UNIVERSITY OF TRENTO

Advanced Programming of Cryptographic Methods

Project Report

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

Matteo Bordignon, Christian Sassi, Alessandro Perez

February 9, 2025

Contents

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.

1.1 Organization of the Report

This document is organized as follows: Description 2 provides an overview of the application's purpose and core functionalities; Requirements 3 defines both the functional and security requirements; Technical Details 4 outlines the system's architecture, the main implementation decisions, and the code structure; Security Considerations 5 discusses cryptographic measures and potential threats; Known Limitations 6 highlights current constraints and possible improvements; and finally, Instructions for Installation and Execution 7.3.2 guides readers through setting up and running the application.

2 Description

The project focuses on developing a secure chat application that runs via a terminal user interface (TUI) in Rust. Our primary objectives are creating a user-friendly system for confidential communication and, to demonstrate the integration of various robust cryptographic methods.

2.1 Cryptographic Primitives

The main focus of the project is Extended Triple Diffie-Hellman [5](X3DH) protocol for key establishment. Once a shared key has been established there is a switch to AES-GCM [6] for message symmetric encryption. Thanks to this two algorithms the application ensures end-to-end confidentiality and integrity. The two algorithms where chosen respectively because:

- **X3DH** has the advantage over **RSA** for public key cryptography because the key is never sent over the channel.
- **AES-GCM** is an **AEAD** thus ensuring not only confidentiality but also integrity. Moreover this mode of operation ensures that the composition of **MAC** and **encryption** is done in a secure way, finally it allows us to have **Associated Data** for additional information.

2.2 Security of Data at Rest

The application is **fully ephemeral**, this means that there is no database which stores data on persistent memory, all data are stored in the Random Access Memory (**RAM**) using efficient data structures such as hash maps. We took advantage of efficient Rust memory management to minimize memory usage. This allows our application to be faster while maintaining a high degree of confidentiality. The Rust's `zeroize` library let us set to zeros the memory used to store cryptographic keys before de-allocating it eliminating some side channel attacks on the memory.

2.3 Overview of the Protocol

The protocol can be broken down into two major phases, the first is the subscription done only once by each user with the server. The second phase is establishing a secure connection between two users, ensuring end-to-end encryption.

2.3.1 Subscription Phase

Upon starting, the program generates a new user profile with the cryptographic material needed for securing communications. Then it tries to connect to the server and perform the first **X3DH** key agreement to establish a secure connection with the server. From this point on, all communications between server and client will be encrypted using **AES-GCM**, the key used is the one derived from this first **X3DH** making this communication secure. The user can choose a **nickname** and register in the application. This user name must be unique and is how other users can find each other, such as phone numbers in *Whatsapp*. In the registration request the client attach both the chosen username and its pre-key bundle composed of:

- **Identity Key** *IK*
- **Signed Prekey** *SPK*
- **Prekey signature** *Sig(IK, Enc(SPK))*
- **One-time Prekeys** $(OPK^1, OPK^2, OPK^3, ...)$

Now, the user can chat with other users by going into phase two.

2.3.2 Chatting Phase

A user can add others to their contact list if they know their nickname. To establish secure communication, the user first requests the key bundle of the intended recipient from the server. The server provides this key bundle, which contains the necessary cryptographic material for deriving session keys.
Once the key bundle is obtained, the user derives the required encryption keys using the **X3DH (Extended Triple Diffie-Hellman) key agreement protocol**. After the initial key exchange, **AES-GCM** is used for encrypting messages due to its speed and strong security guarantees.

Each encrypted message includes **Authenticated Data (AD)**, which consists of:

- The sender's **Public Identity Key**
- The receiver's **Public Identity Key**

When a message is sent:

1. The sender encrypts it using the established **AES-GCM** session key.
2. The message is then **double-encrypted**:
    - First, the text of the message is encrypted usign the key shared between the sender and the receiver.
    - Then, the whole request is encrypted using the session key shared between the sender and the **server**.
    - The server decrypts this outer layer to determine the recipient.
    - After integrity verification, the server re-encrypts the message using the session key shared between the **server and the receiver** before forwarding it.
3. Upon receiving the message, the recipient:
    - First decrypts the outer layer using their session key with the **server**.
    - Then decrypts the inner layer using their session key with the **sender**.

If this is the **first time** the recipient is receiving a message from the sender, they must request the sender's key bundle from the server. Once obtained, they can derive the necessary session keys and proceed with secure communication.

3 Requirements

TRUST needs to follow a list of requirements both **functional** and **security**, this are non negotiable properties we implemented in the application and dictated the reasons why each of the algorithms were chosen.

3.1 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**.

3.1.1 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.

3.1.2 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).

3.2 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.
- **Session-Based Forward Secrecy**: The compromise of a session key must not affect data exchanged in other sessions. Additionally, the compromise of a long-term secret must not compromise past session keys.

4 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 `tokio-tungstenite` framework for networking, while the user interface is implemented with ratatui. Secure communication is achieved through the X3DH (Extended Triple Diffie-Hellman) protocol using primitives provided by the `dalek-cryptography` library, combined with AES-GCM using primitives provided by the `aes_gcm` library, ensuring end-to-end encryption for user messages.

4.1 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:

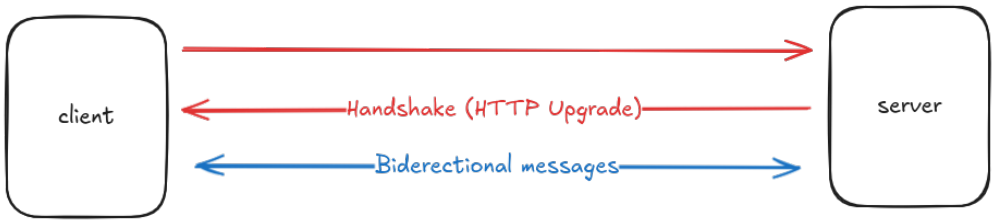


Figure 1: WebSocket Handshake

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.

4.1.1 Establishment of a Secure Connection

The secure connection process begins with a standard **WebSocket handshake** (Figure 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.

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.

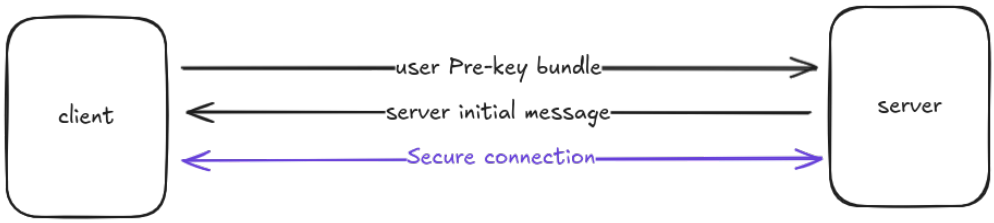


Figure 2: Secure connection establishment

4.1.2 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.
- 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.

Listing 1: Example of a Registration request

```
{
    "action":"register",
    "username":"xyz",
    "bundle:"dGhpcyBpcyB0aGUgdXNlciBwcmVrZXkgYnVuZGxlIGluIGJhc2
       U2NA=="
}
```

4.1.3 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`.

4.1.4 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:

Listing 2: Initial Message Request Format

```
{
  "action": "send_message",
  "msg_type": "initial_message",
  "to": "alice",
  "from": "bob",
  "text": "dGhpcyBpcyB0aGUgdXNlciBpbml0aWFsIG1lc3NhZ2UgaW4
     gYmFzZTY0",
  "timestamp": "2025-02-06T14:21:21+00:00"
}
```

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.

4.2 Implementation

This section will provide a comprehensive explanation of the implementation of each key component of the application. We begin with an in-depth view of the cryptographic protocol, which forms the foundation of secure communication, and then proceed to describe the development of the Server and Client architecture. 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.

4.2.1 Cryptographic Protocol

To ensure **secure end-to-end** communication we leverage on X3DH protocol combined with AES-GCM protocol, we developed the protocol on top of the primitives provided by the `dalek-cryptography` and `aes_gcm` rust libraries.

Key Generation The function responsible for generating the **Pre-Key Bundle** 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**.

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.

