**Security and Vulnerability in Networks**

University of Stavanger

# DAT 510: Assignment 2

## Secure Communication

**Author**: Daniel Linfon Ye Liu

**Year**: 2024

# Index

# Abstract

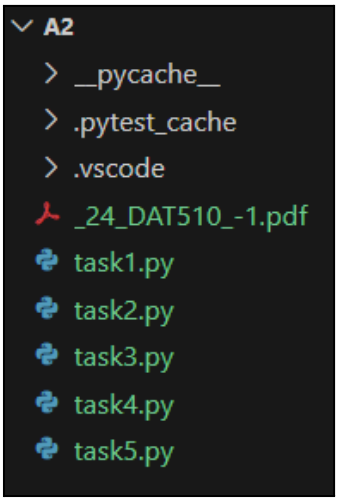
This project explores the Diffie-Hellman key exchange protocol and its use in establishing secure communications between two parties. The focus is on the mathematical foundations and cryptographic relevance of Diffie-Hellman, its implementation to generate shared secrets, and its application in modern messaging apps such as WhatsApp and Signal. In addition, the report discusses how HMAC, using the shared secret from Diffie-Hellman, enhances message authentication, ensuring integrity and authenticity. Forward secrecy is further explored using ratcheting mechanisms, culminating in the implementation of the Double Ratchet Algorithm for secure communications.

# 1. Introduction

This section introduces the concepts of cryptography and secure communication, with a focus on Diffie-Hellman key exchange. Diffie-Hellman is a fundamental protocol used in modern encryption systems, allowing two parties to securely share a secret key. This key is crucial for encrypting and authenticating communications in widely used messaging applications such as WhatsApp and Signal. The protocol relies on the difficulty of the discrete logarithm problem to secure the exchange. Additionally, this report discusses the importance of Hashed Message Authentication Code (HMAC) and ratcheting mechanisms for ensuring message authenticity and forward secrecy in secure communications.

# 2. Design and Implementation

We have decided to create a separate script for each task using **Python** in **Visual Studio Code**. Each task builds on the previous one, increasing complexity and introducing techniques to improve communication security.




***working directory***

## 2.1. Implementing Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange algorithm is a method for agreeing to and calculating a secret symmetric key between parties on an insecure channel or public network. The security of the protocol is based on the difficulty of solving the discrete logarithm problem, which makes this protocol fundamental in many secure communication systems.

To understand how it works, let's look at the key mathematical steps that allow generating a shared secret key:

1. **Selecting public parameters:**

    a. ***p***: A large prime number.
    b. ***g***: A generator or base, which is a primitive root modulo $p$. This means that the integers $g^k \bmod p$ for $k = 0,1,2,\ldots, p-2$, generate all integers from 1 to $p$-1.

    These values are public and known to both parties (Alice and Bob).

2. **Choosing private keys:**

    a. Alice chooses a secret number $a$ that will be her private key.
    b. Bob chooses a secret number $b$ that will be his private key.

3. **Computing Public Keys:**

    a. Alice computes her public key ***A*** using her private key ***a***:

    $$A = g^a \bmod p$$

    b. Bob computes his public key ***B*** using his private key ***b***:

    $$B = g^b \bmod p$$

    Both exchange their public keys $A$ and $B$. This exchange can happen openly because deriving $a$ or $b$ from $A$ or $B$ is mathematically difficult due to the discrete logarithm problem.

4. **Computing the Shared Secret**:

    a. Alice uses Bob's public key $B$ along with her private key $a$ to compute the **shared secret** $S$:

    $$S = B^a \bmod p$$

    b. Bob uses Alice's public key $A$ along with her private key $b$ to compute the **shared secret** $S$:

    $$S = A^b \bmod p$$

In our code, the Diffie-Hellman protocol was implemented by defining a large prime number ***p*** and a generator ***g***, which were publicly shared between Alice and Bob. Each party selected private keys and computed their respective public keys. After exchanging public keys, both parties computed the shared secret, which was identical due to the properties of modular exponentiation.

```
function diffie_hellman(p, g):

    # Step 1: Generate private keys (random integers)
    a ← random number between 1 and p-1  # Alice's private key
    b ← random number between 1 and p-1  # Bob's private key

    # Step 2: Generate public keys
    A ← g^a mod p  # Alice's public key (g raised to the power of a, modulo p)
    B ← g^b mod p  # Bob's public key (g raised to the power of b, modulo p)

    # Step 3: Calculate the shared secret key
    S_Alice ← B^a mod p  # Alice calculates the shared secret key
    S_Bob ← A^b mod p    # Bob calculates the shared secret key

    # Verify that both shared secret keys are the same
    verify that S_Alice = S_Bob

    print "Shared secret key: " + S_Alice

    return S_Alice
```

*Figure 1: Diffie-Hellman Key Exchange Pseudocode*

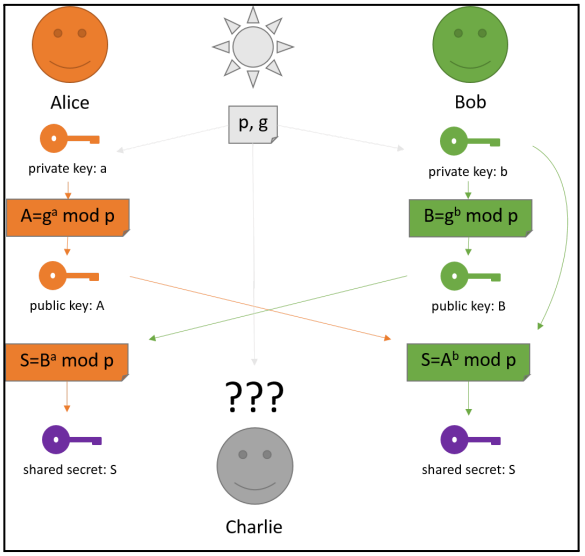



*Figure 2: Diffie-Hellman Key Exchange diagram*

## 2.2. Implementing HMAC for Authentication

After generating the shared secret, Alice and Bob used it as the key for the **HMAC** algorithm. This ensured that each message sent between them was authenticated, verifying its integrity and preventing tampering by a third party.

HMAC works by creating a fixed-size authentication tag (also called a MAC or HMAC tag) from the message and the secret key. The receiver can verify the integrity of the message by generating the same tag and comparing it with the one sent by the sender.

