# Implementing Signal Protocol

Cryptography CS 411 & CS 507 Term Project for Fall 2023

Erkay Savaş & Atıl Utku Ay
Computer Science & Engineering
Sabancı University
İstanbul

**Abstract**

You are required to develop a simplified version of the Signal Protocol, which provides forward secrecy and deniability. Working in the project will provide you with insight for a practical cryptographic protocol, a variant of which is used in different applications such as WhatsApp.

## 1 Introduction

The project has three phases:

- **Phase I** Developing software for the Public Key Registration
- **Phase II** Developing software for receiving messages from other clients
- **Phase III** Developing software to communicate with other clients

1. All coding development will be in the Python programming language. More information about the project is given in the subsequent sections.
2. You should connect to the university's VPN to be able to connect to the server **if you are connecting from outside of the campus**. Otherwise, you won't be able to send your requests.
   Check the IT's website (`https://mysu.sabanciuniv.edu/it/en/openvpn-access`)

## 2 Phase I: Developing software for the Public Key Registration

In this phase of the project, you are required to upload one file: "Client.py". You will be provided with "Client_basics.py", which includes all required communication codes.

In this protocol, Elliptic Curve Cryptography (ECC) is used in key exchange protocol and digital signature algorithm with NIST-256 curve. You should select "`secp256k1`" for the elliptic curve in your python code. There are three types of public keys; Identity Key (`IK`), Signed Pre-key (`SPK`) and One-time Pre-key (`OTK`).

## 2.1 Identity Key (IK)

Identity Key (IK) consists of a long-term public-private key pair, which each party generates once and uses to sign his/her SPK as shown in Section 2.2 .

### 2.1.1 Registration of Identity Keys

The identity public key of the server $\texttt{IK}_S\texttt{.Pub}$ is given below.

```
X:0x1d42d0b0e55ccba0dd86df9f32f44c4efd7cbcdbbb7f36fd38b2ca680ab126e9
Y:0xce091928fa3738dc18f529bf269ade830eeb78672244fd2bdfbadcb26c4894ff
```

Firstly, you are required to generate an identity private and public key pair $\texttt{IK}_A\texttt{.Pri}$ and $\texttt{IK}_A\texttt{.Pub}$ for yourself. The key generation is described in the "Key Generation" algorithm in Section 2.4. Then, you are required to register your public key with the server. The identity key registration operation consists of four steps:

1. After you generate your identity key pair, you should sign your ID (e.g., 26045). The details of the signature scheme is given in the "Signature Generation" algorithm in Section 2.4. Then, you will send a message, which contains your student ID, the signature tuple and $\texttt{IK}_A\texttt{.Pub}$, to the server. The message format is

   {'ID': stuID, 'H': h, 'S': s, 'IKPUB.X': ikpub.x, 'IKPUB.Y': ikpub.y}

   where stuID is your student ID, h and s are the signature tuple and ikpub.x and ikpub.y are the $x$ and $y$ and coordinates of $\texttt{IK}_A\texttt{.Pub}$, respectively.

2. If your message is verified by the server successfully, you will receive an e-mail, which includes your ID and a 6 digit verification code: code.

3. If your public key is correct in the verification e-mail, you will send another message to the server to authenticate yourself. The message format is "{'ID': stuID, 'CODE': code}", where code is the verification code which, you have received in the previous step. A sample message is given below.

   {'ID': 26045, 'CODE': 209682}

4. If you send the correct verification code, you will receive an acknowledgement message via e-mail, which states that you are registered with the server successfully and contains a code to reset your identity key if you need. (You must save the reset code to delete your IK from server, in case you need (e.g., your identity key is lost or compromised).)
   **Note:** Once you receive an acknowledgment email, your Identity key will be saved locally in your computer.