Initial Message Processing The functions responsible for processing the **Initial Message** are `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.

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**.

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**.

4.2.2 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;

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.

Task Sender The Task Sender is responsible for message forwarding. It continuously listens on an `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.

4.2.3 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 an `mpsc channel`. This task then invokes the appropriate handler to process the message and render it in the user interface.

4.3 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.

4.4 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.

Networking and Asynchronous Execution - `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.

Cryptography and Secure Communication - `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.

Data Serialization and Parsing - `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.

Terminal User Interface (TUI) - `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.

Utilities - `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.

5 Security Considerations

In this chapter we are going to discuss formally the underlying mathematical structures that let us have all the security requirements. We will follow the thorough analysis done in other papers [3]. In the following chapter we will explain the structures that we used.

5.1 Groups

A group is a tuple $(A, \circ)$, where A is a set and $\circ$ a binary operation, with the following properties:

- **Closed:** $\forall x, y \in A : x \circ y \in A$,
- **Associative:** $\forall x, y, z \in A : (x \circ y) \circ z = x \circ (y \circ z)$
- **Neutral element:** $\exists e \in A, \forall x \in A : x \circ e = e \circ x = x$
- **Inverse element:** $\forall x \in A, \exists a' \in A : a \circ a' = a' \circ a = e$
- **Commutative:** $\forall x, y \in A : x \circ y = y \circ x$

The commutative property is optional, however groups that satisfy this property are called abelian. A group can be infinite depending on A. The number of elements in A is called the order of the group. The order of and element $a \in A$, denoted as $ord(a)$, is the smallest positive integer n such that $a^n = e$, or $[n]a = e$ in the case of addition. Modular addition is a group with the operation $+$ which is denoted as $(\mathbb{Z}/n\mathbb{Z}, +)$.

5.2 Cyclic Groups

Let $(A, +)$ be a group and $g \in (A, +)$. Consider the set $\{[0]g, [1]g, ...ord(g) - 1\}$. This set is a group, denoted as $\langle g \rangle$:

- **Closed:** $\forall i, j \in \{0, ord(g) - 1\}$ holds that $[i]g + [j]g = [i + j \bmod ord(g)]g \in \langle g \rangle$
- **Associative:** $+$ is associative
- **Neutral Element:** $[0]g$ is the neutral element
- **Inverse Element:** $\forall i \in \{0...ord(g) - 1\}$ the inverse of $[i]g$ is $[ord(g) - i]g$

This group is called a cyclic group and g is called the generator of this group. For example, in a group $(\mathbb{Z}/q\mathbb{Z}, +)$, the generator $g = 1$. cyclic groups are abelian, and cyclic group of order q is isomorphic with $(\mathbb{Z}/q\mathbb{Z}, +)$. Isomorphic means that the groups have the same structure and properties. Having an isomorphism means that there is a mapping function from one group to the other which is a bijection. A subset $H \subset G$ of a group $(G, +)$ is called of subgroup of G, if $(H, +)$ is also a group.

5.3 Multiplicative Groups

To have a group that is based on modular multiplication $(\mathbb{Z}/q\mathbb{Z}, *)$ one has to remove some elements from $[0, ..., n - 1]$, as some elements do not have an inverse. The neutral element of a multiplicative group is 1. The only numbers that have an inverse in $(\mathbb{Z}/q\mathbb{Z}, *)$ are numbers that are coprime to n. A number x is coprime to n if the greatest common divisor is 1. In the case that n is a prime, then all elements $i \in \{1, ..., n - 1\}$ are coprime to n, so we only have to remove 0. Such modular multiplication groups with the elements that violate the group properties removed as $(\mathbb{Z}/q\mathbb{Z})^*$.

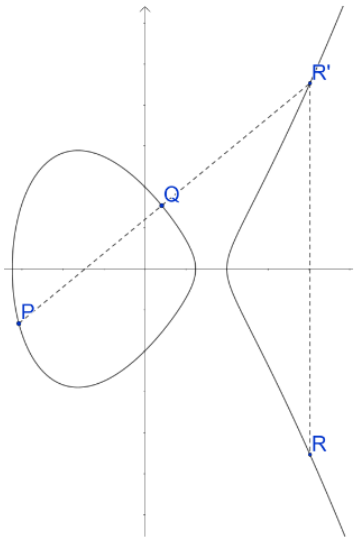
5.4 Fields

A field is a triple $(F, +, *)$, where $(F, +)$ is a group and $(F \backslash \{0\})^*$ is a group and distributivity holds. The field $(\mathbb{Z}/q\mathbb{Z}, +, *)$, where p is a prime, is called a prime field and is denoted as $GF(p)$. There is a unique prime field for each prime number.

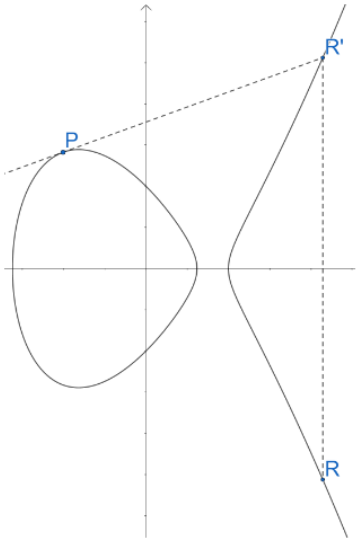
5.5 Elliptic Curves

In public-key cryptography, parties have a public key and a private key. Some public-key cryptography is based on the fact that solving discrete logarithm, is computationally difficult in certain cyclic groups. Another form of public-key cryptography is **Elliptic Curve**

Cryptography (ECC), which is based on subgroups of elliptic curves. Elliptic curves make it possible to use smaller keys compared to non elliptic curve cryptography, wile still being equally hard to break. Let $GF(q)$ be a finite field with q elements, where $q = p^m, p$ is prime and $p \neq 2, 3$. An elliptic curve over $GF(q)$ is the set of pairs (x, y) that satisfies the following equation, called the Weiestrass equation: $y^2 = x^3 + ax + b$



(a) Addition in elliptic curves



(b) doubling in elliptic curves

5.6 Elliptic Curve Group

In the this group we have addition 3a and doubling 3b as shown in the two graphs. Now we can define an ablian group $(E, +)$ of fan elliptic curve defined over a field, with a neutral element that is called point at infinity, ∞. This point is the sum of P and $-P$ and is a point on every vertical line.

Demonstration of properties

- **Closed:** By construction adding two points results in another point on the curve E
- **Associative:** The demonstration is non trivial the proof of this property and all the other can be find in this paper [2]
- **Neutral Element:** $\forall P \in E$ it hold that $P + \infty = \infty + P = P$

- **Inverse Element:** $\forall P \in E$ it hold that $P + (-P) = \infty$
- **Commutative:** We can see that visually from 3a the order in which the line is drawn does not matter, we can start either from P or Q

An elliptic curve over a field can be defined by so-called domain parameters. These domain parameters are a sextuple $T = (p, a, b, G, n, h)$, where the elements are as follows:

- p specifies the finite field $FG(p)$
- a and b specify the curve E
- $G = E(GF(p))$, which is a point on the curve over $GF(p)$, is a base point. This can be any point, but musst be agreed upon by both parties.
- $n = ord(G)$
- $h = \#E(GF(p))/n$ this is called the cofactor of the elliptic curve E

X3DH is implemented with either the Elliptic Curve Diffie-Hellman X25519 or X448 [4], for our implementation we choose the faster X25519. These curves use the Montgomery form representation of elliptic curves $By^2 = x^3 + Ax^2 + x$.

Both use a function *X25519* and *X448*, respectively, in order to do scalar multiplications on the curve. They take as an input a coordinate of a point P and a scalar n and outputs nP. In order for Alice to generate a public key in X25519, first she generates a private key a and converts it into a byte string of 32 bytes. Then her public key is $A = aG$, where G is the base point, which is the point with the minimal, positive x value in the case of these functions. Her public key will be $A + X25519(a, 9)$ where 9 is the x coordinate of the base point. Bob does the same with a private key b and computes $B/$ Now both can compute the shared secret key $K = X25519(a, B) = X25519(b, A)$.