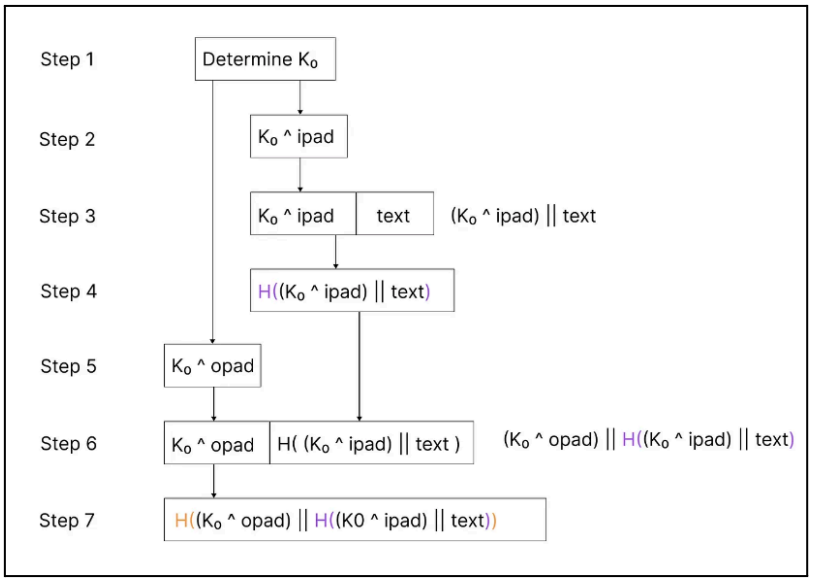


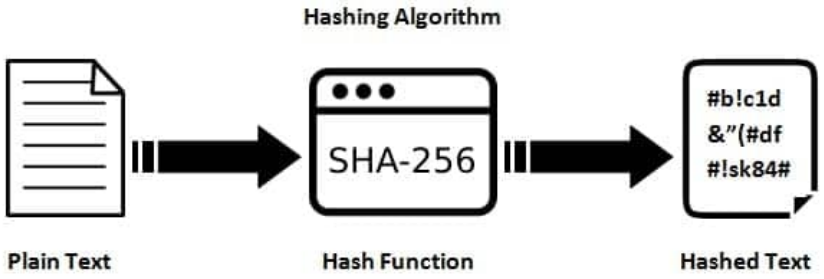

*Figure 3: HMAC diagram*

Figure 3 shows the process to generate the authentication tag. Here's how it work:

1. **Padding the Secret Key**:
    - If the secret key is **longer** than the block size of the hash function, it is **hashed** to produce a shorter key.
    - If the secret key is **shorter** than the block size, it is **padded with zeros** to match the block size.
2. **Inner and Outer Padding**:
    - ***ipad*** is a block of bytes where each byte is **0x36**.
    - ***opad*** is a block of bytes where each byte is **0x5c**.
3. **XOR the key with *ipad* and *opad***
    - First, the key is XORed with the ***ipad*** and concatenated with the **message**. This result is **hashed**.
    - Then, the key is XORed with the ***opad*** and concatenated with the previous hash. Finally, this result is hashed again and we will obtain the **HMAC tag**, which is a fixed-length string of bits used for authentication.

First, we will analyze HMAC using a hash based on the XOR operation. This simple version of HMAC uses the XOR function to generate a message authentication code from a shared key and a message. Although it is less cryptographically robust compared to more advanced hashing algorithms, it allows us to better understand the basic operation of HMAC before we move on to more secure variants.

Then, to improve security, we will use **SHA-256** instead of XOR. SHA-256 is a robust cryptographic hash function that offers collision resistance, better diffusion, and higher overall security.




## 2.3. Applying Encryption-Decryption from the Previous Assignment

We are going to implement the first option, which is *HMAC followed by Encryption.* To perform the encryption - decryption, we will use the **AES** symmetric encryption algorithm in **CTR mode**, implemented in the previous task.

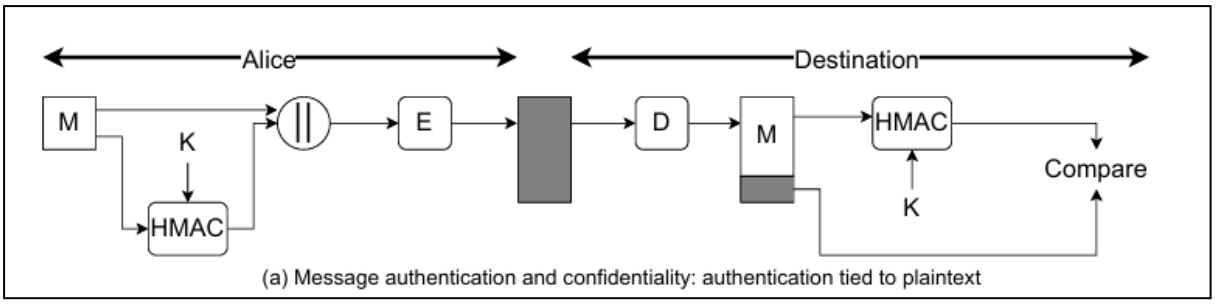



*Figure 3: HMAC followed by Encryption Diagram*

Our implementation follows the scheme in Figure 3. It is practically the same scene as the previous task but with the addition of the encryption and decryption process.

```
# Alice sends a message to Bob and computes the HMAC of the message using the shared key
message ← "Hello, Bob!"
hmac_tag ← hmac_with_sha256(shared_key, message)  # Compute HMAC using shared key and message

# Alice concatenates both the message and the HMAC tag, then encrypts it
plaintext ← message + hmac_tag

aes_key ← random 32-byte value  # Generate a random AES key
nonce, ciphertext ← encrypt_CTR(plaintext, aes_key)  # Encrypt plaintext using AES in CTR mode

# Output the encrypted message and nonce
print "Encrypted message: " + ciphertext
print "Nonce: " + nonce

# Bob receives the encrypted message from Alice. He decrypts it to get the message and the HMAC tag
decrypted_message_with_hmac ← decrypt_CTR(ciphertext, aes_key, nonce)

# Separate the original message and the HMAC tag
original_message ← decrypted_message_with_hmac[:-64]  # The message is everything before the last 64 characters
received_hmac_tag ← decrypted_message_with_hmac[-64:]  # The last 64 characters are the HMAC tag

# Output the original message and the received HMAC tag
print "Original Message: " + original_message
print "Received HMAC tag: " + received_hmac_tag
```

