username of the desired contact. The server then checks whether the requested user is registered in the application.

- If the user **exists**, the server responds with the requested user's **Pre-Key Bundle**.
- If the user **is not found**, the server returns a `"UserNotFound"` error response with `status code` 404.

3.1.4 Sending Messages

Upon receiving the requested user's **Pre-Key Bundle**, the user derives the **shared key** and generates the **initial message**. The user then sends a `send_message` request to the recipient, embedding the initial message in the `"text"` field. The request follows the JSON format below:

```json
{
  "action": "send_message",
  "msg_type": "initial_message",
  "to": "alice",
  "from": "bob",
  "text": "dGhpcyBpcyB0aGXNlciBpbml0aWFsIG1lc3NhZ2UgaW4
     gYmFzZTY0",
  "timestamp": "2025-02-06T14:21:21+00:00"
}
```

Listing 3.2: Initial Message Request Format

The user who will receive the initial message will derive the shared key and then add the sender as a contact in their contacts list.

3.2 Implementation

This section will provide a comprehensive explanation of the implementation of each key component of the application. Each aspect of the application, from the underlying security mechanisms to the user-facing interface, is discussed in detail to give a clear understanding of the technical choices and how they come together to ensure a seamless and secure user experience.

3.2.1 Key Generation for X3DH

The function responsible for generating **Pre-Key Bundles** is `generate_prekey_bundle()`. This function creates the **Private Identity Key**, **Private Signed Pre-Key**, and their corresponding **Public Keys**. Additionally, it returns the **Pre-Key Bundle** along with the **private keys** required for secure communication.

To prevent **replay attacks**, **One-Time Pre-Keys (OTPKs)** are incorporated into the key generation process. For this purpose, a specialized function, `generate_prekey_bundle_with_otpk()`, is introduced. This function takes as input the number of **one-time pre-keys** to generate and returns a tuple containing:

- The **Pre-Key Bundle**,
- The **Private Identity Key**,
- The **Private Signed Pre-Key**,

- A vector of **private One-Time Pre-Keys**.

This approach ensures that each session can maintain **forward secrecy** while mitigating potential security risks such as **key reuse and replay attacks**.

3.2.2 Pre-Key Bundle Processing

The function responsible for processing **Pre-Key Bundles** is `process_prekey_bundle()`, which takes as input the received bundle and the receiver's **Private Identity Key**.

The function first verifies the signature contained in the bundle to ensure its authenticity. After validation, it performs the **X3DH (Extended Triple Diffie-Hellman) key agreement protocol** to derive the shared secret. This is done by passing the secret obtained from the three Diffie-Hellman exchanges through a **Key Derivation Function (KDF)** to generate the final 32-byte shared key.

Once the shared key is established, an **Initial Message (IM)** is generated, containing the following components:

- The receiver's **Public Identity Key**;
- The receiver's **Public Ephemeral Key**;
- The sender's **Public Signed Pre-Key hash** (extracted from the bundle);
- The sender's **Public One-Time Pre-Key (OTPK) hash**;
- **Associated Data**, which includes the Public Identity Keys of both the sender and receiver.

The function then returns the shared key and the initial message.

3.2.3 Initial Message Processing

The functions responsible for processing the **Initial Message** are respectively `process_server_initial_message()` and `process_initial_message()`. These two functions are equivalent, except that the former additionally verifies whether the **Server's Public Identity Key** provided as input matches the one included in the Initial Message.

Both functions take as input the received Initial Message, the receiver's **Private Identity Key**, **Private Signed Key**, and **Private One-Time Pre-Key (OTPK)**. They compute the three Diffie-Hellman (3DH) secrets and pass the resulting value through a **Key Derivation Function (KDF)** to derive the 32-byte shared key. Finally, they return the computed shared key.

3.2.4 AES Encryption

The function responsible for AES encryption is `encrypt()`. This function takes as input the encryption key, the data to be encrypted, and the **Associated Data (AD)**.

First, it generates a nonce using the Cryptographically Secure Pseudo Random Number Generator (CSPRNG) provided by the `rand` crate in Rust (OsRng). Then, it constructs the payload and performs AES-GCM encryption. Finally, it concatenates the nonce, the associated data, and the ciphertext before returning the result as a **Base64-encoded string**.

3.2.5 AES Decryption

The function responsible for AES decryption is `decrypt()`. It takes as input the decryption key, the cipher text to be decrypted, and the **Associated Data (AD)** and the **Nonce**.

This function checks for integrity and then performs the inverse operation of the encryption process if the message has not been tampered with. Finally, it returns the decrypted data as a **byte vector**.

Server

The server implementation is built on an asynchronous runtime provided by the `tokio` library in Rust. This library enables the server to perform non-blocking asynchronous operations, improving both flexibility and performance.

Each new connection is handled by a separate Tokio task, allowing the server to efficiently manage multiple concurrent connections. For each **Connection task**, two additional Tokio tasks are generated:

- the first task (`task_receiver`) continuously listen for incoming requests;
- the second task (`task_sender`) forwards messages to the client;

3.2.6 Task Receiver

This task continuously listens for incoming requests on the WebSocket connection from the client. It exposes four API endpoints:

- `establish_connection`: Used to establish a secure connection between the client and the server. This API endpoint is handled by the function `handle_establish_connection()`, which first verifies whether the Pre-Key bundle is correctly formatted. If the verification is successful, the function calls `process_key_bundle()` to generate the shared secret; otherwise, it returns an error.

Once the shared secret is derived, it is stored in a shared variable called `session`, allowing both tasks to access it. This endpoint is only accessible if no shared secret exists, meaning that a secure connection has not yet been established.

- `register`: Accessible once the secure connection is established, allowing the client to register within the application. This API endpoint is handled by the function `handle_registration_request()`, which first performs the necessary username validation (ensuring it is alphanumeric and not already in use). It then verifies whether the Pre-Key bundle is well-formed. If all checks pass, the user is registered in the system and it sends back a confirmation response.