5.7 Key Derivation Functions

In order to get a shared secret key, a Key Derivation Function (KDF) is used to derive multiple shared secret keys from a shared secret. This function uses as input a secret value and possibly known parameters. An example of accepted parameters is: a value for keying material, a length value l, an optional salt value and an optional context variable. The last two optional arguments can be set to a constant if they are not needed. The salt value is a non-secret random value that provides additional randomness to the function. Adding a salt value makes the KDF stronger, by aiming it less predictable, as using a salt make s different uses of the KDF independent of each other, provided that the KDF is strong enough. With

a salt we can even derive two different outputs from the same secret value input, which is called domain separation. The output of a KDF is a string of bits with length l which can be used in further protocols as a key, in our case AES-GCM. A KDF is a one-way function, meaning that if we know the output of a KDF, it is infeasible to calculate the input values that were used. In our implementation of X3DH we make use of a KDF, namely the HKDF algorithm.

5.8 Computational Problems

Most cryptographic schemes are based on computationally hard problems. These computationally hard problems cannot be solved efficiently, which means that if a cryptographic scheme is based on such a problem in a successful way, then breaking such a scheme could not be done efficiently either. This is done by design, because having a cryptographic scheme that takes a long time to break is desirable.

5.8.1 Discrete Logarithm

Let $\mathbb{G}$ be a cyclic group with generator g and order q. Then we know that for every $y \in \mathbb{G}$, there is a unique $x \in (\mathbb{Z}/q\mathbb{Z})^*$ such that $g^x = y$. This value x is called the discrete logarithm of y with respect to $g : x = \log_g y$. Note that x is unique in $x \in (\mathbb{Z}/q\mathbb{Z})^*$, which means that if there is a x' where $g^{x'} = y$, then $x' \equiv x \bmod q$. The discrete logarithm problem in a cyclic group $\mathbb{G}$ with generator g and order q is as follows: given a randomly chosen $y \in \mathbb{G}$, compute $\log_g y$. In case of Elliptic Curves it is as follows: consider an elliptic curve E over a finite field $GF(p)$ with p a prime and $P = (x, y)$ a point of prime order n, meaning that n is the smallest number such that $nP = \infty$, on the elliptic curve. Let k be an integer. Let E be the cyclic group generated by P. Given a point $Q = kP \in E$, it will be hard to compute k.

5.9 Random Oracles

In order to analyse the security of cryptographic protocols, a random oracle is often used. Random oracles were introduced and first used in proofs by Bellare and Rogaway. These oracles are a black box, which means we know nothing of their inner workings. In the definition of Bellare and Rogaway, the random oracles produce a bit-string of infinite length: $RO : \{0,1\}^* \to \{0,1\}^\infty$, of which we than take the first n bits. In our work however, we focus on random oracles with fixed-length output. Interaction with this coracle is done by queries. A party con query a string x to the random oracle and the oracle will see if x has been queried before. If it has not been queried before, the oracle will reply with a random string y of length n, where $n \in \mathbb{N}$. if x has been queried before, then the oracle will reply the same string that was returned when x was queried for the first time.

5.10 Distinguishability

With the help of random oracles we can conduct an experiment to help us prove the security of this protocol. To measure the capability of the adversary, the following experiment is usually performed. Secretly we will select either our cryptographic protocol CP with a secret key or a random oracle RO. We choose so by generating a random bit b which will either be 0 or 1. We select CP if $b = 1$ and select RO if $b = 0$. It is up to the adversary $\mathcal{A}$ to find what the value of the bit is. Now $\mathcal{A}$ will be in one of two worlds: the *real world* or the *ideal world*. In the real world, $\mathcal{A}$ interact with CP and $\mathcal{A}$ interacts with the RO in the ideal world. $\mathcal{A}$ does not know in which world it is in, but can query the entity it is interacting with, noted as *oracle*. When $\mathcal{A}$ queries the input X, CP will process the input paired with the secret key which is unknown to the adversary and provide output Y of length n bits. RO will take the input X and see if it has already received X before. If that is the case, output Y will be the same as when X was queried before. If X was not queried before, output Y will be a random bit string of length n. Eventually A returns a bit $b' \in \{0, 1\}$. $b' = 1$ if $\mathcal{A}$ thinks it is in the real world, and $b' = 0$ if it thinks it is in the ideal world.

In the case of public key cryptography, we can use a random oracle to prove the security of our protocol. If we give this random oracle a proper interface so that it accepts the same values as our protocol and provides similar looking output, we can query the random oracle for a session key. We can then use the experiment defined above to get a session key from either the protocol or the oracle and ask an adversary to tell us in which world he is in. Using this distinguishability experiment we then prove the security.

5.10.1 Advantages

If the adversary would just guess, then he has a success probability of 1/2. So in order to have a successful attack, he needs to guess b with a probability that is significantly larger than 1/2. The advantage of the adversary is:

$$Adv_{CP}(\mathcal{A}) = 2Pr(b = b') - 1$$

Another formula can be derived.

$$Adv_{CP}(\mathcal{A}) = 2Pr(b = b') - 1 = Pr(b' = 1|b = 1) - Pr(b' = 1|b = 0)$$

This formula is what we will used in order to calculate the advantage of an adversary. So if there is an attack where Adv_{CP} is significantly larger than 0, then this would count as a successful attack.

5.10.2 Security Strength

Proving an upper bound for the advantage of an adversary against the protocol is not possible. We can never know if a protocol is secure, we can only disprove the security of a protocol by providing an attack. The best we can do is claiming an upper bound that corresponds to an exhaustive key search. An exhaustive key search is where an adversary tries all possible keys until the right key is found.

5.11 Diffie-Hellman

In order to understand the Extended Triple Diffie-Hellman protocol I am intoducing the Diffie-Hellman protocol. Diffie-Hellman is a key-exchange protocol, created by Whitfield Diffie and Martin Hellman dating back to 1976. The protocol ensures that two parties can calculate a shared secret, from their own personal secrets. A third party with no access to these personal secrets is unable to calculate the shared secret. Traditionally the two parties are named Alice and Bob and the malicious third party is named Eve.

Diffie-Hellman makes use of public key cryptography in order to create a shared secret that only Alice and Bob know. In order for Alice to send a message to Bob, she calculates the shared secret and uses this as an input for a Key Derivation Function (KDF). This is a function that creates a secret key based on the shared secret that can be used for secure communication. KDF is explained in 5.7 Bob can also compute the shared secret and uses the KDF to calculate the same secret key. The protocol makes use of either a multiplicative group G of order q and a generator g or an elliptic curve group of order q and base point g. The protocol goes as follows:

- **Key Generation:** Alice chooses a number a and calculates $A \equiv g^a \bmod q$, Bob chooses a number b and calculates $B \equiv g^b \bmod q$
- **Alice sends A to Bob**
- **Bob sends B to Alice**
- **Now both parties can calculate** $K \equiv A^b \equiv g^{ab} \equiv g^{ba} \equiv B^a \bmod q$

The domain parameters of the Diffie-Hellman protocol are (q, g). These parameters define the group that is used in the protocol. The public keys are (A, B) and the private keys are (a, b). It is important for each party to know the corresponding private key for one of the public keys.

5.11.1 Security

The Diffie-Hellman protocol is based on the computationally hard Diffie-Hellman assumption. This means that an attacker needs to solve this computationally hard problem if it wants to break the protocol. However this is dependent on the choice of the group and the generator. If the group is too small, solving the computational problem is not difficult. In order for Eve to find out what the shared secret is, she has to either compromise one of the private keys, or find a way to calculate the shared secret with only the public keys. For example, in order for Eve to compromise the shared secret key g^{ab}, she has to either calculate a from g^a with a discrete logarithm, or Eve can have another method to calculate g^{ab} from g^a and g^b. This means that breaking the Diffie-Hellman protocol by trying to compute the shared secret key is not efficient and as such Diffie-Hellman is secure against these kinds of attacks.