*Figure 4: HMAC followed by Encryption Pseudocode*

The text we encrypt is composed of Alice's message and the HMAC tag of the message. In this way, Bob decrypts the message and computes the HMAC of the received message. Finally, he compares both HMAC tags and if they match, it means that the message has been authenticated.

## 2.4. Implementing Single Ratchet for Chain Key

The **Single Ratchet** mechanism is a key component in ensuring **forward secrecy** in secure communication protocols. It ensures that each message between two parties uses a unique key by evolving the cryptographic keys over time. This mechanism guarantees that even if a session key is compromised, both past and future communications remain secure.

The Single Ratchet mechanism works by continuously updating a **chain key** after each message is sent or received. The updated chain key generates a fresh message key for each message.

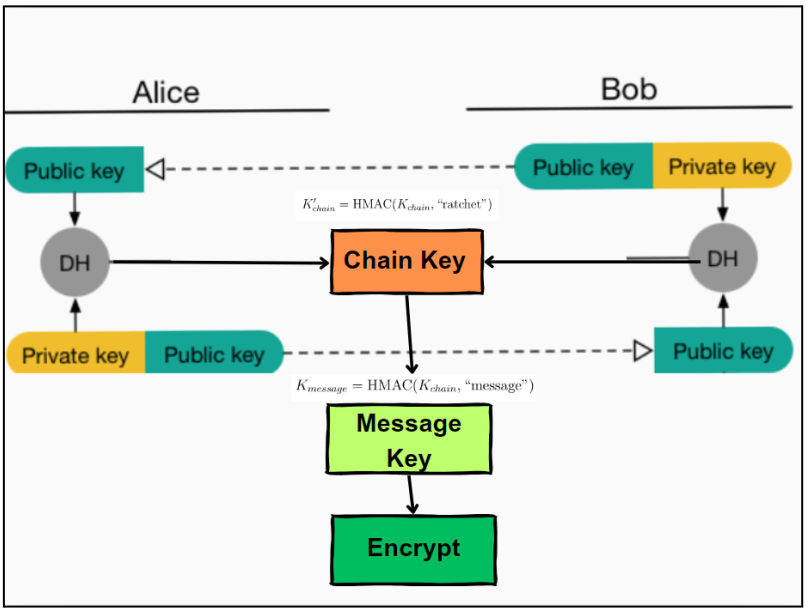



*Figure 5: Single Ratchet Mechanism*

To integrate this mechanism into our project, we developed the **`SingleRatchet`** class. Below is an overview of how the Single Ratchet mechanism functions and how it has been implemented:

1. **Initial Chain Key**:
    - Alice and Bob begin with an initial chain key, $K_{chain}$, which is derived from a shared Diffie-Hellman secret..
2. **Chain Key Ratcheting**:
    - Every time a message is sent or received, the current chain key is updated using an HMAC-based **pseudo-random function** (PRFG) to derive a new chain key.
    - This update is performed using the following function:

$$K'_{chain} = HMAC(K_{chain}, "ratchet")$$

Here, $K_{chain}$ is the current chain key, and ***"ratchet"*** is an optional label to differentiate the key derivation step.

3. **Message Key Derivation**:
    - After updating the chain key, a **message key** is derived from the newly updated chain key:

$$K'_{chain} = HMAC(K_{message}, "message")$$

    - This message key is unique for each message and is used for encryption or authentication. As a result, even if one message key is compromised, future communications will remain secure.

This mechanism ensures that even if a message key is compromised, future communications remain secure due to the continuous evolution of the chain key.