- `get_user_bundle`: Accessible after the secure connection is established, enabling the client to retrieve a user's key bundle. This API endpoint is handled by the function `handle_get_bundle_request()`, which verifies whether the requested user is registered in the system. If the user exists, the function generates a response containing the corresponding key bundle; otherwise, it returns a `User Not Found` error.

- `send_message`: Available once the secure connection is established, allowing the client to exchange messages with other users. This API endpoint is handled directly by the task. It first checks whether the recipient is registered (and thus connected) to the system. If the recipient is online, the task forwards the message to the recipient's sender task, which then delivers it to the recipient's client.

All API requests received after the secure connection is established are decrypted upon arrival, and all responses are encrypted before being sent to the client.

3.2.7 Task Sender

The Task Sender is responsible for message forwarding. It continuously listens on a `mpsc channel` for incoming messages. Upon receiving a message, it encrypts the content using the session key shared between the recipient client and the server, ensuring secure communication. Once the message is encrypted, it forwards the encrypted message to the recipient.

The content of messages is encrypted using the shared key between the two clients. The `send_message` request undergoes two layers of encryption: first, the message text is encrypted with the shared key between the clients, and then the entire request is encrypted using the shared key between the client and the server, ensuring end-to-end security.

3.2.8 Client & TUI

The client backend also utilizes `tokio` for asynchronous runtime. When a new client is created (i.e., when the TUI program is started), it attempts to establish a secure connection with the server. If successful, it spawns a Tokio listener task, which continuously listens for incoming responses or new messages.

To manage the correlation between requests and responses, we introduced a struct called `RequestWrapper`. This struct contains two fields: `request_uuid`, which uniquely identifies the request, and `body`, which holds the actual request to be sent to the server. When the server responds, it includes the same `request_uuid` in the `ResponseWrapper`, allowing the client to match each response to its corresponding request.

Chat messages, on the other hand, are handled as regular `send_message` requests. When the client receives a message, it forwards it to a task running in the TUI binary, which listens for incoming messages on a `mpsc channel`. This task then invokes the appropriate handler to process the message and render it in the user interface.

3.3 Code Structure

The codebase is organized into five separate Cargo projects, each serving a distinct purpose:

- **Protocol**: A library that implements the cryptographic protocols used in the system, including X3DH and AES-GCM.
- **Common**: A shared library providing utility functions used by both the client and server.
- **Server**: Contains the server binary along with all necessary utility functions for server-side operations.
- **Client**: A library that facilitates client interactions with the server.
- **Tui**: Includes the TUI application binary, along with all files required for the user interface and front-end logic.

3.4 Dependencies

The implementation of **TRUST** application relies on several external Rust crates that provide critical functionalities such as asynchronous networking, cryptographic operations, and terminal user interface (TUI) rendering. All dependencies used in this project are well-known, actively maintained, and considered de facto standards in their respective domains, ensuring reliability, security, and long-term support. Below is an overview of the main dependencies.

3.4.1 Networking and Asynchronous Execution

The list of dependencies used for the asynchronous execution is:

- `tokio` **(1.42.0)**: Provides the asynchronous runtime used throughout the application to handle concurrent tasks efficiently, enabling non-blocking communication between clients and the server.
- `tokio-tungstenite` **(0.26.1)**: A WebSocket library that integrates with Tokio, allowing real-time, bidirectional communication over WebSockets.
- `futures` **(0.3.31) &** `futures-util` **(0.3.31)**: Provide abstractions for asynchronous programming, including streams and futures for handling asynchronous events.
- `tokio-stream` **(0.1.17)**: Enhances stream handling within Tokio-based applications, making it easier to process asynchronous data flows.

3.4.2 Cryptography and Secure Communication

The list of dependencies used for cryptographic purposes on the other hand is:

- `aes` **(0.8.4) &** `aes-gcm` **(0.10.3)**: Provide authenticated encryption for securing messages using the AES-GCM encryption scheme.
- `rand` **(0.8.5)**: A cryptographically secure random number generator, essential for generating nonces securely.
- `ed25519-dalek` **(2.1.1)**: Implements the Ed25519 signature scheme, ensuring authentication and integrity of messages.
- `x25519-dalek` **(2.0.1)**: Used for implementing the X3DH key exchange protocol, allowing secure key agreement between clients.
- `curve25519-dalek` **(4.1.3)**: Provides elliptic curve operations, specifically for Diffie-Hellman key exchange and digital signatures.
- `sha2` **(0.10.8)**: Implements the SHA-2 family of cryptographic hash functions, ensuring data integrity and secure hashing of credentials.

- `hkdf` **(0.12.4)**: Implements the HMAC-based Key Derivation Function (HKDF) used to derive cryptographic keys securely.
- `zeroize` **(1.8.1)**: Ensures that sensitive cryptographic data is securely erased from memory when no longer needed.

3.4.3 Data Serialization and Parsing

The dependencies used for data serialization is:

- `serde` **(1.0.216)** & `serde_json` **(1.0.137)**: Used for serializing and deserializing data exchanged between the client and server.
- `base64` **(0.22.1)**: Handles encoding and decoding of binary data, particularly for securely transmitting encrypted messages.

3.4.4 Terminal User Interface (TUI)

For the TUI we chose the following dependencies:

- `ratatui` **(0.29.0)**: A Rust TUI library used to create the command-line interface for the chat application.
- `crossterm` **(0.28.1)**: Provides cross-platform support for handling terminal input and output, including event handling and text rendering.

3.4.5 Utilities

Finally some extra packeges chosen for extra utilities are:

- `uuid` **(1.11.0)**: Used for generating unique request identifiers, allowing the system to match server responses with client requests.
- `arrayref` **(0.3.9)**: Provides utilities for working with fixed-size arrays, useful in cryptographic operations.

These dependencies collectively enable our application to provide secure, efficient, and user-friendly encrypted messaging.

Chapter 4

Security Considerations

In this chapter we are going to discuss in more details the cryptographic principles that ensure security. Firstly we are going to analyze X3DH and later the double ratchet mechanism. But before starting we need to define some preliminaries used in each of the two algorithms.