5.11.2 Public Key Authentication

It is possible, however, for Eve to impersonate Alice, by just sending Bob a public key and saying she is Alice. Therefore, it is important for Bob to authenticate the public key that was sent to him. One way of doing this, is by using an out-of-band authentication. This is a verification method that is on a different communication channel, for example Bob can physically meet Alice and compare the public key he received with the public key of Alice herself. This is difficult to do on a large scale, especially if keys are replaced often or the parties that communicate with each other live very far away.

5.11.3 Signatures

A signature scheme is a way to cryptographically sign a message in such a way that the receiver can authenticate that the message came from the sender and that the message is not changed. The use of signatures provide authentication of the party that is sending the message. It also provides integrity of the message, meaning that it is visible if the message is altered during transmission. It also provides non-repudiation of origin, meaning that the sender cannot deny having sent the message. The problem with using signatures for authentication is that the public key that is used has to be authenticated itself before it can be used. Signature schemes make use of the public and private keys of public key cryptography. If Alice sends a message to Bob, but wants to sign it, she signs the message with Alice her private key and sends both the signature and the original message to Bob. Bob can verify the signature using Alice her public key. Only Alice has access to her private key, so Bob can be almost sure that the message came from her.

5.11.4 Certificates

Another way of authenticating is by introducing a trusted third party (TTP), for example a company, also called a Certificate Authority (CA), and certificates. A certificate is a signature that binds an identity to a public key. For example, Alice and Bob have generated key pairs (a, A) and (b, B). If Alice is sure that Bob his public key is B, she can sign a message containing the public key of Bob and Bob his identity. For example, the message could be "Bob's public key is B". Bob can then use the signature from Alice to show other parties that he is communicating with that B is indeed his public key. However, for this to work Alice needs to be trusted by the other parties that Bob is communicating with. A CA can be such a party that is trusted by every other party. Any party that wants to use a CA to obtain and verify certificates must obtain the public key of the CA. This has to be done in a secure way, else it is not possible to authenticate the public keys of other parties using this CA. This can be done by physical means and is better to scale compared to the previous method, since every party only has to visit the CA once and they do not have to physically meet the other parties. In our case we hard coded the key of the server so that no one can impersonate it.

5.11.5 Diffie-Hellman Problem

In the case of Diffie-Hellman an attacker wants to calculate the shared secret $K = g^{ab}$, where a and b are the private keys of the communicating parties. There are several variants of computational problems that arise for the attacker. First there is the Computational Diffie-Hellman Problem (CDH): given a triple of elements (g, g^a, g^b) where a and b are randomly chosen, find $K = g^{ab}$. This is based on the discrete logarithm problem. If an adversary could easily solve the discrete logarithm problem, then the adversary can also solve CDH easily. First the adversary uses the discrete logarithm to calculate a from g^a. Then the adversary can compute $(g^b)^a = g^{ab}$. However, there is no proof as of yet that this is the only way to solve CDH. In some special cases it can be shown that the discrete logarithm assumption is the same as the CDH assumption.

Second there is the Decisional Diffie-Hellman Problem (DDH): given a quadruple of elements (g, g^a, g^b, g^c) where a and b are randomly chosen, decide whether $c \equiv ab \bmod p$. In other words, if we have (g^a, g^b, x) it is difficult to decide if $x = g^{ab}$ or x is a random element in the group generated by g. It is the case that if CDH is easy to solve, then so is DDH, however the opposite is not true. There are some groups where solving DDH is easy, but CDH is not. These groups are called Gap Diffie-Hellman groups.

There is also a computational problem related to these Gap Diffie-Hellman groups, namely the Gap Diffie-Hellman Problem (GDH): given a triple of elements (g, g^a, g^b) where a and b are randomly chosen, find $K = g^{ab}$ with help of a DDH Oracle. Having access to a DDH

Oracle means that we can query the oracle with a quadruple of elements (g, g^a, g^b, g^c). The oracle will return 1 if $g^c = g^{ab}$ and 0 otherwise. This can be used to verify potential solutions to the GDH problem and query them to the DDH oracle to see if they are correct. The proof of security of X3DH makes use of the GDH hardness assumption.

5.12 Game Hopping

In this proof we will calculate the advantage of the adversary using *game hopping*. Game hopping is a technique used in security proofs of protocols. We will construct a series of games in order to prove the security. Each game differs slightly from the preceding game. These changes have to be small in order to analyze them better. The advantage that the adversary has in each game is bounded in some way by the preceding game. The first game will be the original experiment. The following games will lead to a game for which we can prove a bound. Because each game is bounded in some way by their preceding games, we can then prove a bound for the original experiment

5.13 The X3DH Protocol

Extended Triple Diffie-Hellman or X3DH is a key agreement protocol. This means that through this protocol two parties can agree on a shared secret key through an unprotected channel of communication. In order to make the explanations easier to read, we will give names to the parties that are involved in the protocol. In cryptography it is common to name these parties Alice and Bob. X3DH is designed for asynchronous messaging. It is possible to use Diffie-Hellman for asynchronous messaging, however it is not designed specifically for this. Asynchronous messaging makes it possible for Bob to have published some information on a server and be offline, while Alice can establish a shared secret key and send encrypted messages using the information on the server. The server needs to be a trusted server in order to ensure secrecy and a secure connection. If that is not the case, then the server could provide a key to Alice of which the server knows the private key. In that case, the server would be able to read the messages.

5.13.1 Trusted Server

A trusted server will distribute the keys in the way that we expect and will not provide keys so that the server can read the messages. If the server is not a trusted server, then the messages that are sent between Alice and Bob could be deleted by the server. In order to establish trust between a party and a server, it is possible to use the Trust On First Use (TOFU) principle. In short, a party connects to a server which the party does not trust yet and hopes nothing goes wrong. If nothing goes wrong at first, then the server can be trusted.

As already mentioned we use certificates hard-coded in the application. During this analysis we do not assume that the server is trusted. However we do assume that Alice and Bob have authenticated the public keys of each other, so that an impersonification of either party by the server or other parties would be noticed.

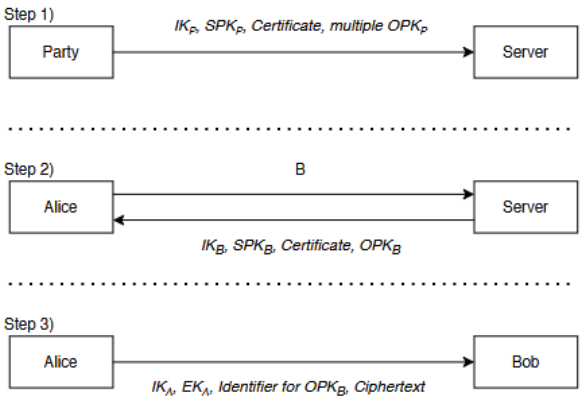


Figure 4: A visual representation of the X3DH protocol

5.14 Prekeys

We assume that Alice is the party that initiates the X3DH protocol and wants to send a message to Bob. In order for Bob to be able to receive a message, he needs to generate and store some keys on the server. The generation of these keys is implementation specific, but as specified in the documentation we used X25519 form. These form have been discussed in 5.6. Bob has a private key for every key that is uploaded to the server. First, Bob needs to generate and store his public identity key IK_B to the server. The private key corresponding to the identity key is ik_B and Bob should keep this key a secret. The identity key is a long-term key that Bob generates when he connects to the server for the first time. This key is tied to his identity so this key will not be replaced over time. The second key that needs to be generated and stored is a medium-term public key SPK_B , called a signed prekey. This is called a signed prekey, because Bob also needs to store a public key certificate of SPK_B using IK_B . This signed prekey and the certificate have to be replaced regularly, by generating a new signed prekey. The server will be updated to use the new signed prekey. After replacing the key, Bob keeps the old private key of SPK_B for a short time, in order to be able to decrypt delayed messages. It is important that he deletes the old keys after that short period of time, in order to provide forward secrecy as these keys are needed to recalculate the shared secret that is used to decrypt past messages. Due to our implementation choices

this is not a problem. The amount of time that the old keys are stored are dependent on the implementation. Since one-time prekeys are used, replaying older signed prekeys are not harmful, since they have to be accompanied with a one-time prekey. If Bob notices a one-time prekeys has been reused, Bob does not accept the message. These one-time prekeys OPK_B are the last keys that have to be uploaded by Bob. These keys are used a single time, after which the server deletes the key. When the server notices that the amount of one-time prekeys is getting low, or whenever Bob wants, he can upload more of these keys to the server. These keys and SPK_B are called *prekeys* because the keys are uploaded prior to the beginning of the protocol. So in short, the values Bob needs to store are:

- IK_B
- SPK_B
- **A certificate of** SPK_B **using** ik_b
- **Multiple** OPK_B

The first two keys and the certificate combined with a single OPK_B is called a *prekey bundle*.

