Implementing Signal Protocol

Cryptography CS 411 & CS 507 Term Project for Fall 2021

E. Savaş
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

All coding development will be in the Python programming language. More information about the project is given in the subsequent sections.
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:93223115898197558905062012489877327981787036929201444813217704012422483432813
Y:89856292032257671854649200941983642557409873467439120718433039755876953376l9
```

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., 18007). 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. A sample message is given in 'samples.txt'.

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': 18007, '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).)

2.1.2 Reseting 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': 18007, '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 Prekey 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. If your signed pre-key is registered successfully, the server will return its signed pre-key $\texttt{SPK}_S\texttt{.Pub}$ in the same format. After you check the validity of the signature of $\texttt{SPK}_S\texttt{.Pub}$, you may use it.

2.2.2 Reseting 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 ($\mathtt{K_{HMAC}}$)

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 ($\mathtt{K_{HMAC}}$) before the registration of OTKs. $\mathtt{K_{HMAC}}$ will be computed as follows:

- T = SPK_A.Pri · SPK_S.Pub (Diffie-Hellman with SPKs of the client and the server)
- U = {T.x ‖ T.y ‖ b'NoNeedToRideAndHide'} [1]
- $K_{\mathtt{HMAC}}$ = SHA3_256(U)

2.3.2 Registration of OTKs

Before communicating with other clients, you must generate 10 one-time public and private key pairs, namely $\mathtt{OTK}_{A_0}, \mathtt{OTK}_{A_1}, \ldots, \mathtt{OTK}_{A_9}$. 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 $\mathtt{OTK}_{\mathtt{A}_i}$ using using the HMAC-SHA256 function and $\mathtt{K_{HMAC}}$ as the key. The HMAC value must be generated for concatenated form of the one-time public keys (e.g., ($\mathtt{OTK}_{A_i}$.Pub.x ‖ $\mathtt{OTK}_{A_i}$.Pub.y) for the i^{th} one-time pre-key). Finally, you must sent 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 Reseting your OTKs

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 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] https://www.youtube.com/watch?v=u1ZoHfJZACA

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.

Note that the signature generation and verification of this scheme are different from the one discussed in the lecture.

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_vector.txt', which is also provided on SUCourse.

You are required to develop a software for downloading 6 messages from the server, which were uploaded to the server originally by a pseudo-client/s, which is implemented by us, in this phase[2]. 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.

In this protocol, a session key, $\texttt{K}_\texttt{S}$, is generated (will be explained in Section 3.2.1) for each message block [3] using ECDH. Then, one encryption and one HMAC key are generated from $\texttt{K}_\texttt{S}$ for each message of the block using a key derivation function `KDF` chain (will be explained in Section 3.2.2). The messages are encrypted using AES128-CTR and authentication is provided using HMAC-SHA256.

[2]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.

[3]Message block states a group of messages which are sent by a client consecutively without receiving any message from the other party. Let you think about Whatsapp. 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 as message block.

3.1 Downloading Messages from the server

As mentioned above, you will download 6 messages from the server. In order to download one message from the server, you must sign your ID using your long-term private 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 , 'EK.X': EK.x ,'EK.Y': EK.y },

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 both the ciphertext and its MAC value, and `EK.x` and `EK.y` are x and y coordinates of the ephemeral public key ($\mathtt{E_K.Pub}$) of the sender, respectively.

3.2 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. In order to achieve the secure communication, the communicating parties will use ECDH to compute a session key $\mathtt{K_S}$ for a message block. Then, they must generate an encryption and an HMAC key for each message in the corresponding message block using a key derivation function (`KDF`) chain. Finally, they encrypt the messages using AES128-CTR for confidentiality and compute the HMAC-SHA256 value of the ciphertext for authentication.

3.2.1 Session Key ($\mathtt{K_S}$)

As mentioned, both parties should compute a session key ($\mathtt{K_S}$) for a message block to generate encryption and HMAC keys at first. The computation of ($\mathtt{K_S}$) is given below for the receiver:

- $\mathtt{T = OTK\_A.Pri \cdot EK\_B.Pub}$
- $\mathtt{U} = \{\mathtt{T.x} \parallel \mathtt{T.y} \parallel \mathtt{b'MadMadWorld'}\}$ [4]
- $\mathtt{K_S} = \mathtt{SHA3\_256}(\mathtt{U})$

In the given computation, $\mathtt{OTK_A.Pri}$ denotes the private `OTK` of the receiver. The receiver may identify his/her `OTK` by checking `otkID` field in the message. $\mathtt{EK_B.Pub}$ is public ephemeral key of the sender. The ephemeral key is a one-time key (unique for each message block) and provided in the message. (**Note:** there is only one message format. Therefore all messages in the same message block contain the same `otkID` and $\mathtt{EK_B.Pub}$ information.)

[4] `https://www.youtube.com/watch?v=Vmq-nWlT8xw`

3.2.2 Key Derivation Function (KDF) Chain

Key Derivation Function (KDF) is a cryptographic function that takes a secret KDF key 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:

- K_{ENC} = SHA3_256(K_S || b'LeaveMeAlone')
- K_{HMAC} = SHA3_256(K_{ENC} || b'GlovesAndSteeringWhell')
- $K_{KDF.Next}$ = SHA3_256(K_{HMAC} || b'YouWillNotHaveTheDrink')

After K_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, K_{ENC} and K_{HMAC} will be used for encryption and HMAC values, respectively. $K_{KDF.Next}$ will not used. Otherwise, $K_{KDF.Next}$ is used as KDF key to generate K_{ENC} and K_{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, m_1, m_2 and m_3 namely.

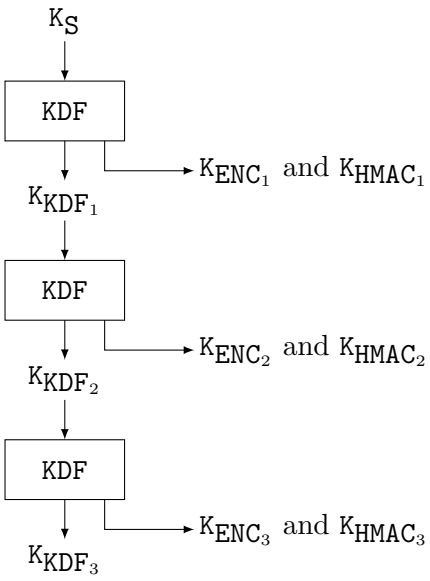

Figure 1: KDF Chain for a message block with 3 messages

In this example, K_S is computed by both parties before the communication and used as K_{KDF} to generate K_{ENC_1} and K_{HMAC_1}, where K_{ENC_1} and K_{HMAC_1} are the encryption and HMAC keys of m_1, respectively. K_{KDF_1} is used to generate K_{ENC_2} and K_{HMAC_2} for m_2. And so on...

3.3 Decrypting the messages for grading

In this phase, you will receive 6 messages from two pseudo-clients (3 message from each). After you download the messages, you need to compute K_Ss and KDF chains at first. Then, you should check the MAC values of the messages and decrypt them (5 messages will have valid MAC values, but one of them will not.). Finally you should send a message for each message, which you downloaded, in the form of:

```
{'IDA': stuIDA, 'IDA': stuIDA, 'MSGID': msgID, 'DECMSG': decmsg ,},
```

where `stuIDA` and `stuIDB` are the ID of you and the other party, respectively. `msgID` is the ID of the message (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` value to `"INVALIDHMAC"`.

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 purpose, you may send 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 are given in Section 2. Moreover, you must be able to receive messages from other clients(Explained in Section 3.If you are working alone, you may communicate with "18007" and "XXXXX", which are pseudo-clients.

In this phase of the project you will work on two parts. In the first part, you will develop a software to send message to other users.(You are not limited to send message to your group-mate or pseudo-clients. You may send everybody who takes CS411/507.). In second part, you will manage your one-time pre-keys. The details are given in the subsequent sections.

4.1 Sending Messages

In this part of Phase III, you are required to develop a software to send messages to other clients. Firstly, you are required to generate session key K_S. The details of session key generation is given in Section 3.2.1. In order to generate K_S to send a message to another client, you need a one-time pre-key of the receiver. Therefore, you must request a one-time pre-key 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 generated for `stuIDB` using your long-term private key. If your request is valid, the server will send the `OTK` of the receiver as follows:

```
{'KEYID': i, 'OTK.X': OTK.x, 'QTK.Y': OTK.y, }
```

where `i` denotes the one-time pre-key ID the receiver (you will put it to your message to identify which `OTK` of the receiver you are using). Moreover, `OTK.x` and `OTK.y` are x and y coordinated of the i^{th} `OTK` of the receiver.

After, you computed the K_S, you are required to create a `KDF` chain (Explained in Section 3.2.2). 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 server in the form of

{‘IDA’: stuIDA, ‘IDB’: stuIDB, ‘OTKID’: otkID, ‘MSGID’: msgID, ‘MSG’: msg, ‘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. (**Note:** if there are more than one message in a message block, you should put the same otkID and EK information to all messages of the message block.)

4.2 Status Control

As explained in several 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 get his/her messages. Moreover, OTKs, which are generated by the clients, are used once (for only one message block) to generate a new session key pair (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. In order to be informed for new messages and remaining ephemeral keys, 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 ephemeral keys. Then, you may either get your messages from the server or register your new ephemeral keys with the server.

4.2.1 Registration of new OTKs

In Phase I, you have generated 10 ephemeral keys and registered with the server and each OTK is used for one message block. On the other hand, you may receive more than 10 different clients. Therefore, if some of your OTKs are used, you must generate new OTKs to cover to 10 and register with the server again. The registration protocol is explained in section 2.3.2

On the other hand, more than 10 sessions may be created for a client, when s/he is offline. In this case, his/her last OTK will be used more than once. For instance, if 12 sessions are created for an offline client, his/her first 9 OTKs will be used for the first 9 sessions and his/her 10^{th} OTK will be used for last 3 sessions. As mentioned in section 2.3.2, you must follow the order for key IDs.

4.3 Sending message to pseudo-client for grading

The task you should demonstrate for Phase III is similar with Phase II (Explained in Section 3.3). First of all, you should download 6 messages from the server and decrypt them. The messages are from 2 pseudo-clients (3 messages from each). All messages will be meaningful messages with random numbers and have valid HMAC values. Therefore, you should complete Phase II before Phase III.

5 Appendix I: Timeline & Deliverables & Weight & Policies etc.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement		10/12/2021	
First Phase	File: Client.py	17/12/2021	40%
Second Phase		24/12/2021	30%
Third Phase		31/12/2021	30%

5.1 Policies

- You may work in groups of two.
- Submit all deliverables in the zip file "cs411_507_tp1_yourname.zip".
- You may be asked to demonstrate a project phase to a TA or the instructor.
- In every phase, we will provide you with a validation software in Python language that can be used to check your implementation for correctness. 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.