4.1 Preliminaries

As explained above an application using X3DH must decide on several parameters:

Name	Definition
curve	X25519 or X448
hash	A 256 or 512-bit hash function (e.g. SHA-256 or SHA-512)
info	An ASCII string identifying the application

In our protocol we used **X25519**, as hash **SHA**. An application must additionally define an encoding function Encode(PK) to encode an X25519 or X448 public key PK into a byte sequence. Since we decided to implement only one of the two curves there was no need to differentiate between them.

4.2 Cryptographic Notation

The used notation in this chapter is:

- The concatenation of byte sequences **X** and **Y** is **X**$\|$**Y**
- **DH(PK1,PK2)** represents a byte sequence which is the shared secret output from an Elliptic Curve Diffie-Hellman function involving the key pairs represented by public keys $PK1$ and $PK2$. The Elliptic Curve Diffie-Hellman function will be the $X25519$.
- **Sig(PK,M)** represents a byte sequence that is an **XEdDSA** signature on the byte sequence M and verifies with public key PK, and which was created by signing M with PK's corresponding private key.
- **KDF(KM)** represents 32 bytes of output from the $HKDF$ algorithm with inputs:
  - $HKDF\ input\ key\ material = F\|KM$, where KM is an input byte sequence containing secret key material, and F is a byte sequence containing 32 $0xFF$. F is used for cryptographic domain separation with $XEdDSA$.
  - $HKDF\ salt =$ a zero-filled byte sequence with length equal to the $hash$ output length.
  - $HKDF\ info =$ the $info$ parameter.

4.3 Roles

The X3DH protocol involves theree parties: **Alice, Bob** and a **server**.

- **Alice** wants to send to Bob some initial data using encryption, and also establish a shared secret key.
- **Bob** wants to allow parties like Alice to establish a shared key with him and send encrypted data. However, Bob might be offline when Alice attempts to do this. To enable this, Bob has a relationship with some server.
- **The server** can store messages from Alice to Bob which Bob can later retrieve. The server also lets Bob publish some data which the server will provide to parties like Alice.

4.4 Keys

X3DH used the following elliptic curve keys:

Name	Definition
IK_A	Alice's identity key
EK_A	Alice's ephemeral key
IK_B	Bob's identity key
SPK_B	Bob's signed prekey
OPK_B	Bob's one-time prekey

All public keys have a corresponding private key, but to simplify description we will focus on the public keys. Each party has a long-term identity public key (IK_A for Alice, IK_B for Bob). Bob also has a signed prekey SPK_B, which he will change periodically, and a set of one-time prekeys OPK_B, which are each used in a single X3DH protocol run. During each protocol run, Alice generates a new ephemeral key pair with pubic key EK_A. After a successful protocol run Alice and Bob will share a 32-byte secret key SK that later will be used a root key for the double ratchet.

4.5 The X3DH protocol

X3DH has three pahses:

1. Bob publishes his identity key and prekeys to a server.
2. Alice fetches a prekey bundle from the server, and uses it to send an initial message to Bob.
3. Bob receives and processes Alice's initial message.

4.6 Publishing Keys

Bob publishes a set of elliptic curve public keys to the server, containing:

- 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 $(OPK_B^1, OPK_B^2, OPK_B^3, ...)$

Bob only needs to upload his identity key to the server once. However, Bob may upload new one-time prekeys at other times. Bob will also upload a new signed prekey and prekey signature at some interval. The new signed prekey and prekey signature will replace the previous values.

4.7 Sending the Initial Message

To perform an X3DH key agreement with Bob, Alice contacts the server and fetches a prekey bundle containing the following values:

- Bob's identity key IK_B
- Bob's signed prekey SPK_B
- Bob's prekey signature $Sig(IK_B, Encode(SPK_B))$
- Bob's one-time prekey OPK_B

The serve provides one of Bob's one-time prekeys, and then delete it. Alice verifies the prekey signature and aborts the protocol if verification fails. Alice then generates an ephemeral key pair with public key EK_A.

Alice calculates:

- $DH1 = DH(IK_A, SPK_B)$
- $DH2 = DH(EK_A, IK_B)$
- $DH3 = DH(EK_A, SPK_B)$
- $DH4 = DH(EK_A, OPK_B)$
- $SK = KDF(DH1 \| DH2 \| DH3 \| DH4)$

Note that $DH1$ and $DH2$ provide mutual authentication, while $DH3$ and $DH4$ provide forward secrecy. After calculating SK, Alice deletes her ephemeral private key and the DH outputs. Alice then calculates an AD byte sequence that contains identity information for both parties: $AD = Encode(IK_A) \| Encode(IK_B)$.

Alice then sends Bob an initial message containing:

- Alice's identity key IK_A

- Alice's ephemeral key EK_A
- Identifiers starting which of Bob's prekeys Alice used
- An initial ciphertext encrypted with AES-GCM using AD as associated data.

This first ciphertext is used both as the first message within the double ratchet, and as part of Alice's X3DH initial message.

4.8 Receiving the Initial Message

Upon receiving Alice's initial message, Bob retrieves Alice's identity key and ephemeral key from the message. Bob also loads his identity private key, and the private key(s) corresponding to whichever signed prekey and one-time prekey Alice used.

Using these keys, Bob repeats the DH and KDF calculations from the previous section to derive SK, and then deletes the DH values. Bob then constructs the AD byte sequence using IK_A and IK_B, as described in the previous section. Finally, Bob attempts to decrypt the initial ciphertext using SK and AD. If the initial ciphertext fails to decrypt, then Bob aborts the protocol and deletes SK. If the initial ciphertext decrypts successfully the protocol is complete for Bob. Bob deletes any one-time prekey private key that was used, for forward secrecy.