### 2.1.2 Resetting your IK

If you lose your private identity key $\texttt{IK}_A\texttt{.Pri}$, and need to reset your identity key pair, you should send a message to the server to delete your public identity key $\texttt{IK}_A\texttt{.Pub}$ from the server. The message format is "{'ID': stuID, 'RCODE': rcode}", where `rcode` is reset code which was provided in the acknowledgement e-mail. A sample message is as follows

```
{'ID': 26045, 'RCODE': 209682}
```

If you lose the reset code, you should send an e-mail to `cs411tpserver@sabanciuniv.edu`. Your IK will be deleted from the server after 8 hours.

## 2.2 Signed Pre-key (SPK)

Signed Pre-key is another long-term key pair, which all parties generate once and is used in the registration of one-time pre-keys (OTK).

### 2.2.1 Registration of SPK

After you have registered your identity key, you are required to generate one pair of signed pre-key; $\texttt{SPK}_A\texttt{.Pub}$ and $\texttt{SPK}_A\texttt{.Pri}$. Then, you must sign the public key part of the signed pre-key, $\texttt{SPK}_A\texttt{.Pub}$ using your identity key $\texttt{IK}_A$. The signature, for which a scheme is given in Section 2.4, must be generated for the concatenated form of the public signed pre-key: $(\texttt{SPK}_A\texttt{.Pub.x} \parallel \texttt{SPK}_A\texttt{.Pub.y})$. Finally, you must send your signed pre-key to the server in the form of

```
{'ID': stuID , 'SPKPUB.X': spkpub.x, 'SPKPUB.Y': spkpub.y, 'H': h, 'S': s},
```

where `h` and `s` denote the signature tuple.

### 2.2.2 Resetting your SPK

If you lose your private signed pre-key $\texttt{SPK}_A\texttt{.Pri}$, and need to reset your signed pre-key pair, you should sign your `stuID` using your identity key $\texttt{IK}_A$ and send a message to the server to delete your public identity key $\texttt{SPK}_A\texttt{.Pub}$ from the server. The message format is

```
{'ID': stuID, 'H': h, 'S': s}
```

## 2.3 One-time Pre-key (OTK)

One-time Pre-keys are the keys which are used to generate symmetric session keys in communication with other clients. Therefore, each client in the system must register his/her OTKs to the server before the communication.

### 2.3.1 Generating HMAC Key ($\texttt{K}_{\texttt{HMAC}}$)

In the registration of OTKs, a hash-based MAC (HMAC) function will be used for authentication to provide deniability (instead of digital signature). Therefore, you should generate a symmetric HMAC Key ($\texttt{K}_{\texttt{HMAC}}$) before the registration of OTKs. $\texttt{K}_{\texttt{HMAC}}$ will be computed as follows:

- `T = SPK_A.Pri` $\cdot$ `IK_S.Pub` (Diffie-Hellman with `SPK` of the client and the `IK` of the server)
- `U` = {`b'TheHMACKeyToSuccess'` ∥ `T.y` ∥ `T.x`}
- $K_{\texttt{HMAC}}$ = `SHA3_256`(U)

#### 2.3.2 Registration of `OTK`s

Before communicating with other clients, you must generate 10 one-time public and private key pairs, namely $\texttt{OTK}_{A_0}, \texttt{OTK}_{A_1}, \ldots, \texttt{OTK}_{A_9}$. The key generation is described in the "Key Generation" algorithm in Section 2.4.

Then, you must compute an HMAC value for each of your public one-time pre-key $\texttt{OTK}_{\texttt{A}_i}$ using the `HMAC-SHA256` function and $\texttt{K}_{\texttt{HMAC}}$ as the key. The HMAC value must be generated for concatenated form of the one-time public keys (e.g., ($\texttt{OTK}_{A_i}$`.Pub.x` ∥ $\texttt{OTK}_{A_i}$`.Pub.y`) for the $i^{th}$ one-time pre-key). Finally, you must send your one-time public keys to the server in the form of

`{'ID': stuID, 'KEYID':i , 'OTKI.X': OTKi.x, 'OTKI.Y': OTKi.y, 'HMACI': hmaci},`

where `i` is the ID of your one-time pre-key. You must start generating your one-time pre-keys with IDs from 0 and follow the order.

