Pythonic Implementation of Secure End-End Encrypted Messaging

Sankalp Bhamare 200110096, **Rohan Kalbag** 20d170033

May 5, 2024

1 Introduction

We have implemented the **Signal Protocol** which is used for end-end encryption in messaging systems. We have then developed a secure messaging app based on our native implementation of the signal protocol which internally uses the python `cryptography` library and techniques used in hands-on course labs to implement the Extended Triple Diffie-Hellman (X3DH) Protocol, Double Ratchet Algorithm using basic the basic blocks offered by this library such as Elliptic Curves Generators, AES, KDFs, HMAC and so on.

2 System Setup

The chat applications make use of `socketio` for establishing the communication between the client and server. The application can support communication between **multiple users at the same time**. The code to the our implementation can be found at `https://github.com/rohankalbag/cryptography-signal-protocol`

2.1 Chat Application

2.1.1 Client-Side

The client side chat application consists of user friendly GUI utilizing `PySide6` python library. That allows a user to authenticate and login to the app, for the time being purely on their username, additional functionality such as a password based authentication can be added later. After logging in, a user can see all other users who have currently logged in and can select to choose the person they wish to communicate securely with as shown in Section 4.3.3.

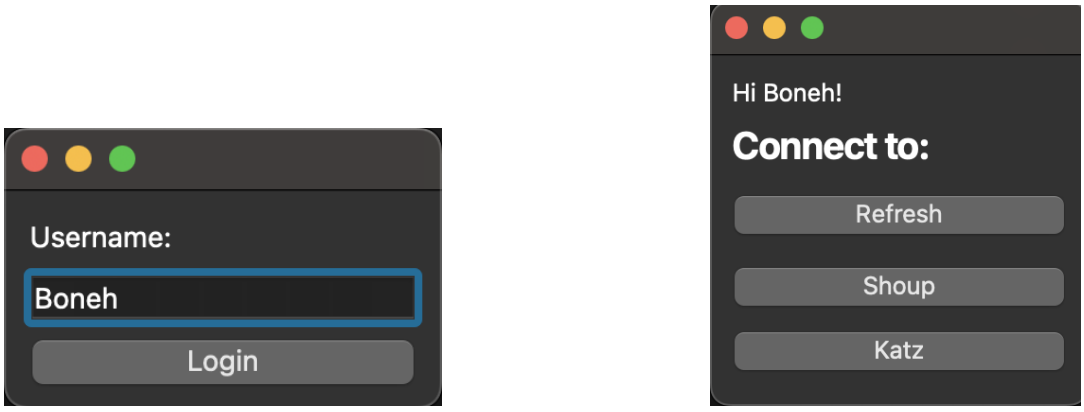


Figure 1: The interface for sender to login and also to select the recipient

This allows the sender to then start chatting with the recipient. The chat history is **stored locally** on the client's device. The chat interface is as shown in Figure 2

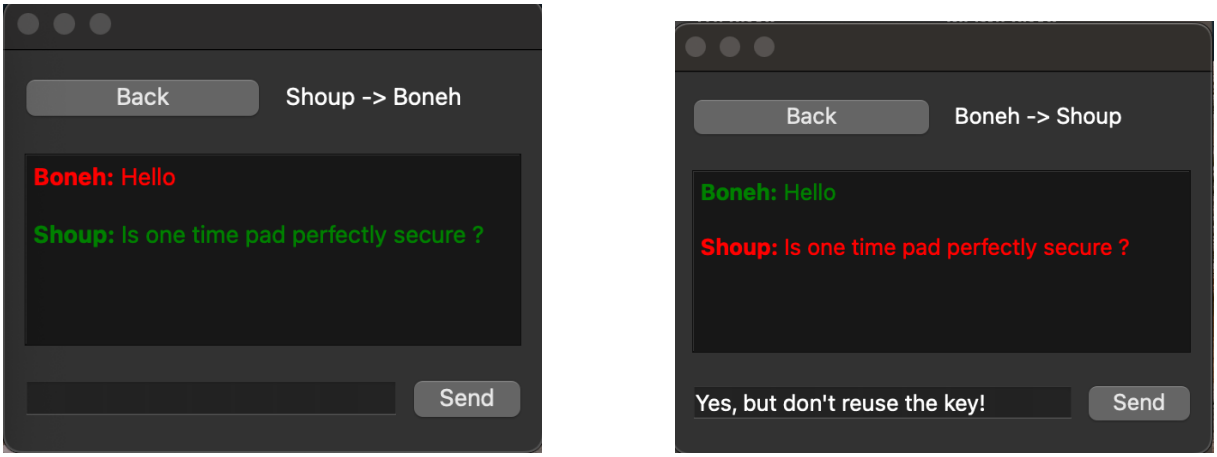


Figure 2: The chat interface which allows the sender to enter message to send to the recipient, it also shows the locally stored chat history