## 2.5. Implementing Double Ratchet for Diffie-Hellman

The Single Ratchet mechanism provides **forward secrecy** by updating the chain key with each message. However, it lacks **post-compromise security**, meaning it does not safeguard future communications if a key is compromised. The Double Ratchet addresses this limitation by combining **symmetric key ratcheting** with periodic **Diffie-Hellman (DH) key exchanges**, ensuring that even if a key is exposed, future messages remain protected.

The **Double Ratchet** works by periodically performing a **Diffie-Hellman key exchange** between Alice and Bob. After each exchange, they derive a new **root key**, which is then used to generate fresh **chain keys** for both sending and receiving messages.

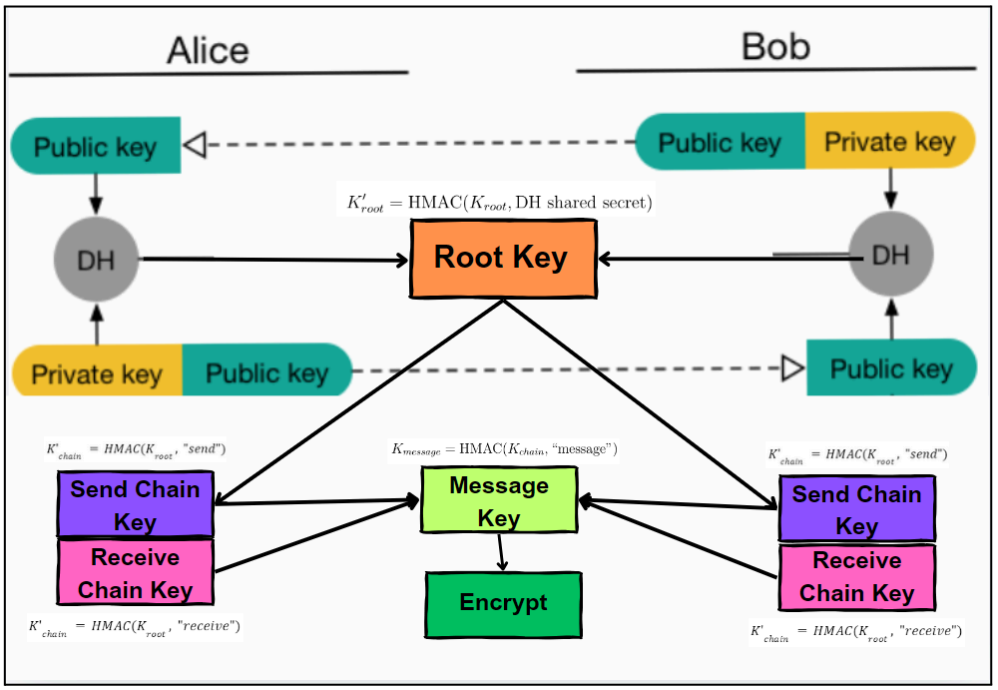



*Figure 6: Double Ratchet mechanism*

To incorporate this mechanism into our project, we created the `DoubleRatchet` class. Below is an outline of how the **Double Ratchet** mechanism works and how it has been implemented in our code:

1. **Initial Setup**:
    - Alice and Bob generate an initial **Diffie-Hellman key pair** (private and public keys) and exchange their public keys.
    - They use the DH shared secret to derive an initial **root key ($K_{root}$)**, which is then used to generate the first set of chain keys for sending and receiving messages.

2. **Diffie-Hellman Ratcheting**:
    - After a certain number of messages or at predefined intervals, Alice and Bob generate new DH key pairs and exchange their new public keys.
    - A new **DH shared secret** is used to derive a new **root key** ($K'_{root}$):

$$K'_{root} = HMAC(K_{root}, DH\ shared\ secret)$$

    - This new root key resets the chain keys, providing fresh encryption keys for future communication.
3. **Symmetric Key Ratchet** (as in the Single Ratchet):
    - After each message is sent or received, the chain key is updated using an HMAC-based PRFG.
    - A new **message key** is derived from the updated chain key:

$$K'_{message} = HMAC(K_{chain}, "message")$$

    - $K_{chain}$ is the **send** or **receive** chain key depending on the action.