#### 2.3.3 Resetting your `OTK`s

If you lose the private part of your one-time pre-keys, and need to reset your one-time pre-key pairs, you should sign your `stuID` using your identity key `IK` and send a message to the server to delete your registered one-time pre-keys. The message format is

`{'ID': stuID, 'H': h, 'S': s}`

### 2.4 The Digital Signature Scheme

Here, you will develop a Python code that includes functions for signing given any message and verifying the signature. For this digital signature scheme, you will use an algorithm, which consists of three functions as follows:

- **Key Generation:** A user picks a random secret key $0 < s_A < n - 1$ and computes the public key $Q_A = s_A P$.
- **Signature Generation:** Let $m$ be an arbitrary length message. The signature is computed as follows:
  1. $k \leftarrow \mathbb{Z}_n$, (i.e., $k$ is a random integer in $[1, n-2]$).
  2. $R = k \cdot P$
  3. $r = R.x \pmod{n}$, where $R.x$ is the $x$ coordinate of $R$
  4. $h = \text{SHA3\_256}(r||m) \pmod{n}$
  5. $s = (k - s_A \cdot h) \pmod{n}$
  6. The signature for $m$ is the tuple $(h, s)$.

- **Signature Verification:** Let $m$ be a message and the tuple $(s, h)$ is a signature for $m$. The verification proceeds as follows:

  1. $V = sP + hQ_A$
  2. $v = V.x \pmod{n}$, where $V.x$ is x coordinate of $V$
  3. $h' = \text{SHA3\_256}(v||m) \pmod{n}$
  4. Accept the signature only if $h = h'$
  5. Reject it otherwise.

**IMPORTANT:**

- This step is about generating the necessary keys for our communication protocol. Therefore, make sure to run each step alone. For instance, if you already generated your `Identity Key`, then you moved to the part where you generate the `Signed Pre-Key`, you make sure you do not run the previous code of generating `IK` again.

- **WHAT CAN GO WRONG?** The reason behind the previous note is that you may end up asking the server to send you the `rcode` multiple times (of course after resetting your IK). After a while, you may not see the message (falling within the SPAM box). Anyhow, you can test all the functions at once if you have completed the respective phase and want to see if everything is working flawlessly.

# 3 Phase II: Developing software for receiving messages from other clients

In this phase of the project, you are required to upload one file: "Client_phase2.py". You will be provided with "Client_basic_phase2.py", which includes all required communication codes. Moreover, you will find sample outputs for this phase in 'sample_run.txt', which is also provided on SUCourse.

You are required to develop a software for downloading five (5) messages from the server, which were uploaded to the server originally by a pseudo-client, which is implemented by us, in this phase[1]. The details are given below.

In Phase I, you have already implemented the registration protocol. The details are explained in Section 2. If you have not implemented yet, you must implement the protocol before Phase II and register your long-term public key with the server.

[2]In this protocol, a session key, $\mathtt{K_S}$, is generated (the details will be explained in Section 3.1.1) for each message block[2] using ECDH. Then, one encryption and one HMAC key are generated from $\mathtt{K_S}$ for each message in the message block using a key derivation function `KDF` chain (the details will be explained in Section 3.1.2). The messages are encrypted using AES256-CTR and authentication is provided using HMAC-SHA256[3].

[1]This is indeed an asynchronous messaging application, whereby other users can send you a message even if you are not online. Yes, exactly like WhatsApp application.

[2]Message block refer to a group of messages, which are sent by a client consecutively without receiving any message in between from the other party. You can think of WhatsApp as an example. You may send 3 messages consecutively to your friend. Then, s/he may send 2 messages. Finally, you may send 4 messages. And so on. Each of them is called a message block.