5.15 Initiating the Protocol

In order for Alice to initiate the protocol and send a message to Bob, she first has to request a *prekey bundle*. Alice needs to verify the certificate before she will use this bundle. Since she has access to IK_B, she can check the contents of the certificate. If the contents are equal to SPK_B, then the certificate is correct. If the certificate is incorrect, then Alice will not accept the message. After checking the certificate, Alice creates an ephemeral Diffie-Hellman key pair with public key EK_A. Ephemeral keys are newly generated keys for each new instance of the protocol and are only used for that instance. Identity keys and medium-term keys can be used for multiple instances of the protocol, but ephemeral keys are only used for a single instance. Alice also has her own identity key IK_A of which she has the private key. Alice has access to the following private keys: ik_a and ek_a . Now she can compute the Diffie-Hellman secret keys that are needed to calculate the shared secret key K_{AB}.

- $K_1 = DH(IK_A, SPK_B)$
- $K_2 = DH(EK_A, IK_B)$
- $K_3 = DH(EK_A, SPK_B)$
- $K_4 = DH(EK_A, OPK_B)$

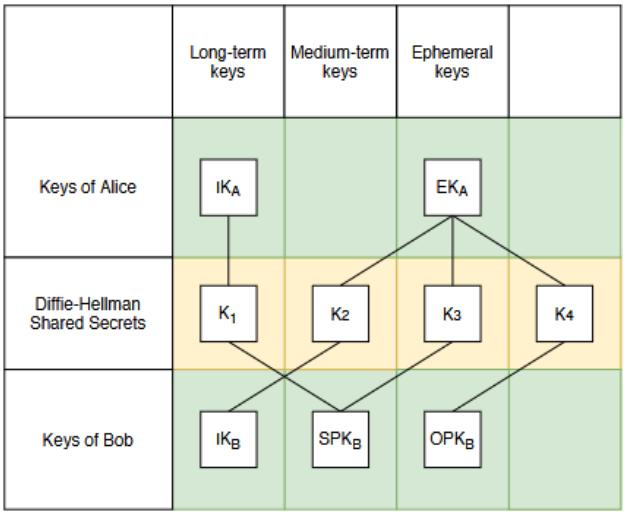


Figure 5: Visual representation of the necessary steps to derive all the DH secret keys

For a visual representation: Using a KDF Alice and Bob can subsequently calculate the shared secret key: $K_{AB} = KDF(K_1||K_2||K_3||K_4, l, salt)$, where $||$ means concatenation of the keys. The l and $salt$ are respectively 256, $\{0\}^{256}$. After Alice has calculated K_{AB} she deletes her ephemeral key, ek_A, and all the K_i. Now that Alice has the shared secret key K_{AB}, she can send a message to Bob so that Bob can calculate K_{AB} as well. Alice will send the following to Bob:

- IK_A
- EK_A
- **An identifier of which one-time prekey is used**
- **An initial ciphertext that is encrypted with** K_{AB}

5.15.1 Deciphering

In order for Bob to decipher the message, he needs to calculate K_{AB}. After receiving the initial message, Bob knows IK_A, EK_A and which one-time prekey Alice used. He also has access to the following private keys: spk_B, ik_B and opk_B . Now Bob can do the same Diffie-Hellman calculations that Alice did, because the public keys that are used are either already known by Bob or sent by Alice. It is important that Bob also deletes the intermediary K_i values and opk_B after calculating K_{AB}. Now Bob can decrypt the initial ciphertext sent by

Alice and checks if it is correct. It is important that if it is incorrect, for example if the message decrypts to a random string, Bob stops all communication and deletes all associated values. If it is correct, both parties can use shared secret key K_{AB} for the rest of their communication.

5.16 Threat Model

We first need to define our threat model. We assume an active adversary Eve that has control over the network on which the messages will be sent. This means that Eve can delete messages, but also alter them. Eve can set up a session between two parties and can choose ephemeral keys. This way, if Eve cannot break the protocol, then attackers with less control over the network cannot either. We do not consider the use of side-channel attacks, for example a timing attack. These attacks are focused not on the protocol itself, but on the devices that run the protocol. We assume that the KDF function is working as it should and that there is no other way to calculate the shared secret key than by using the Diffie-Hellman values. We do assume that Alice and Bob have both verified the public keys of each other. The property we want to prove is secrecy of the shared secret key. Authentication will be proven alongside secrecy by showing that only the intended parties could compute the key K_{AB} . Secrecy of the shared secret key will be proven by showing that the shared key will stay a secret even if some secret values from either Alice or Bob would be compromised.

5.17 Proof

5.17.1 Scope of the Proof

In this paragraph we will fucus son one-to-one messaging even if this algorithm can be extended to group chats our implementation does not support this feature.

5.17.2 Proof of Security

With the help of game hopping we can now provide a proof of security. The goal is to prove that Adv_0 is bounded by the Gap Diffie-Hellman hardness assumption. In the X3DH protocol there are four keys that need to be kept a secret. This means that we have to consider four cases in the proof. These cases are somewhat similar, so we will give a proof for a specific case and for the other cases we will discuss briefly the differences and how they impact the outcome.

First we will construct five games that affect all four cases, after which we will construct four games for each case. In the last game for each case the shared secret key will be replaced by a random value. Then we can calculate a bound on the success probability of the adversary.

We will construct each game towards a reduction to Gap Diffie-Hellman (GDH). This means that we will replace some Diffie-Hellman keys with values that we query from a GDH oracle. The games that are used in the proof are from the proof given by Cohn-Gordon et al. [1]. The games are rewritten to be used with the X3DH protocol. The entity that the adversary sends his messages to is called the challenger, which is denoted as C. This entity *challenges* the adversary to break the system.

Theorem 1 (Security of X3DH)**.** *The Extended Triple Diffie-Hellman (X3DH) protocol is a secure key exchange protocol under the Gap Diffie Hellman assumption and assuming the Key Derivation Function is a random oracle.*

Proof. We start with constructing five games which are used in all four cases. We will define variables:

- q which is the order of the group that are working in
- p is the number of parties
- s is the maximum number of sessions
- e is the maximum number of ephemeral or medium term keys for every party

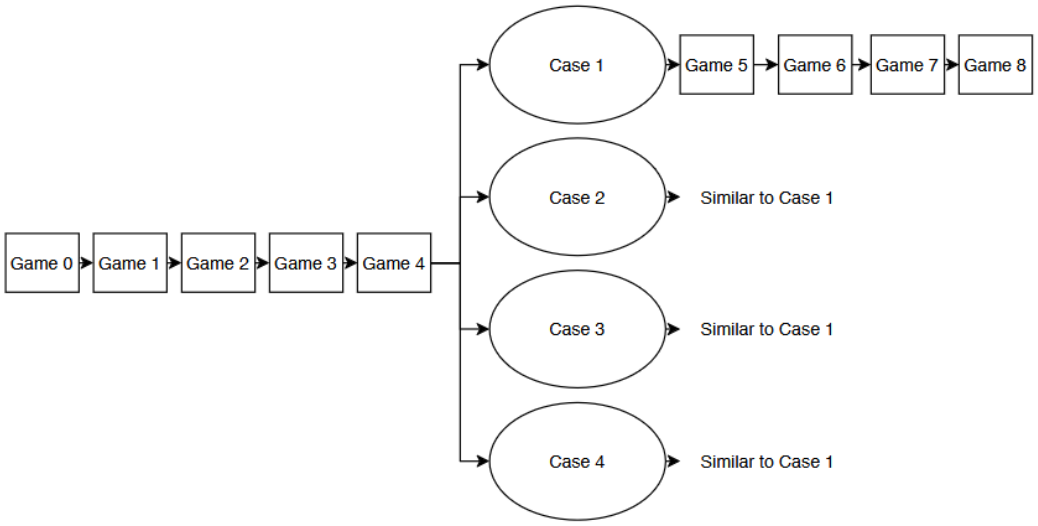


Figure 6: An overview of the proof.