By combining Diffie-Hellman key exchanges with symmetric ratcheting, the Double Ratchet ensures that future communications are not only forward secure but also **post-compromise secure**, effectively safeguarding the communication against long-term key exposure.

# 3. Test Results

## Task 1

To verify the correctness of our **Diffie-Hellman key exchange** implementation, we conducted a test over three iterations. In each iteration, both Alice and Bob generate random private keys and compute their respective public keys. Using these public keys, both parties calculate a shared secret key. The test ensures that both Alice and Bob derive the same shared secret key in each iteration, validating the integrity of the Diffie-Hellman process.

```
Parameters
p = 23456
g = 5

--- Test 1 ---
Shared secret key: 5153

--- Test 2 ---
Shared secret key: 23409

--- Test 3 ---
Shared secret key: 16473
```

# Task 2

To test the HMAC for authentication, let's assume that Alice wants to send a message to Bob. She will compute the HMAC of the message using the shared key and this will produce an authentication tag. Bob, upon receiving the message, will compute the HMAC of the received message using the same shared key. He will compare the computed tag with the one sent by Alice. If the tags match, the message is authenticated and verified as untampered. We will have two scenarios, one with **XOR hash** and another with **SHA-256 hash.**

```
--- Testing HMAC with XOR Hash ---
Alice sends the next message: Hello, Bob!
HMAC tag with XOR hash: 220f060605464a2805084b6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a...
Message authenticated and verified as untampered.
------------------------------------------------------------

--- Testing HMAC with SHA-256 Hash ---
Bob sends the next message: Bye, Alice!
HMAC tag with SHA-256 hash:  2a58878e5f1bd5cf9d01f06415762cd3a04aa6c9ce1ebcf6451d40231001fb3f
Message authenticated and verified as untampered.
```

Let's test them separately in different scenarios, using different keys and different messages.

## HMAC with XOR hash

### Test 1

Shared key: 805
Message: “Hello, Bob!”

```
--- Testing HMAC with XOR Hash ---
Alice sends the next message: Hello, Bob!
Shared secret key: 805
HMAC tag with XOR hash: 220f060605464a2805084b6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a...
```

### Test 2

Shared key: 10281
Message: “Hello, Bob!”

```
--- Testing HMAC with XOR Hash ---
Alice sends the next message: Hello, Bob!
Shared secret key: 10281
HMAC tag with XOR hash: 220f060605464a2805084b6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a...
```

### Test 3

Shared key: 805
Message: “Bye, Alice!”

```
--- Testing HMAC with XOR Hash ---
Bob sends the next message: Bye, Alice!
Shared secret key: 805
HMAC tag with XOR hash: 28130f464a2b0603090f4b6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a...
```

### Test 4

Shared key: 10281
Message: “Bye, Alice!”

```
--- Testing HMAC with XOR Hash ---
Bob sends the next message: Bye, Alice!
Shared secret key: 10281
HMAC tag with XOR hash: 28130f464a2b0603090f4b6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a...
```

## HMAC with SHA-256 hash

### Test 1

Shared key: 805
Message: “Hello, Bob!”

```
--- Testing HMAC with SHA-256 Hash ---
Alice sends the next message: Hello, Bob!
Shared secret key: 805
HMAC tag with SHA-256 hash: 9989c0890763784a5f5aedb7c4d8cceba7b314a0092dd51d49a5a71a23facfed
```

### Test 2

Shared key: 10281
Message: “Hello, Bob!”

```
--- Testing HMAC with SHA-256 Hash ---
Alice sends the next message: Hello, Bob!
Shared secret key: 10281
HMAC tag with SHA-256 hash:  df82b69cb65f26c77b1ec37c5387a28734b6bc66616fe3d8f64106f4aa2b8188
```

### Test 3

Shared key: 805
Message: “Bye, Alice!”

```
--- Testing HMAC with SHA-256 Hash ---
Bob sends the next message: Bye, Alice!
Shared secret key: 805
HMAC tag with SHA-256 hash:  21497281ef3f0a0fd75b3e67e51fb8e1451248abef0b12512dd0d5485375dbc2
```