[3]Note that, SHA3_256 is used for key generation. On the other hand, SHA2_256 is used for MAC computation.

### 3.1 End-to-End Secure Communication Scheme

The protocol, which you implement in this project, provides end-to-end security in communication. In other words, no other party (including the server) can read and/or modify original messages. Moreover, forward secrecy is another feature of the protocol. Therefore, long-term keys should not be used for encryption. In order to achieve the end-to-end secure communication with forward secrecy, the communicating parties must use one-time keys to compute a session key ($\mathtt{K_S}$). Then, they must generate an encryption and an HMAC key for each message of the corresponding message block using a key derivation function (KDF) chain. Finally, they encrypt the messages using AES256-CTR for confidentiality and compute the HMAC-SHA256 value of the ciphertext for integrity.

#### 3.1.1 Session Key ($\mathtt{K_S}$)

As mentioned before, both parties should establish a session key ($\mathtt{K_S}$) for a message block to generate encryption and HMAC keys at first. $\mathtt{K_S}$ is computed using X3DH (Extended Triple Diffie-Hellman) key exchange protocol. Therefore, when a user wants to send a message block to another user, s/he requests the *pre-key bundle* of the receiver from the server. The pre-key bundle contains the public IK, public SPK and one of the OTKs of the receiver. Then, the sender generates a DH key pair which is called as "Ephemeral Key (EK)" and it is used in the DH key exchange along with the sender's public IK and receiver's IK, SPK, and OTK. The computation of ($\mathtt{K_S}$) in the receiver side is given below:

- $\mathtt{T1 = IK_B.Pub \cdot SPK_A.Pri}$[4]
- $\mathtt{T2 = EK_B.Pub \cdot IK_A.Pri}$
- $\mathtt{T3 = EK_B.Pub \cdot SPK_A.Pri}$
- $\mathtt{T4 = EK_B.Pub \cdot OTK_A.Pri}$
- $\mathtt{U} = \{\mathtt{T1.x} \parallel \mathtt{T1.y} \parallel \mathtt{T2.x} \parallel \mathtt{T2.y} \parallel \mathtt{T3.x} \parallel \mathtt{T3.y} \parallel \mathtt{T4.x} \parallel \mathtt{T4.y} \parallel \mathtt{b'WhatsUpDoc'}\}$ [5]
- $\mathtt{K_S} = \mathtt{SHA3\_256}(\mathrm{U})$

where A refers to the receiver and B refers to the sender. For example, $\mathtt{EK_B.Pub}$ is public ephemeral key of the sender, and $\mathtt{OTK_A.Pri}$ is the private OTK of the receiver.

#### 3.1.2 Key Derivation Function (KDF) Chain

The Key Derivation Function (KDF) is a cryptographic function that takes a secret KDF key ($\mathtt{K_{KDF}}$) and returns an encryption key, an HMAC key and KDF key for the next iteration. If there are more than one message in a message block, KDF is used more than once to generate different encryption and HMAC keys for each message. The outputs of KDF are computed as follows:

- $\mathtt{K_{ENC}} = \mathtt{SHA3\_256(K_{KDF}} \parallel \mathtt{b'JustKeepSwimming')}$[6]

[4] The sender will compute $\mathtt{T1 = IK_B.Pri \cdot SPK_A.Pub}$. If you are confused, check how the Diffie-Hellman Key Exchange Protocol is working. Except T1, T2, T3, and T4, the rest of the computation is the same for the sender.
[5] https://youtu.be/QBcK63h3Avw?si=sDh4_iqceebeYVkj
[6] https://youtu.be/0Hkn-LSh7es?si=Nfw2oCBrS8x0x_KF

- $\mathtt{K}_{\mathtt{HMAC}}$ = SHA3_256($\mathtt{K}_{\mathtt{KDF}} \parallel \mathtt{K}_{\mathtt{ENC}} \parallel$ b'HakunaMatata')
- $\mathtt{K}_{\mathtt{KDF.Next}}$ = SHA3_256($\mathtt{K}_{\mathtt{ENC}} \parallel \mathtt{K}_{\mathtt{HMAC}} \parallel$ b'OhanaMeansFamily')