2.1.2 Server-Side

The server uses `tinyDB` for storing the information about the clients who are currently connected to it, and serves a pipe for intercommunication between the clients. The schema of the entries stored in the database are of the form

2.1.3 Diffie Hellman initiation race condition

Because of the blocking nature of `sio.call`[1], when a race condition occurs during initiation from both parties, both of them will stall until timeout and then re-initiate. This eliminates the possibility of two independent sessions being established between them. A potential improvement here would be to implement priority assignment during session creation. For instance, assigning priority r_1 to initiation from A and priority r_2 for B. Tie-breaking would be based on the highest value, ensuring that a single session is established in the end.

2.1.4 Usage Instruction

```
{
    <user_id>:
        {
            "username": <user_name>,
            "ik": <user's public identity key>,
            "sik": <user's public signing identity key>,
            "spk": <user's public signed prekey signed using sik>,
            "spk_sign": <signature of spk>
        }
}
```

[1]RPC procedure calling used for communication

The identity of a user is verified by another user by checking `ik` and `sik` via a secure external authenticated channel, i.e scanning the other user's QR Code to verify the identity (this is done in actual Signal App)

3 Instructions to Setup

3.1 To install dependencies

```
pip install -r requirements.txt
```

3.2 To setup the server

```
python3 server/server.py
```

3.3 For each client to connect to the server and open the chat interface

```
python3 main.py
```

4 Implementing Signal Protocol

4.1 Extended Triple Diffie-Hellman

4.1.1 Introduction

The following section provides an overview of the X3DH (Extended Triple Diffie-Hellman) key agreement protocol. X3DH facilitates the establishment of a shared secret key between two parties who authenticate each other using their respective public keys. Offering forward secrecy and cryptographic deniability, X3DH enhances security in communication scenarios. Specifically designed for asynchronous environments, X3DH addresses situations where one participant, denoted as **Bob** is offline but has shared relevant information via a server. Conversely, another participant, referred to as **Alice** seeks to utilize this information for encrypted communication with Bob, while also establishing a shared secret key for subsequent interactions.

Table 1: Parameters for the X3DH protocol

Name	**Definition**
Elliptic Curve	25519
Hash Function	SHA-256
Encoding	Base-64

The X3DH protocol involves three parties:

- **Alice** intends to transmit encrypted data to Bob and simultaneously establish a shared secret key.
- **Bob** aims to establish a shared key with other parties for encrypted data exchange. However, Bob may be offline at times. To address this, Bob maintains a connection with a designated server.
- The **server** enables Bob to share key bundles/ messages that it will deliver to parties such as Alice for X3DH-exchange / communication.

- $Encode(x)$ denotes obtaining the base64 encoding of x

Name	Definition
IK_A	Alice's identity key pair
IK_B	Bob's identity key pair
SPK_B	Bob's signed prekey pair
EK_A	Alice's ephemeral key pair
SIK_B	Bob's signing identity key pair

- $priv(x)$ denotes the private key of the key pair x and $pub(x)$ represents its public key.
- Each party has a long-term identification public key ($pub(IK_A)$ and $pub(IK_B)$)
- Bob also has a signed prekey pair SPK_B, which will change periodically, and $pub(SPK_B)$ is sent to the server prior to the protocol run
- During each run, Alice generates a new ephemeral key pair $priv(EK_A), pub(EK_A)$
- After a successful protocol run Alice and Bob will establish a 32-byte shared secret SK

4.2 The X3DH protocol

4.2.1 Publishing Keys

Bob sends the following set of keys to the server:

- Identity key $pub(IK_B)$ (Upload identity key to the server once)
- Signing identity key $pub(SIK_B)$ (Upload identity key to the server once)
- Signed prekey $pub(SPK_B)$ (Upload identity key to the server once)
- Prekey signature Sign(SIK_B, $Encode(pub(SPK_B))$) (Will upload a new signed prekey and prekey signature at some interval)

4.2.2 Sending the initial message

Alice contacts the server and fetches a "prekey bundle" containing the following values:

- Bob's identity key $pub(IK_B)$
- Bob's signed prekey $pub(SPK_B)$
- Bob's prekey signature Sign(SIK_B, $pub(SPK_B)$))