4.9 Double Ratchet

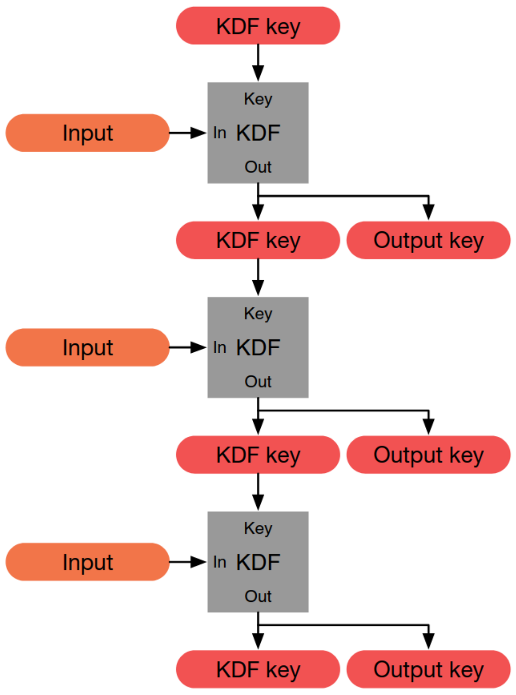
4.9.1 KDF Chains

We define a **KDF** as a cryptographic function that takes a secret and random **KDF key** and some input data and returns output data. The output data is indistinguishable from random provided the key isn't known. If the key is not secret and random, the KDF should still provide a secure cryptographic hash of its key and input data. The HMAC and HKDF construction, we chose has a secure hash algorithm, as it meets the KDF definition.

The term **KDF chain** is used when some of the output from KDF is used as an **output key** and some is used to replace the KDF key, which can then be used with another input.

A KDF chain has the following properties:

- **Resilience:** The output keys appear random to an adversary without knowledge of the KDF keys. This is true even if the adversary can control the KDF inputs.



- **Forward security:** Output keys from the past appear random to an adversary who learns the KDF key at some point in time.
- **Break-in recovery:** Future output keys appear random to an adversary who learns the KDF key at some point in time, provided that future inputs have added sufficient entropy.

In a **Double ratchet session** between Alice and Bob each party stores a KDF key for three chains: a **root chain**, a **sending chain**, and a **receiving chain**.

As Alice and Bob exchange messages they also exchange new Diffie-Hellman public keys, and the Diffie-Hellman output secretes become the inputs to the root chain. The output keys from the root chain become new KDF keys for the sending and receiving chains. This is called the **Diffie-Hellman ratchet**.

The sending and receiving chains advanced as each message is sent and received. Their output keys are used to encrypt and decrypt messages. This is called the **symmetric-key ratchet**.

4.9.2 Symmetric-key ratchet

Every message sent or received is encrypted with a unique **message key**. The message keys are output keys from the sending and receiving KDF chains. The KDF keys for these chains will be called **chain keys**.

The KDF inputs for the sending and receiving chains are constant, so these chains don't provide break-in recovery. The sending and receiving chains just ensure that each message is encrypted with a unique key that can be deleted after encryption or decryption. Calculating the next chain key and message key from a given chain key is a single **ratchet step** in the **symmetric-key ratchet**.

Since message keys are not used to derive any other keys, message keys may be stored without affecting the security of other message keys. This property is useful for handling lost or out of order messages.

4.9.3 Diffie-Hellman ratchet

If an attacker steals on party's sending and receiving chain keys, the attacker can compute and decrypt all future messages. To prevent this, the Double Ratchet combines the symmetric-key ratchet with a **DH ratchet** which updates chain keys based on Diffie-Hellman outputs.

To implement the DH ratchet, each party generates a DH key pair which becomes their current **ratchet key pair**. Every message from either party begins with a header which contains the sender's current ratchet public key. When a new ratchet public key is received from the remote party, a **DH ratchet step** is performed which replaces the local party's current ratchet key pair with a new key pair.

This results in a "ping-pong" behavior as the parties take turns replacing ratchet key pairs. An eavesdropper who briefly compromises one of the parties might learn that value of a current ratchet private key, but that private key will eventually be replaced with an uncompromised one. A that point, the Diffie-Hellman calculation between ratchet key pairs will define a DH output unknown to the attacker.

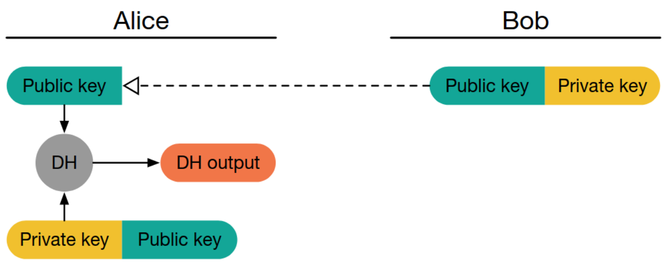


Figure 4.1

Alice is initialized with Bob's ratchet public key. Alice's ratchet public key isn't yet known to Bob. A part of initialization Alice perform a DH calculation between her ratchet private key and Bob's ratchet public key as shown in figure 4.1

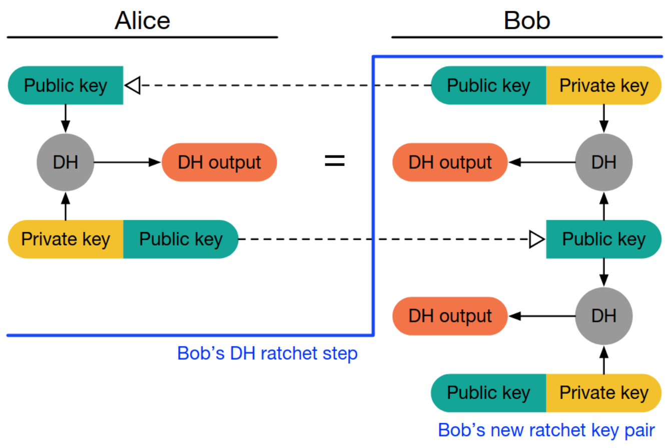


Figure 4.2: Bob's calculates his DH ratchet step

Alice's initial messages advertise her ratchet public key. Once Bob receives one of these messages, Bob performs a DH ratchet step: he calculates the DH output between Alice's ratchet public key and his ratchet private key, which equals Alice's initial DH output. Bob then replaces his ratchet key pair and calculates a new DH output like in figure 4.2