### Test 4

Shared key: 10281
Message: “Bye, Alice!”

```
--- Testing HMAC with SHA-256 Hash ---
Bob sends the next message: Bye, Alice!
Shared secret key: 10281
HMAC tag with SHA-256 hash:  e9f2bc0966b1dec7623e2a8c443f5e0bb4e75cb44d049d4285b2e2c3df952db8
```

# Task 3

To check the operation of HMAC followed by Encryption, we only had to add the encryption process after calculating the HMAC tag.

```
Shared secret key: 22773
Alice sends the next message: Hello, Bob!
HMAC tag with SHA-256 hash: e86fe493334687a8516ef6cf96c7ea8ca6f3975ba785d31559b353843b217a85
Encrypted message: b'\xa2\x07\xa5A\x89\x12N\xd3B\xa5\xa0\x96\x89M\xde\xb2J{,\x96\xe1#i.Z\xd2\x89I...
Nonce: b'\x06\xb9\xb2\x12W\xe6\x15\xddz\x14\xd4\xf1\xfceA\xdd'

Bob receives the Encrypted message and deciphers it: Hello, Bob!e86fe493334687a8516ef6cf96c7ea8...
Original Message: Hello, Bob!
Received HMAC tag: e86fe493334687a8516ef6cf96c7ea8ca6f3975ba785d31559b353843b217a85
Message authenticated and verified as untampered.
```

# Task 4

To test the Single Ratchet mechanism, let's simulate that Alice wants to send several messages to Bob:

```
--- Round 1 ---

Alice sends message 1: Hello, Bob!
HMAC tag: c12d740c54c2cae7b2008e49a57dc881bbf12e4a99f85de5fcab890fb1e4b5dd
Alice Chain Key: 622420a3be39c43352633596843a0a5cd4a76a9381827acb5a2e97859f9ace37

Bob receives message 1
Bob Chain Key: 622420a3be39c43352633596843a0a5cd4a76a9381827acb5a2e97859f9ace37

Decrypted message from Alice: Hello, Bob!c12d740c54c2cae7b2008e49a57dc881bbf12e4a99f85de5fcab890fb1e4b5dd
Original Message: Hello, Bob!
Received HMAC tag: c12d740c54c2cae7b2008e49a57dc881bbf12e4a99f85de5fcab890fb1e4b5dd

Message authenticated and verified as untampered.

--- Round 2 ---

Alice sends message 2: How are you?
HMAC tag: 1c412a689eb7c71f885deea2da45773796327beb72db10da7cda254ee169189c
Alice Chain Key: d471bb7c91d249aba5870b4c9924364c78a69678c93d05b692457de2e45aacca

Bob receives message 2
Bob Chain Key: d471bb7c91d249aba5870b4c9924364c78a69678c93d05b692457de2e45aacca

Decrypted message from Alice: How are you?1c412a689eb7c71f885deea2da45773796327beb72db10da7cda254ee169189c
Original Message: How are you?
Received HMAC tag: 1c412a689eb7c71f885deea2da45773796327beb72db10da7cda254ee169189c

Message authenticated and verified as untampered.
```

```
--- Round 3 ---

Alice sends message 3: Did you receive my message?
HMAC tag: 9c3300351ab77facf72b12bdc2be5656bf11640e6cd1d1111c458eda768994ef
Alice Chain Key: 372d498c6f79c0b2faa8b55ce34b55058ad7c7327e039902134d9e7ffcdf1d79

Bob receives message 3
Bob Chain Key: 372d498c6f79c0b2faa8b55ce34b55058ad7c7327e039902134d9e7ffcdf1d79

Decrypted message from Alice: Did you receive my message?9c3300351ab77facf72b12bdc2be5656bf11640e6cd1d1111c458eda768994ef
Original Message: Did you receive my message?
Received HMAC tag: 9c3300351ab77facf72b12bdc2be5656bf11640e6cd1d1111c458eda768994ef

Message authenticated and verified as untampered.

--- Round 4 ---

Alice sends message 4: See you soon!
HMAC tag: 7510d1ee2cca851a61e92d726a20c0113f65bc3ecfa2dd81a6c857cfa04ffc80
Alice Chain Key: 4a5486f0937f3fe371011460d6bc770cf03399756b2171a4b6519d38b9babc85

Bob receives message 4
Bob Chain Key: 4a5486f0937f3fe371011460d6bc770cf03399756b2171a4b6519d38b9babc85

Decrypted message from Alice: See you soon!7510d1ee2cca851a61e92d726a20c0113f65bc3ecfa2dd81a6c857cfa04ffc80
Original Message: See you soon!
Received HMAC tag: 7510d1ee2cca851a61e92d726a20c0113f65bc3ecfa2dd81a6c857cfa04ffc80

Message authenticated and verified as untampered.
```