Game 0: The initial experiment

The advantage of the adversary $\mathcal{A}$ against the game is Adv_0

$$Adv_0(\mathcal{A}) = |2Pr(Exp_0(\mathcal{A}) = 1) - 1|$$

Game 1: No Diffie-Hellman collisions

In order to help us in further game hops, we will first make sure that there are no Diffie-Hellman public keys that are the same. Having two keys that have the same value is also called a collision. To be able to make sure that there are no collisions, the challenger $\mathcal{C}$ will keep a list containing all the Diffie Hellman private keys. This way it is easy to check for $\mathcal{C}$ if a value appears twice. If this is the case, the game is aborted and the adversary loses. Each party has generated a long-term identity key and a maximum of e ephemeral keys. This gives us a total of $p + ep$ Diffie-Hellman keys and a total of $\begin{pmatrix} p+ep \\ 2 \end{pmatrix}$ pairs of keys. Every single pair of these keys must not collide. Two keys in the same group with order q will collide with a probability of $1/q$. Now we can give the following bound on the advantage of the adversary:

$$Adv_0 \leq \frac{\begin{pmatrix} p+ep \\ 2 \end{pmatrix}}{q} + Adv_1$$

Game 2: Guess test session

In this game, the challenger must guess which session will be tested by the adversary with Test(x, i). This means that $\mathcal{C}$ must guess both x and i correctly. For x there are a total of p possibilities, and for i there are s possibilities. That means that the probability that the challenger guesses both of them correctly is $1/xp$. This will be a game hop with a large failure event. Let $\mathcal{C}$ guess $(x^*, i^*) \in [1...p] \times [1...s]$. We will define an event E where $\mathcal{A}$ will query Test(x, i), where $(x, i) \neq (x^*, i^*)$. We will abort the game when E occurs. The probability that $\neg E$ occurs is $1/ps$. From this we get the following:

$$Adv_1 \leq psAdv_2$$

Game 3: Guess unique partner session

We must also guess the partner session that is tied to the test session. Since in Game 1 we made sure that all the Diffie-Hellman keys are unique and as such the combination of identity keys, we know that the partner session is also unique. This closely follows Game 2,

as in Game 2 we guessed a certain session as well. This is a game hop with a large failure event. So $\mathcal{C}$ guesses a pair $\mathcal{C}$ guess $(x^*, i^*) \in [1...p] \times [1...s]$. We will define an event E where the partner session is π_x^i where $(x, i) \neq (x^*, i^*)$. Since the partner session is unique, we know that the probability of $\neg E$ occurring is $1/ps$.

The session identifiers contain the public keys. However, since the adversary can pick ephemeral keys, it is possible that there is another session that has the same identifier as the test session that is not this partner session. We will make sure that if such a session exists, then the challenger knows about it. This is another hop with a large failure event. We do this by guessing the index $j \in \{1...s\}$ of this session. We will define an event F where this session is π_x^i and $i \neq j$. The probability that $\neg F$ occurs is $1/s$. Combining both failure events gives us the following bound:

$$Adv_2 \leq ps^2 Adv_3$$

Game Hops for Specific Cases

Now we have to move to a game distinction. We can now write the advantage of $\mathcal{A}$ in Game 3 as follows:

$$\mathrm{Adv}_3 \leq \underbrace{\mathrm{Adv}_3^{\mathrm{clean}_{LM}(x,i)}}_{\text{Case 1}} + \underbrace{\mathrm{Adv}_3^{\mathrm{clean}_{EL}(x,i)}}_{\text{Case 2}} + \underbrace{\mathrm{Adv}_3^{\mathrm{clean}_{EM}(x,i)}}_{\text{Case 3}} + \underbrace{\mathrm{Adv}_3^{\mathrm{clean}_{EE}(x,i)}}_{\text{Case 4}}$$

Where $Adv_3^{clean_{LM}(x,i)}$ is the advantage of the adversary in the case that $clean_{XY}(x, i)$ must stay true.

We will define all the game hops for the case $Adv_3^{clean_{XY}(x,i)}$ and $\pi_x^i.role = initiator$. The case of $\pi_x^i.role = responder$ is analogous, and only one of the two appears.

Case 1: $clean_{LM}(x, i)$

In this case we must make sure that the $clean_{LM}(x, i)$ stays true. This means that $\mathcal{A}$ cannot have issued the queries RevealIdentityKey(x) and RevealPreKey(n, y) where $\pi_x^i.otherprekeyid = n$

Game 4: Guessing the prekey of the other party

In this game the challenger has to guess the index of the signed prekey that is used by the other party that is used in the Test session.$\mathcal{C}$ will guess $n \in \{1...e\}$. This is a game hop with a large failure event. We will define an event E where n is not the index of the signed prekey that is used. The probability of $\neg E$ is $1/e$, which gives us the following:

$$Adv_3 \leq eAdv_4$$

Game 5: Allowing some duplicate values

To prepare for future game hops, we will allow the identity key of party x, IK_x, and the signed prekey of party y with index n which we guessed in Game 4, SPK_n^y, to be identical. We do this because we want to introduce a GDH challenger. With this challenger these keys are allowed to be the same. We are working in a group of order q, so the probability that the keys are the same is $1/q$. This is a game hop with a small failure event and gives us the following:

$$Adv_4 \leq \frac{1}{q} + Adv_5$$

5.17.3 Game 6: Introducing GDH

In order to calculate the shared secret key of the X3DH protocol, we need four parts to be used as input to the KDF. We are in the case $clean_{LM}(x, i)$, which means that ik_x and spk_n^y remained a secret to the adversary. The shared secret key is calculated as follows: $K_{AB} = KDF(K_1||K_2||K_3||K_4)$. Without loss of generality $\mathcal{A}$ knows K_2, K_3 and K_4 and it wins if it queries $KDF(K_1||K_2||K_3||K_4)$. So in order for $\mathcal{A}$ to win, $\mathcal{A}$ must find $K_1 = g^{ik_x spk_n^y}$ for the generator g of the group. We will abort this game when $\mathcal{A}$ queries K_1 to the KDF oracle it has access to, because that is the case that the protocol is broken. We will define this event as E . We will show that if E happens, the adversary will win the game, which we have defined as $Adv(break_6)$. So we can split the advantage as follows:

$$Adv_5 \leq Adv(break_6) + Adv_6$$

We will now show that in the case of E, we can create an algorithm $\mathcal{A}^*$ that can win the GDH problem. In the GDH problem $\mathcal{A}^*$ is given a triple (g, g^a, g^b), and its goal is to find $K = g^{ab}$ with access to a DDH oracle.

Now $\mathcal{A}^*$ will replace IK_x with g^a and SPK_n^y with g^b, with both a and b unknown to $\mathcal{A}^*$. $\mathcal{A}^*$ will then simulate Game 5. Since IK_x and SPK_n^y are replaced, it is not possible to calculate some session keys between certain parties. Since the adversary $\mathcal{A}$ can set up sessions between parties, $\mathcal{A}^*$ must simulate these calculations by giving random keys. These sessions are:

1. A session which is not the Test session that is between parties x and y, where x is the initiator. Since g^a and g^b are both used to calculate the session key, this would require knowledge about a or b.

2. A session which is not the Test session that is between party x and any other party, including y, where x is the responder. Since g^a is used and some other ephemeral public key on which we have no information, this would require knowledge about a.

3. A session which can be the Test session that is between any party that is not x and party y, where y is the responder. Since g^b is used and some other ephemeral public key on which we have no information, this would require knowledge about b.