After $\mathtt{K}_{\mathtt{S}}$ is computed, it will be used as the KDF key for the first message. If there is only one message in the message block, $\mathtt{K}_{\mathtt{ENC}}$ and $\mathtt{K}_{\mathtt{HMAC}}$ will be used for encryption and HMAC values, respectively. However, $\mathtt{K}_{\mathtt{KDF.Next}}$ will not be used (only if there is one message in the block). Otherwise, $\mathtt{K}_{\mathtt{KDF.Next}}$ is used as the KDF key to generate $\mathtt{K}_{\mathtt{ENC}}$ and $\mathtt{K}_{\mathtt{HMAC}}$ for the next message of the message block. Figure 1 shows how the KDF chain is working for a message block which contains 3 messages, $\mathtt{m}_1$, $\mathtt{m}_2$ and $\mathtt{m}_3$ namely[7].

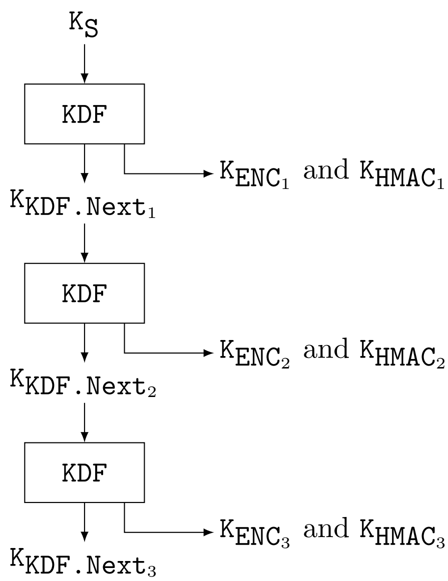


Figure 1: KDF Chain for a message block with 3 messages

In this example, $\mathtt{K}_{\mathtt{S}}$ is computed by both parties before the communication and used as $\mathtt{K}_{\mathtt{KDF}}$ to generate $\mathtt{K}_{\mathtt{ENC}_1}$ and $\mathtt{K}_{\mathtt{HMAC}_1}$, where $\mathtt{K}_{\mathtt{ENC}_1}$ and $\mathtt{K}_{\mathtt{HMAC}_1}$ are the encryption and HMAC keys of $\mathtt{m}_1$, respectively. $\mathtt{K}_{\mathtt{KDF}_1}$ is used to generate $\mathtt{K}_{\mathtt{ENC}_2}$ and $\mathtt{K}_{\mathtt{HMAC}_2}$ for $\mathtt{m}_2$. And so on...

### 3.2 Downloading Messages from the server

As mentioned above, you will download 5 messages from the server. In order to download one message from the server, you must sign your ID using your Identity Key and send a message to the server in the form of

{'ID': stuID, 'S': s, 'H': h }

The message will be downloaded in the form of

{'IDB': stuIDB, 'OTKID': otkID, 'MSGID': msgID, 'MSG': msg , 'IK.X': IK.x ,'IK.Y': IK.y, 'EK.X': EK.x ,'EK.Y': EK.y },

[7]For convenience in implementation, message index starts from 1 instead of 0.

where `stuIDB` is the ID of the sender, `otkID` is the ID of your one-time prekey, which is used to generate session key, `msgID` indicates the order number of the message in the corresponding message block, `msg` contains the nonce, the ciphertext and MAC value of the ciphertext[8], and `EK.x` and `EK.y` are $x$ and $y$ coordinates of the ephemeral public key (`EK.Pub`) of the sender, respectively. Similarly, `IK.x` and `IK.y` are $x$ and $y$ coordinates of the public identity key (`IK.Pub`) of the sender (**Note:** there is only one message format. Therefore, all messages in the same message block contain the same `otkID` and $\mathtt{EK}_{\mathtt{B}}$`.Pub` information.)