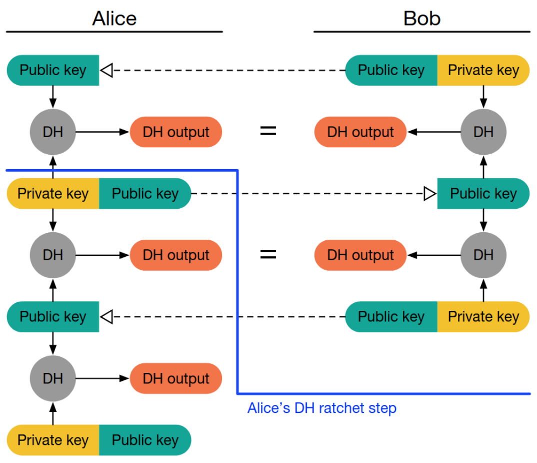


Figure 4.3

Messages sent by Bob Advertise his new public key. Eventually, Alice will receive one of Bob's messages and perform a DH ratchet step, replacing her ratchet key pair and deriving tow DH outputs, one that matches Bob's latest and a new one 4.3

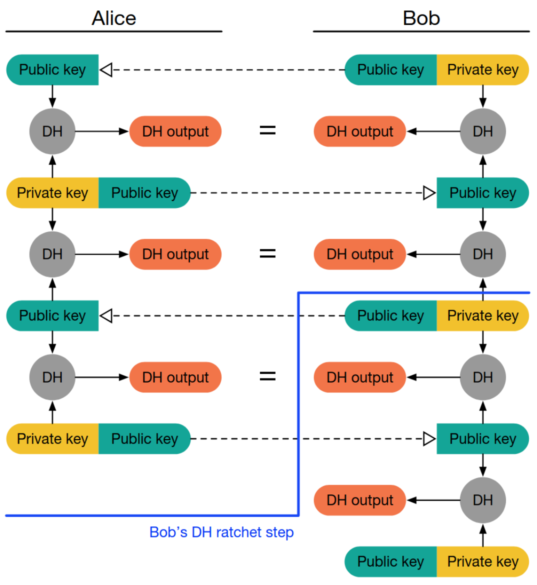


Figure 4.4

Messages sent by Alice advertise her new public key. Eventually, Bob will receive one of these messages and perform a second DH ratchet step, and so on 4.4

The DH outputs generated during each DH ratchet step are used to derive new sending and receiving keys. Bob uses his first DH output to derive a receiving chain that matches Alice's sending chain. Bob uses the second DH output to derive a new sending chain. As the parties take turns performing DH ratchet steps, they take turns introducing new sending chains.

However, instead unlike the simplified pictures shown above instead of taking the chain keys directly from DH outputs, the DH outputs are used as KDF inputs to a root chain, and the KDF outputs from the root chain are used as sending and receiving chain keys. Using a KDF chain here improves resilience and break-in recovery. A full DH ratchet step consists of updating the root KDF chain twice, and using the KDF output keys as new receiving and sending chain keys:

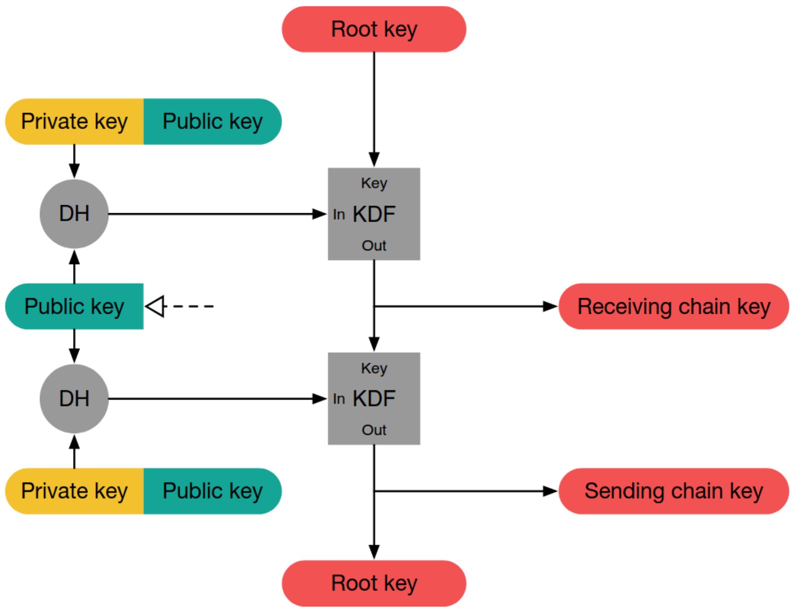


Figure 4.5

4.9.4 Combining both Ratchets

Combining the symmetric-key and DH ratchets gives the Double Ratchet:

- When a message is sent or received, a symmetric-key ratchet step is applied to the sending or receiving chain to derive the message key.

- When a new ratchet public key is received a DH ratchet step is per formed prior to the symmetric-key ratchet to replace the chain keys.

In figure 4.6 Alice has been initialized with Bob's ratchet public key and a shared secret which is the initial root key. As part of initialization Alice generates a new ratchet key pair, and feeds the DH output to the root KDF to calculate a new root key (RK) and sending chain key (CK):

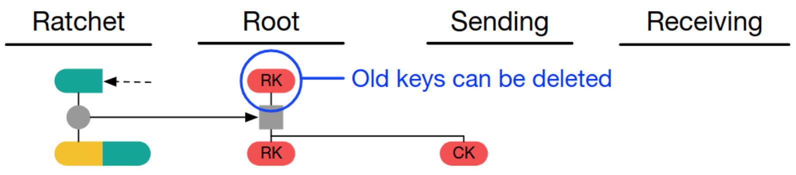


Figure 4.6

When Alice sends her first message $A1$, she applies a symmetric-key ratchet step to her sending chain key, resulting in a new message key. The new chain key is stored, but the message key and old chain key can be deleted 4.7