$\mathcal{A}^*$ will keep a list which contains, for each session for which a random key is used, the random key and which public keys should have been used for the calculations. By keeping this list, $\mathcal{A}^*$ must be consistent in answering queries from $\mathcal{A}$ by giving the right random keys for the right sessions. Before simulating a session and picking a random key, it is also important that $\mathcal{A}^*$ will use the DDH oracle to check if a previous random oracle query matches the session, in order to give the right key. Otherwise, the simulation will not be consistent. It is also important that $\mathcal{A}^*$ will not answer the reveal queries which will violate the clean predicate, as this predicate must stay true or the adversary loses. If $\mathcal{A}$ queries the KDF oracle with a query of the form $g^{z_1}||g^{z_2}||g^{z_3}||g^{z4}$, then $\mathcal{A}^*$ must first go through its list and use the DDH oracle to check if the public keys match the corresponding g^{z_i}, for $i \in \{1...4\}$. These g^{z_i} each match the K_i that is used to calculate the shared secret key. For each session type that uses random keys we can then construct DDH oracle queries to give us the information we need.

For sessions of type 1, $\mathcal{A}^*$ has no knowledge about a, b and some other ephemeral private key e, so in that case $\mathcal{A}^*$ will query the DDH oracle (g, g^b, g^a, g^{z_1}) to see if the first key is correct, and (g, g^e, g^a, g^{z_1}) to see if the third key is correct. $\mathcal{A}^*$ knows enough to check the other keys by itself.

For sessions of type 2, $\mathcal{A}^*$ has no knowledge about a and e, which is some other ephemeral private key. In that case we will query the DDH oracle (g, g^a, g^e, g^{z_2}), in order to see if the second key is correct. $\mathcal{A}^*$ knows enough to check the other keys itself.

For sessions of type 3, $\mathcal{A}^*$ has no knowledge about b and e, which is some other ephemeral private key. In that case we will query the DDH oracle (g, g^b, g^e, g^{z_3}), in order to see if the third key is correct. $\mathcal{A}^*$ knows enough to check the other keys itself.

For each case, if the DDH oracle returns 1, meaning that the key is correct, then $\mathcal{A}^*$ can use the random keys that it generated from the list it keeps. If the oracle does not return 1, then $\mathcal{A}^*$ must generate a new random value and query what that value to the oracle.

It is important to note that if $\mathcal{A}^*$ queries (g, g^a, g^b, g^{z_1}) to the DDH oracle, and it returns 1, then $\mathcal{A}$ has found the solution to the GDH problem, and $\mathcal{A}^*$ will just return g^{z_1} as the answer to the GDH challenger. In other words, if $\mathcal{A}^*$ breaks Game 6, then $\mathcal{A}^*$ breaks the GDH problem.

So from this we have that if E occurs, then we have a solution to the GDH problem. This gives us the following:

$$\mathrm{Adv}_{\mathrm{break}_6} \leq \varepsilon_{\mathrm{GDH}}(\mathcal{A}^*). \tag{1}$$

Where $\varepsilon_{\mathrm{GDH}}(\mathcal{A}^*)$ is the probability of guessing the right answer for the GDH problem.

Game 7: Replacing the session key

The last game is where the adversary tries to find the session key. However, before that happens, the experiment will generate a randomly chosen string that is from the same space that all the keys in this protocol come from. This means that it is not detectable by the adversary whether the key has been replaced. We know from Game 6 that $\mathcal{A}$ did not query the KDF oracle with the correct input. The session key is now replaced with a random other key, depending on the bit b that was randomly chosen at the start of the experiment. The adversary will have no advantage anymore, which gives us the following:

$$\mathrm{Adv}_6 = \mathrm{Adv}_7 = 0. \tag{2}$$

Of all these games combined in case 1, we can derive the following:

$$\mathrm{Adv}_3^{\mathrm{clean}_{\mathrm{LM}}(\S,i)} \leq \rceil \left(\frac{1}{q} + \varepsilon_{\mathrm{GDH}}(\mathcal{A}^*) \right). \tag{3}$$

Case 2: $\mathbf{clean}_{EL}(x, i)$

In this case we make sure that the $\mathrm{clean}_{EL}(x, i)$ predicate stays true. This means that $\mathcal{A}$ cannot have issued the queries RevealEphemeral(x, i) and RevealIdentityKey(y) in the case that π_x^i.role = initiator and π_x^i.otherik $= y$ is the other party of the session. In the case that π_x^i.role = responder the queries RevealEphemeral(y, i) and RevealIdentityKey(x) cannot have been issued.

The differences from the previous games is that we no longer have to guess the index in Game 4. In Game 5 we allow EK_x and IK_y to be duplicate, and in Game 6 we replace these keys with the GDH values g^a and g^b. From this we get the following:

$$\mathrm{Adv}_3^{\mathrm{clean}_{EL}(x,i)} \leq \frac{1}{q} + \varepsilon_{\mathrm{GDH}}(\mathcal{A}^*). \tag{4}$$

Case 3: $\mathbf{clean}_{EM}(x, i)$

In this case we make sure that the $\mathrm{clean}_{EM}(x, i)$ predicate stays true. This means that $\mathcal{A}$ cannot have issued the queries RevealEphemeral(x, i) and RevealPreKey(n, y), where π_x^i.otherprekeyid $= n$ and π_x^i.role = initiator. For π_x^i.role = responder the queries RevealEphemeral(y, i) and RevealPreKey$(\pi_x^i$.prekeyid$, x)$ cannot have been issued.

In this case we do have to guess the index again in Game 4. We will allow EK_x and SPK_y to be duplicate in Game 5. In Game 6 we will replace these keys with the GDH values. From this we get the following:

$$\mathrm{Adv}_3^{\mathrm{clean}_{EM}(x,i)} \leq \rceil \left(\frac{1}{q} + \varepsilon_{\mathrm{GDH}}(\mathcal{A}^*) \right). \tag{5}$$

Case 4: $\mathbf{clean}_{EE}(x, i)$

In this case we make sure that the $\mathrm{clean}_{EE}(x, i)$ predicate stays true. This means that $\mathcal{A}$ cannot have issued the queries RevealEphemeral(x, i) and RevealEphemeral(y, i) in both the cases that π_x^i.role = initiator and π_x^i.role = responder.

In this case we guess the index again in Game 4. However, this time we do not incur a factor of e but a factor of s. This is because ephemeral keys are generated on a session basis and not beforehand. In Game 5 we allow the ephemeral keys of both parties to be the same, and in Game 6 we replace these keys with the GDH values. From this we get the following:

$$\mathrm{Adv}_3^{\mathrm{clean}_{EM}(x,i)} \leq s \left(\frac{1}{q} + \varepsilon_{\mathrm{GDH}}(\mathcal{A}^*) \right). \tag{6}$$

Combining the four cases, we have:

$$\mathrm{Adv}_3 \leq e \left(\frac{1}{q} + \epsilon_{\mathrm{GDH}}(\mathcal{A}^*) \right) + \frac{1}{q} + \epsilon_{\mathrm{GDH}}(\mathcal{A}^*) + e \left(\frac{1}{q} + \epsilon_{\mathrm{GDH}}(\mathcal{A}^*) \right) + s \left(\frac{1}{q} + \epsilon_{\mathrm{GDH}}(\mathcal{A}^*) \right) \tag{7}$$

From this, we can derive a bound for Game 0, the original experiment:

$$Adv_0 \leq \frac{\binom{p + e \cdot p}{2}}{q} + p^2 \cdot s^3 \cdot (e\left(\frac{1}{q} + \epsilon_{GDH}(\mathcal{A}^*) \right) + \frac{1}{q} + \epsilon_{GDH}(\mathcal{A}^*) + e\left(\frac{1}{q} + \epsilon_{GDH}(\mathcal{A}^*) \right) +$$
$$s\left(\frac{1}{q} + \epsilon_{GDH}(\mathcal{A}^*) \right)\Big) \leq \frac{\binom{p + e \cdot p}{2}}{q} + p^2 \cdot s^3 \cdot (\frac{2e + s + 1}{q} + (2e + s + 1)\epsilon_{GDH}(\mathcal{A}^*) \Big) \tag{8}$$

□

6 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. 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.
Another critical issue is the lack of encrypted data in the initial message, preventing the receiver from verifying whether they have derived the correct shared key. This opens the door to potential attacks, such as miss-binding attacks, where an attacker might send invalid messages.
Additionally, 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.
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 or the recipient.
Furthermore, while the system relies on secure, well-established libraries, it does not address potential side-channel attacks, which remain an unaccounted-for vulnerability.