### 3.3 Decrypting the Messages From the Pseudo-Client

In this phase, you will request and receive **five** messages from a pseudo-client, which simulates your correspondence sending messages to you. Therefore, you need to request to generate 5 messages from pseudo-client (via the server) at first. The message format is

{'ID': stuID, 'H': h, 'S': s}

When you request five messages, the messages will be ready on the server and you may download them (explained in Section 3.2). After you download the messages, you need to compute $\mathtt{K}_{\mathtt{S}}$ and `KDF` chain at first. Then, you should check the MAC values of the messages and decrypt them (4 messages will have valid MAC values, but one of them will not.). Finally you should send a message for each received message, which you downloaded, in the form of:

{'IDA': stuIDA, 'IDB': stuIDB, 'MSGID': msgID, 'DECMSG': decmsg },

where `stuIDA` and `stuIDB` are the ID of you and the pseudo-client, respectively. `msgID` is the ID of the message (The order of the sent messages will be the same as the received messaged. In other words, you should put the same message ID with the received message.) and `decmsg` is the decrypted form of the message if the MAC value is valid. Otherwise, you should set `decmsg` field to `"INVALIDHMAC"`.

### 3.4 Displaying the Final Message Block

After receiving the messages from the pseudo-client and **decrypting them successfully**, you are asked to ask the server about whether the pseudo-client has deleted some messages from the received block or not. This case is similar to the case in WhatsApp messages, a sender may delete some messages after sending them. You won't implement the exact scenario. However, you will display the messages of the received block as given in the sample run. Here are the steps:

1. Send the server a request that includes your `stuID` and its signature using your `IK` in the form:
   {'ID': stuID, 'H': h, 'S': s}
   It is the same request as in Step 3.3 but uses a different service (given to you in the `Client_basic_phase2.py`).

[8] `msg = nonce‖ciphertext‖MAC`
Where the nonce is the AES-CTR nonce of 8 bytes. You can conclude the size of the MAC value from the used hash function algorithm.

2. After the server verifies your signature, then it will send (as a result to your request) a list that has the message ids (`MSGID`) of the ones that got deleted.

3. Finally, you will display the messages in the same manner given in the sample run.

4. **Note:** Bear in mind that there is an invalid message among the block. You should not print it as you could not verify its signature. However, the pseudo-client can delete it, and you may receive its `MSGID` as well.

# 4 Phase III: Developing software to communicate with other clients

In this phase of the project, you are required to upload one file: "Client_phase3.py". You will be provided with "Client_basic_phase3.py", which includes all required communication codes. Moreover, you will find sample outputs for this phase in 'sample_vector.txt', which is also provided on SUCourse.

For testing purposes, you may send a message to your groupmate and s/he may receive your message, if you are working in groups of two. Note that, if you have not registered with the server yet, you must register. The details for the registration are given in Section 2. Moreover, you must be able to receive messages from other users (Explained in Section 4). If you are working alone, you may communicate with `18007` (the ID of pseudo-client).

In this phase of the project, you will work on two parts. In the first part, you will develop software to send messages to other users. (You are not limited to sending messages to your groupmate or pseudo-clients. You may send messages to anybody who takes CS411/507.) In the second part, you will manage your `OTK`s. The details are given in the subsequent sections.

## 4.1 Sending Messages

In this part of Phase III, you are required to develop software to send messages to other users. Firstly, you are required to generate a session key $\mathtt{K_S}$. The details of session key generation are given in Section 3.1.1. To generate $\mathtt{K_S}$ to send a message to another user, you need the receiver's pre-key bundle, which consists of the receiver's identity key, signed pre-key, and one one-time pre-key. Therefore, you must request the pre-key bundle of the receiver from the server. The message format is given below.