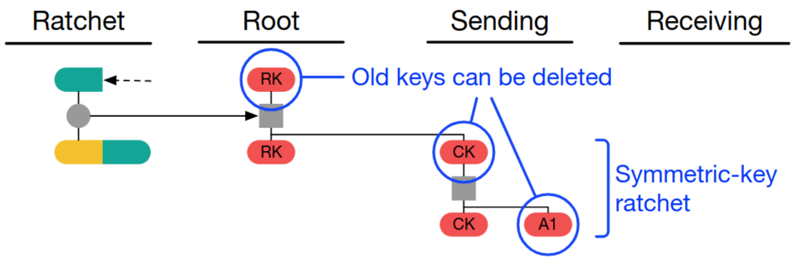


Figure 4.7

If Alice next receives a response $B1$ from Bob, it will contain a new ratchet public key. Alice applies a DH ratchet step to derive new receiving and sending chain keys. Then she applies a symmetric-key ratchet step to the receiving chain to get the message key for the received message 4.8

Suppose Alice next sends a message $A2$, receives a message $B2$ with Bob's old ratchet public key, then sends messages $A3$ and $A4$. Alice's sending chain will ratchet three steps, and her receiving chain will ratchet once 4.9

Suppose Alice then receives messages $B3$ and $B4$ with Bob's next ratchet key, then sends a message $A5$. Alice's final state will be as follows 4.10

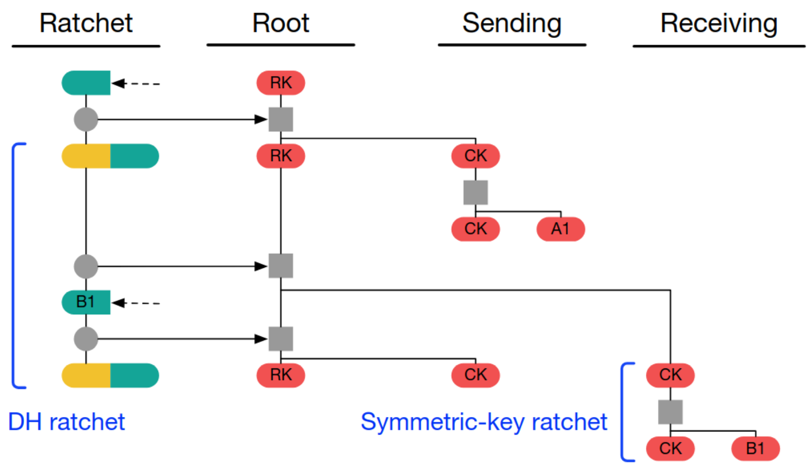


Figure 4.8

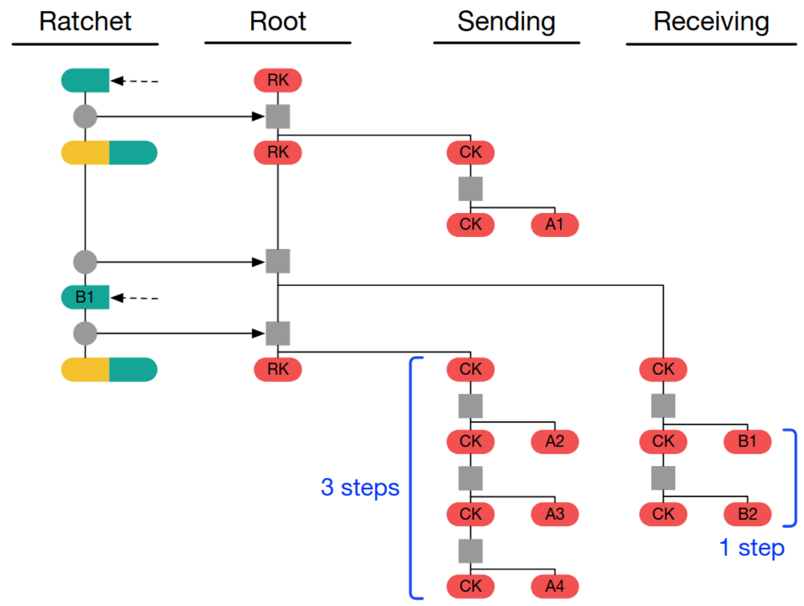


Figure 4.9

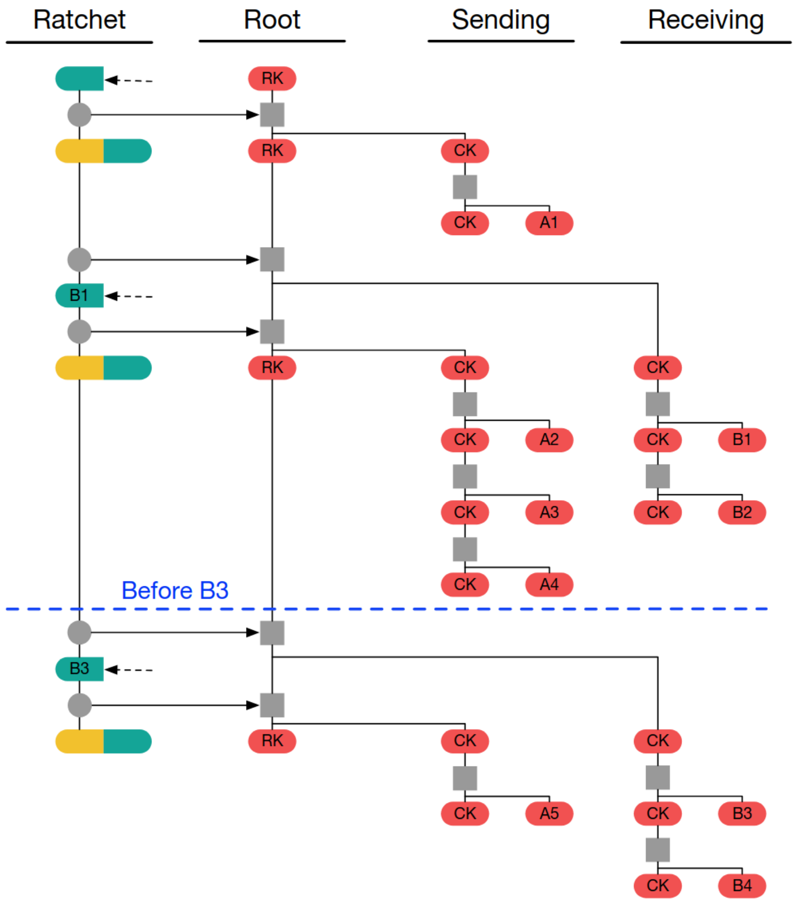


Figure 4.10

4.9.5 Out-of-order messages

The Double Ratchet handles lost or out-of-order messages by including in each message header the message's number in the sending chain and the length in the previous sending chain PN. This enables the recipient to advance to the relevant message key while storing skipped message keys in case the skipped messages arrive later.