Once the bundle is received:

- Alice verifies the prekey signature and aborts the protocol if verification fails. Alice then generates an ephemeral key pair EK_A.
- We will use Diffie Hellman Exchange (DH) to setup the shared exchange, it is function that works as follows, given a private key and public key ($priv(A) = x$ and $pub(B) = Y = g^y$), where $DH(A, B) = Y^x$ denotes the following computation, similar to the diffie hellman key exchange.
- $DH_1 = DH(priv(IK_A), pub(SPK_B))$

- $DH_2 = DH(priv(EK_A), pub(IK_B))$
- $DH_3 = DH(priv(EK_A), pub(SPK_B))$
- $SK = KDF(DH_1||DH_2||DH_3)$
- Here, Key Derivation Function (KDF) is used to derive secret SK from inputs $DH_1, DH_2 and DH_3$ (similar to a PRF)
- After calculating SK, Alice deletes her ephemeral private key and the DH outputs.
- Alice then calculates an "associated data" byte sequence
  $AD = Encode(pub(IK_A))||Encode(pub(IK_B))$
- Alice then sends Bob an initial message:
- Alice Identity key $pub(IK_A)$
- Alice ephemeral key $pub(EK_A)$
- An initial ciphertext encrypted with an AEAD encryption scheme using associated data and an encryption key which is derived from SK

4.2.3 Receiving the 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 signed prekey and one-time prekey used by Alice, using the AD.
- Repeat the DF calculations and compute the secret.
- Bob then constructs the AD byte sequence using $pub(IK_A)$ and $pub(IK_B)$, as described in the previous section. Finally, Bob attempts to decrypt the initial ciphertext using SK and AD

4.3 Double Ratchet

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**. A skipped message here refers to out of order arrived message

4.3.1 Initialize

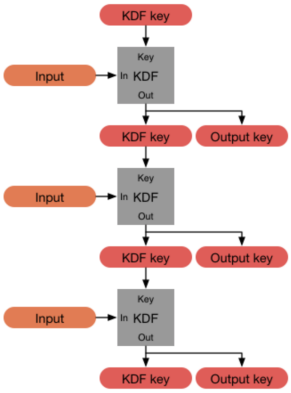
- After Alice and Bob decide on a shared key SK using X3DH, also Alice has access to Bob's $pub(SPK_B)$
- These values will be used to populate Alice's sending chain key and Bob's root key.

4.3.2 Ratchet Encryption

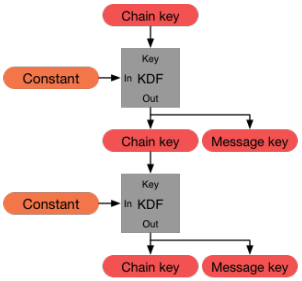
- A **Symmetric-Key Ratchet** step is performed by Alice which then encrypts the message with the resulting message key obtained. In addition to the message's plaintext it takes an AD byte sequence which is prepended to the header to form the associated data for the underlying AEAD encryption

4.3.3 Ratchet Decryption

- If the message corresponds to a skipped message key Bob decrypts the message and deletes the message key
- If a new ratchet key has been received this function stores any skipped message keys from the receiving chain and Bob performs a **Diffie-Hellman Ratchet** step to replace the sending and receiving chains.
- Bob stores any skipped message keys from the current receiving chain, performs a symmetric-key ratchet step to derive the relevant message key and next chain key, and then decrypts the message.



(a) KDF Chain



(b) Symmetric-Key Ratchet

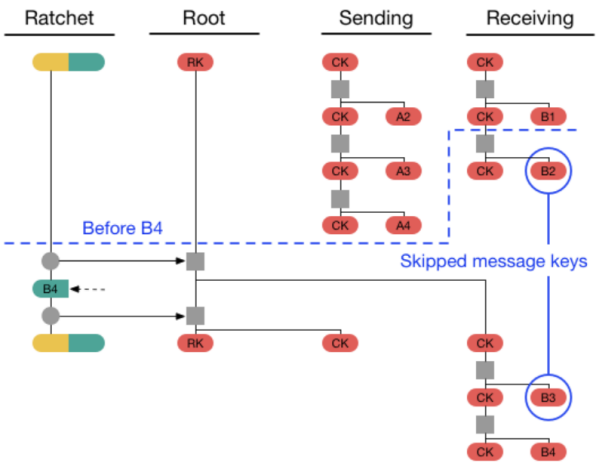


Figure 4: All the three chains of double ratchet in action