{`'IDA': stuIDA, 'IDB': stuIDB, 'S': s, 'H': h`}

where `stuIDA` and `stuIDB` are your and the receiver's IDs, respectively. `s` and `h` is a signature tuple which is generated for `stuIDB` using your long-term private key. If your request is valid, the server will send the pre-key bundle of the receiver as follows:

{`'KEYID': i, 'IK.X': IK.x, 'IK.Y': IK.y, 'SPK.X': SPK.x, 'SPK.Y': SPK.y, 'H': h, 'S': s, 'OTK.X': OTK.x, 'OTK.Y': OTK.y`}

where `i` denotes the `OTK` ID of the receiver (you will use it in your message to identify which `OTK` of the receiver you are using). Moreover, `OTK.x` and `OTK.y` are $x$ and $y$ coordinates of the $\mathtt{i}^{\mathtt{th}}$ `OTK` of the receiver. Note that the signature `(H, s)` is generated for the concatenated form of the public signed pre-key ($\mathtt{SPK}_A\mathtt{.Pub.x} \parallel \mathtt{SPK}_A\mathtt{.Pub.y}$).

Upon receiving the pre-key bundle, you need to verify the pre-key signature. If verification succeeds, you should generate the ephemeral key pair EK and compute $K_S$.

After, you computed the $K_S$, you are required to create a KDF chain (Explained in Section 3.1.2). Then, you should encrypt your message/s and compute MAC values using the encryption and MAC key/s. Finally, you should send your message to the server in the form of

{'IDA': stuIDA, 'IDB': stuIDB, 'OTKID': otkID, 'MSGID': msgID, 'MSG': msg, 'IK.X': IK.x ,'IK.Y': IK.y, 'EK.X': EK.x ,'EK.Y': EK.y}

where stuIDA and stuIDB are the IDs of you and the receiver, respectively. otkID denotes the OTK ID of the receiver which is sent you by the server. msgID indicates the message index in the corresponding message block. msg includes the nonce, ciphertext and MAC of the ciphertext. EK.x and EK.y are $x$ and $y$ coordinates of your EK. IKPUB.x and IKPUB.y are $x$ and $y$ coordinates of your public IK. (**Note:** If there are more than one message in a message block, you should use the same otkID and EK information in all messages of the same message block.)

## 4.2 Status Control

As indicated in several places in previous sections, the protocol, which you are implementing in this project is an offline protocol. In other words, the messages which are sent by the sender are stored in the server until the receiver requests to retrieve his/her messages. Moreover, OTKs, which are generated by the users, are used once (for only one message block) to generate a new session key ($K_S$). Therefore, you are required to check the server for how many new messages you have and how many OTKs are remaining. Moreover, you must generate new OTKs and register with the server if needed. To be informed of new messages and remaining OTKs, you must send your ID to the server as follows:

{'ID': stuID, 'S': s, 'H': h }

where s and h is a signature tuple, which is generated for stuID using your long-term private key. If your request is valid, the server will send the number of new messages and remaining OTKs. Then, you may either get your messages from the server or register your new OTKs with the server.

### 4.2.1 Registration of new OTKs

In Phase I, you have generated 10 OTKs and registered with the server and each OTK is used for one message block. On the other hand, you may receive more than 10 message blocks. Therefore, if some of your OTKs are used, you must generate new OTKs to reach a total of 10 keys then register them in the server again. The registration protocol is explained in section 2.3.2 As mentioned in section 2.3.2, you must follow the order for key IDs.

As you have probably experienced in the previous phases, you keep receiving either an Internal Server Error or another message from the server saying that you ran out of OTKs. As a result, you will have to re-generate new OTKs and demonstrate that in your output.

## 4.3 Sending messages to pseudo-clients

The task, you should perform for Phase III, is similar to Phase II (Explained in Section 3.3). In Phase II, you have received 5 messages from a pseudo-client, decrypt them then check which message(s) was(were) deleted. In this phase, you will -also- re-encrypt [9] the messages you received and send them back to the pseudo-client whose ID number is `18007`.