On receiving a message, if a DH ratchet step is triggered then the received PN minus the length of the current receiving chain is the number of skipped messages. The received N is the number of skipped messages in the new receiving chain.

If a DH ratchet step is not triggered, then the received N minus the length of the receiving chain is the number of skipped messages in that chain.

For example, consider the message sequence from the previous section 4.10 when messages $B2$ and $B3$ are skipped. Message $B4$ will trigger Alice's DH ratchet step (instead of $B3$). Message $B4$ will have $PN = 2$ and $N = 1$. On receiving $B4$ Alice will have a receiving chain of length 1 ($B1$), so Alice will store message keys for $B2$ and $B3$, so they can be decrypted if they arrive later as it is shown in figure 4.11:

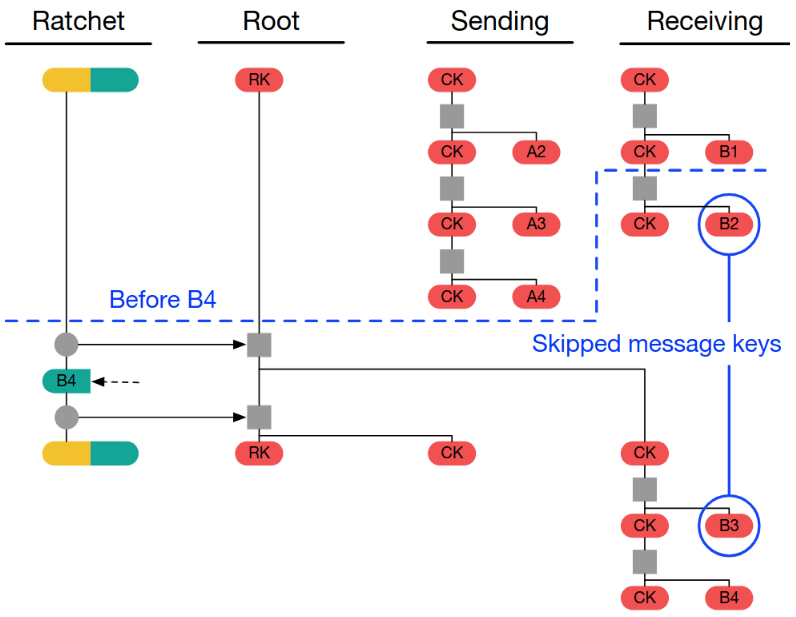


Figure 4.11

4.10 Integration with X3DH

The outputs of the X3DH are used by the Double Ratchet:

- The SK output from X3DH becomes the SK input to Double ratchet initialization.
- The AD output from X3DH becomes the AD input to Double Ratchet encryption and decryption.
- Bob's signed prekey form X3DH (SPK_B) becomes Bob's initial ratchet public key for double ratchet initialization.

4.11 Security Considerations

4.11.1 Authentication

Before or after an X3DH key agreement, the parties may compare their identity public keys IK_A and IK_B through some authenticated channel. For example, they may compare public key fingerprints manually, or by scanning a QR code.

During the designing phase of our project we assumed that the server could not be impersonated by any malicious attacker and can not be tampered with thus making sure that what Alice or Bob get from the server are the correct keys.

4.11.2 Protocol Replay

One of the potential issues of X3DH is the possibility of a replay attack, but this is no possible thanks to the implementation of the one-time prekys. In Signal's documentation other proposed mitigations are: use a ratchet mechanism, keep a blacklist of observed messages, or replace old signed prekeys more rapidly.

4.11.3 Deniability

X3DH doesn't give either Alice or Bob a publishable cryptographic proof of the contents of their communication or the fact that they communicated.

Like in the OTR protocol [3], in some cases a third party that has compromised legitimate private keys from Alice or Bob could be provided a communication transcript that appears to be between Alice and Bob and that can only have been created by some other party that also has access to legitimate private keys from Alice or Bob.

4.11.4 Key Compromise

Compromise of a party's private keys has a huge impact on security, though the use of ephemeral keys and prekeys provides some mitigation.

Compromise of a party's identity private key allows impersonation of that party to others. Knowledge of a party's preky private keys may affect the security of older or newer SK values, depending on many factors.

Since we implemented one-time prekeys then the compromise of identity keys and preky private keys at some future time will not compromise older SK since the OPK_B was deleted. Moreover, we also implemented double ratchet which replaces SK with new keys to provide fresh forward secrecy.

4.11.5 Server Trust

A malicious server could cause communication between Alice and Bob to fail for example by refusing to deliver messages.

The other possible issue is the inability to check in person keys, but a system similar to the one implemented by many Linux distros should suffice as integrity check for the key of the server. Once we have established a root of trust we can rely on the checks implemented by the protocol.

Another final possible attack is due to the limited number of OTK if an attacker drains another party's one-time prekys, making them unable to create new connections. A system to replenish one-time prekys should be implemented, and the server should limit the rate keys are requested.

4.11.6 Secure Deletion

The double ratchet algorithm is designed to provide security against and attacker who records encrypted messages and then compromises the sender or receiver at a later time. This security could be deleted if deleted plaintext or keys could be recovered by an attacker with low-level access to the compromised device. But the main focus of this project is security of data in transit.

4.11.7 Recovery from Compromise

The DH ratchet is designed to recover security against a passive eavesdropper who observes encrypted messages after compromising the parties to a session. Compromise of secrete keys will:

- The attacker could use the compromised keys to impersonate the compromised party.
- The attacker could substitute her own ratchet keys via continuous active MitM attack, to maintain eavesdropper on the compromised session.
- The attacker could modify a compromised party's RNG so that future ratchet private keys are predictable.

If a party suspects its keys or devices have been compromised, it must replace them immediately.