6.1 Further Improvements

Further improvements, other that fixing the problems already mentioned, could involve the implementation of **Double Ratchet** algorithm giving this implementation not only forward secrecy for each session but also for each message. Further more the Double Ratchet algorithm could give as self-healing properties, which means even if key is compromised, futures communication will not.
One final possible addition is the implementation of **Post-Quantum Extended Diffie-Hellman** (PQXDH) to give us post-quantum security. Another more immediate quick fix is to make keys longer (512-bits instead of 256-bits).

7 Instructions for Installation and Execution

7.1 Installation

Clone the GitHub repository available here.
For **Unix-like systems** (e.g. *Linux*), use the provided scripts `server.sh` and `tui.sh`. Before running them, make sure they are executable by running: `chmod u+x server.sh tui.sh`
For **Windows systems**, you can use the corresponding bat scripts `server.bat` and `tui.bat` directly.
Alternatively, you can run the executables using `cargo` by following these steps:

1. Navigate to the `server` directory: `cd server`
2. Run the server with `cargo run`
3. Open a new terminal instance
4. Navigate to the `tui` directory: `cd tui`
5. Run the TUI with `cargo run`

7.2 Configuration

You can modify the `config.toml` file located in the `config` directory to specify different application configurations:

- `server_ip`: The IP address of the server (default: `127.0.0.1`).
- `server_port`: The port of the server (default: `3333`).
- `log_level`: The logging level (default: `info`).

Do not modify `private_key_server` and `public_key_server`, as these will be automatically generated when the server is started.

7.3 Getting Started

7.3.1 Server

Start the server by executing the ‘server‘ script from the root directory. You should see something like Figure 7:

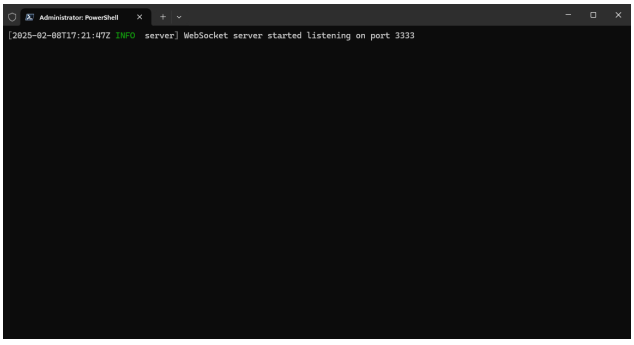


Figure 7: Server window

7.3.2 Client

After the server is running, you can start the client by executing the ‘tui‘ script from the root directory.
The client operates in two modes:

- **NORMAL**: Entered by pressing the `ESC` key. This mode allows interaction with the TUI. For a list of available key combinations, check the bottom of the TUI for instructions relevant to the current window.
- **INPUT**: Entered by pressing the `i` key. As the name suggests, this mode allows you to input text into the application's input fields.

Registration
Before using the client, you need to register by choosing a unique username. Usernames must be non-empty and unique—multiple users cannot share the same username simultaneously. Figure 8 shows a registration example:

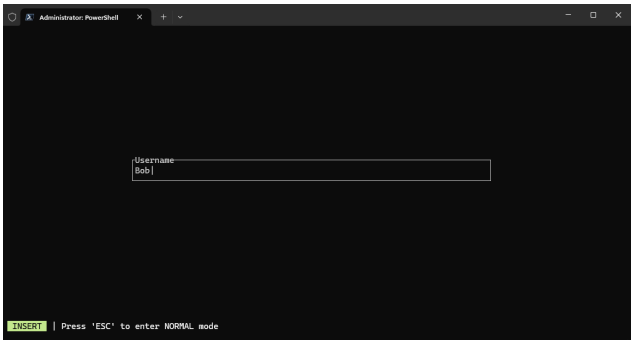


Figure 8: Registration window

Main Window
After registration, you'll be taken to the main window. Initially, your chats list will be empty, as showed in Figure 9.

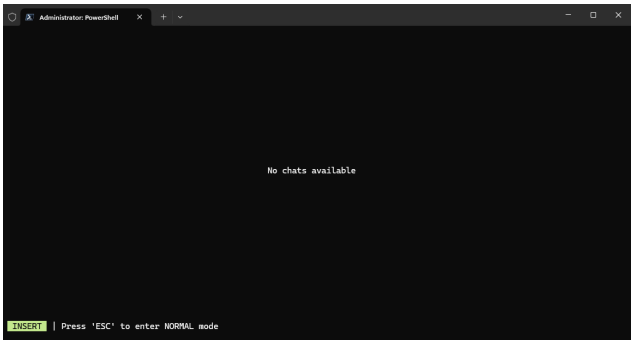


Figure 9: Main window

Adding Friends
To start chatting, you can add new friends at any time by pressing the `a` key in `NORMAL` mode. Of course, the person you want to chat with must be online.
Figure 10 shows an example:

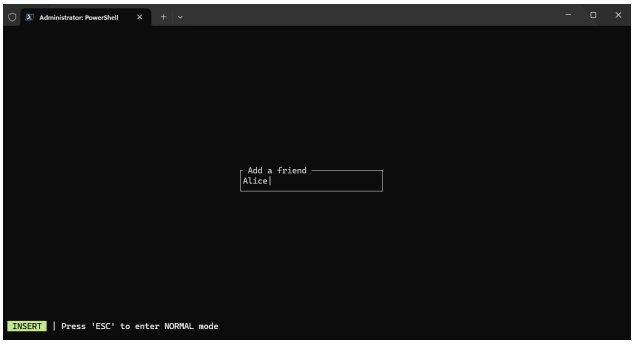


Figure 10: Add a friend window

Navigating Chats
While in `NORMAL` mode:

- Use the **left/right arrow keys** to switch between the chats list and the selected chat.
- In the chat list:
  - Use the **up/down arrow keys** to navigate through your chats.
  - Press `ENTER` to select a chat and start messaging.

Figure 11 presents an overview of the final client interface:

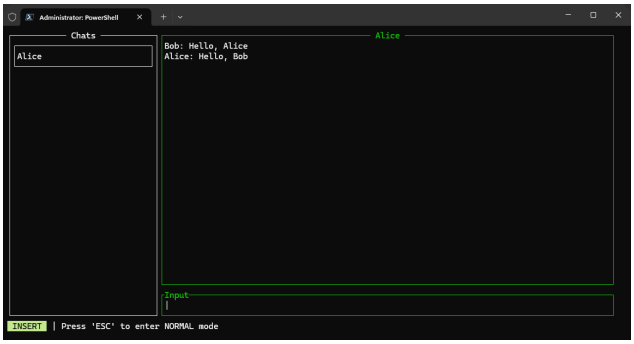


Figure 11: Chats overview window

References

[1] Katriel Cohn-Gordon et al. *A Formal Security Analysis of the Signal Messaging Protocol*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7961996. 2017.

[2] Stefan Friedl. *An Elementary Proof of The Group Law Fof Ellipptic Curves*. https://arxiv.org/pdf/1710.00214. 2017.

[3] Ferran van der Have, Dr. B.J.M. Mennink, and Prof. Dr. J.J.C. Daemen. *The X3DH Protocol: A Proof of Securit*. https://www.cs.ru.nl/bachelors-theses/2021/Ferran_van_der_Have___4104145___The_X3DH_Protocol_-_A_Proof_of_Security.pdf. 2022.

[4] A. Langley, M. Hamburg, and sn3rd S. Turner. *RFC 7748*. https://datatracker.ietf.org/doc/rfc7748/. USA, 2016.

[5] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. https://signal.org/docs/specifications/x3dh/x3dh.pdf. 2016.

[6] J. Salowey, A. Choudhury, and D. McGrew. *RFC 5288: AES Galois Counter Mode (GCM) Cipher Suites for TLS*. https://dl.acm.org/doi/pdf/10.17487/RFC5288. USA, 2008.