### 4.3.1 The process flow

You should download 5 messages from the server and decrypt them. Therefore, you should complete Phase II before Phase III. Then, you are asked to create a session with the pseudo-client. Later on, you should encrypt the messages you received with valid MAC values. Finally, you will send the encrypted messages back to the pseudo-client (i.e. ID: `18007`) in the same order. The details are explained in Section 3.1 and Section 4.1.

## 5 Bonus (10 pts)

In the Bonus part of the project, we expect you to generate a conference key with 3 other users for group communication. The other 3 users are pseudo-clients who you will be able to exchange values with via the server. In the end, all 4 users (you and the other 3 pseudo-clients) should agree on the same conference key, which would be used for the communication between the group participants. However, for this part, you are not required to implement the communication protocol using the generated conference key.

In the conference setting, you are referred to as $U_1$, and the rest of the group members are $U_2$, $U_3$, and $U_4$, respectively. You will follow a similar conference keying algorithm as given in the lecture slides. However, this algorithm will be adapted to the Elliptic Curve setting.

Initially, each group member generates a private and public partial key pair, $r_i$ and $z_i$, using the key generation algorithm given in Section 2.4. You must sign your public partial key $z_1$ using your identity key $\mathtt{IK}_A$. The signature, for which a scheme is given in Section 2.4, must be generated for the concatenated form of the public partial key: $(\mathtt{z}_1\mathtt{.x} \| \mathtt{z}_1\mathtt{.y})$. Finally, you must send your signed partial key to the server in the form given below using the `ExchangePartialKeys` service shared with you in `Client_basic_phase3.py`.

```
{'ID': stuID , 'z1.X': z1.x, 'z1.Y': z1.y, 'H': h, 'S': s},
```

This function call will also return you the public partial keys of the other users, specifically of the users $U_2$ and $U_4$.

After receiving these values, you are expected to calculate and broadcast $x_1$ using the `ExchangeXs` service call in the `Client_basic_phase3.py`. Again, you should sign the concatenated form of $x_1$ $(\mathtt{x}_1\mathtt{.x} \| \mathtt{x}_1\mathtt{.y})$ using your identity key $\mathtt{IK}_A$ an send your $x_1$ value in the form of:

```
{'ID': stuID , 'x1.X': x1.x, 'x1.Y': x1.y, 'H': h, 'S': s},
```

Once again, this function call will also return you the $x_i$ values of the other participants.

Finally, you are expected to generate the conference key and check if your key is correct using the `BonusChecker` service.

[9] Check the API of AES under PyCryptodome for more details about encryption & decryption: https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html

# 6 Appendix I: Timeline, Deliverables, Weight & Policies

| Project Phases | Deliverables | Due Date | Weight |
|---|---|---|---|
| Project announcement | | 14/12/2023 | |
| First Phase | File: Client_phase1.py | 24/12/2023 | 30% |
| Second Phase | File: Client_phase2.py | 31/12/2023 | 40% |
| Third Phase | File: Client_phase3.py | 07/01/2024 | 30% |

## 6.1 Policies

- The only URL that you will send queries to is: `http://harpoon1.sabanciuniv.edu:9999/`
- You may work in groups of two.
- Submit all deliverables in the zip file "cs411_507_tp3_surname1_surname2.zip".
- Only one submission from one student is enough. However, the $2^{nd}$ policy point should be respected.
- For codes, only .py scripts should be uploaded.
- **Points will be deducted if you don't respect the naming and file format criteria.**
- You may be asked to demonstrate a project phase to a TA or the instructor.
- Since you will interact with a remote server to register your keys, send and receive messages. This server script itself is a validation method to check whether your codes are working correctly or not. We will also use it to check your implementation. If your implementation in a project phase fails to pass the validation, you will get no credit for that phase.
- Your codes will be checked for their similarity to other students' codes; and if the similarity score exceeds a certain threshold you will not get any credit for your work.