4.11.8 Cryptanalysis and Ratchet Public Keys

Because all DH ratchet computations are mixed into the root key, an attacker who can decrypt a session with passive cryptanalysis, an attacker who can decrypt a session with passive cryptanalysis might lose this ability if she fails to observe some ratchet public keys.

But in our implementation cryptanalysis should not be an issue due to the use of strong encryption function like AES-GCM.

4.11.9 Deletion of Skipped Message Keys

Storing skipped message keys introduces some risks:

- A malicious sender could induce recipients to store large numbers of skipped message keys, possibly causing DoS due to consuming storage space.
- The lost messages may have been seen by an attacker, even though they did not reach the recipient. The attacker can compromise the intended recipient at a later time to retrieve the skipped message keys.

The implemented mitigation is limiting the number of skipped messages to 1000. For the second one we created a set timer after which keys are deleted as proposed by the Signal documentation.

Chapter 5

Known Limitations

While the system is designed to provide a secure and efficient messaging platform, there are several limitations arising from design choices, cryptographic constraints, time constraints, and practical implementation trade-offs.

5.1 Limited Number of OTP

One of the most significant limitations is that once a user exhausts their **One-Time Pre-Keys**, they cannot upload more to the server, effectively limiting the number of sessions (and thus contacts) a user can have.

5.2 Lack of Persistent Database

The absence of a database means that the system is not scalable, as all data are stored in RAM, which limits its long-term viability. Another limitation is that when a user logs out, all session data and keys are discarded, so upon logging back in—even with the same username—the user cannot retrieve past messages.

5.3 Check for Crashes

Since we did not implement heart-beat system, the server has no way to verify if a client has crashed or is online, this could lead to messages towards disconnected users, thus making the message sent unreceived even if there is no indication of that behavior neither from the server nor the recipient. But this goes beyond the scope of this project.

5.4 Side Channels

While the system relies on secure, well-established libraries, it does not address potential side-channel attacks, which remain an unaccounted-for vulnerability. One example are time based side channels due to the complexity of the primitives used or known vulnerabilities like power consumption at decryption time.

5.5 Further Improvements

Further improvements, other that fixing the problems already mentioned, could involve the implementation of **Encrypted Header of Double Ratchet** making impossible for an attacker to know the sender or recipient of a message.

One final possible addition is the implementation of **Post-Quantum Extended Diffie-Hellman** (PQXDH) to give us post-quantum security. The DH input of the double ratchet needs to be changed as well, as for the encryption system doubling the keys size should be enough. One final possible change is the use of **XChaChaPoly** which is as secure as AES-GCM, but it is faster especially on IoT devices.

Chapter 6

Installation and Configuration

6.1 Getting Started

1. Clone the repository:

```
git clone https://github.com/christiansassi/advanced-
   programming-of-cryptographic-methods-project
```

2. Navigate to the project directory:

```
cd advanced-programming-of-cryptographic-methods-project
```

6.2 Configuration

Modify the `config.toml` file located in the `config` directory to adjust the application settings:

- `server_ip`: The IP address of the server (default: `server`). If you do not plan to use Docker (see Section 6.3.2 and/or Section 6.3.3), set this to `"127.0.0.1"` or another valid IP address.
- `server_port`: The port on which the server listens (default: `3333`).
- `log_level`: The logging level (default: `info`).

Warning: Do not modify `private_key_server` and `public_key_server`. These values are automatically generated when the server starts.

6.3 Installation

You can choose between installing the project using Docker (the default and recommended method), using the pre-built executables with Python as a launcher, or building the executables yourself. The Docker-based method is recommended, as it minimizes compatibility issues by encapsulating all dependencies within containers. The local installation is less user-friendly, especially on macOS (see the warning in Section 6.3.3), as it *may* require manually building the source files. However, all methods provide the same functionality. Choose the installation method that best fits your environment and preferences.

6.3.1 Installation with Docker (Recommended)

1. Verify that Docker is installed by running:

   ```
   docker --version
   ```

2. Build the Docker image:

   ```
   docker compose build --no-cache
   ```

3. Run the updater to generate server keys:

   ```
   docker compose run --rm updater
   ```

4. Start the server and clients:

   ```
   docker compose up server client1 client2
   ```

5. Open a new terminal and attach to the TUI of `client1`:

   ```
   docker container attach client1
   ```

6. Repeat the previous step for `client2`, if desired.

7. To stop the containers, press `CTRL+C` in the terminal where you ran `docker compose up`, or run `docker compose down`.

6.3.2 Local Installation from Source

1. Ensure that Rust and Cargo are installed by running:

   ```
   rustc --version
   cargo --version
   ```

2. Build the server:

   From the root of the repository:

   (a) Navigate to the `server` directory:

   ```
   cd server
   ```

   (b) Build the server source files:

   ```
   cargo build --release
   ```

3. Build the TUI:

   From the root of the repository:

   (a) Navigate to the `tui` directory:

   ```
   cd tui
   ```

   (b) Build the TUI source files:

   ```
   cargo build --release
   ```

4. You can run the compiled executables located in the `release` subdirectory of both the `server` and `tui` directories.

6.3.3 Python Launcher (No Installation Required)

Warning: With this installation method, you will use pre-built executables. If you are using macOS, please follow the instructions in Section 6.3.2, as no pre-built executable is provided. If you are using another platform (e.g., Windows or Linux) and encounter issues, you can also refer to Section 6.3.2. Once you have successfully built the executables, you can return to this section.

1. Ensure Python is installed by running:

   ```
   python --version
   ```

   There is no specific Python version or dependency requirement, as Python is only used to simplify the launching process.

2. Run the server:

   ```
   python server.py
   ```

3. Run one or more clients:

   ```
   python tui.py
   ```

Bibliography

[1] Moxie Marlinspike and Trevor Perrin. “The Double Ratchet Algorithm”. In: (2016).

[2] Moxie Marlinspike and Trevor Perrin. “The X3DH Key Agreement Protocol”. In: (2016).

[3] I. Goldberg N. Borisov and E. Brewer. “Off-the-record Communication, or, Why Not to Use PGP”. In: (2004).