As we can see, the chain keys are synchronized in each round, every time Alice sends her message and Bob receives it.

# Task 5

```
--- Round 1 ---

Alice sends message 1: Hello, Bob!
HMAC tag: ad50c9c7414a903504aa2220d7b60db51d0a3c8394979880d230d7cb5354bcba
Alice Send Chain Key: 4c481803b3624a225a191671e92b358f742284f69cacabf45e98ab4708c76b86

Bob receives message 1
Bob Receive Chain Key: 4c481803b3624a225a191671e92b358f742284f69cacabf45e98ab4708c76b86
Decrypted message from Alice: Hello, Bob!ad50c9c7414a903504aa2220d7b60db51d0a3c8394979880d230d7cb5354bcba
Original Message: Hello, Bob!
Received HMAC tag: ad50c9c7414a903504aa2220d7b60db51d0a3c8394979880d230d7cb5354bcba

--- Round 2 ---

Alice sends message 2: How are you?
HMAC tag: 2e933261abedfb0b0337fb04e6acd037d1db4d2b6bc6474d28d0d49c3662c8a9
Alice Send Chain Key: 4e10c185434120cf2d2fb3c1cc3eb0432cb4689f6b11b695ceeb32df2739850d

Bob receives message 2
Bob Receive Chain Key: 4e10c185434120cf2d2fb3c1cc3eb0432cb4689f6b11b695ceeb32df2739850d
Decrypted message from Alice: How are you?2e933261abedfb0b0337fb04e6acd037d1db4d2b6bc6474d28d0d49c3662c8a9
Original Message: How are you?
Received HMAC tag: 2e933261abedfb0b0337fb04e6acd037d1db4d2b6bc6474d28d0d49c3662c8a9

Alice and Bob generate new DH key pairs

Alice:
New root key: 07617a28adcbc5d4daa6a62c8aacc1fa9c686adf118c5b84854f88829f42fd80
New chain key send: d5ae620c29bd1174a238a39266edc77963f2cbc24270ee021b9f3808e81fb445
New chain key receive: 162c4bcc0b560b491785c7a55aeba9cf1b8b67c9f98307f08ca7eac435bbb8b0

Bob:
New root key: 07617a28adcbc5d4daa6a62c8aacc1fa9c686adf118c5b84854f88829f42fd80
New chain key send: 162c4bcc0b560b491785c7a55aeba9cf1b8b67c9f98307f08ca7eac435bbb8b0
New chain key receive: d5ae620c29bd1174a238a39266edc77963f2cbc24270ee021b9f3808e81fb445
```

```
--- Round 3 ---

Bob sends message 3: Hello, Alice!
HMAC tag: 7655d9f6c8215f003d8321af6d138833080fe64080ce8d97b770c2ae1b5f67b7
Bob Send Chain Key: c0d95e75a8d9435c731b20b33532d9dc3dc7bb5f438b85b072e39e1a5312e034

Alice receives message 3
Alice Receive Chain Key: c0d95e75a8d9435c731b20b33532d9dc3dc7bb5f438b85b072e39e1a5312e034
Decrypted message from Bob: Hello, Alice!7655d9f6c8215f003d8321af6d138833080fe64080ce8d97b770c2ae1b5f67b7
Original Message: Hello, Alice!
Received HMAC tag: 7655d9f6c8215f003d8321af6d138833080fe64080ce8d97b770c2ae1b5f67b7

--- Round 4 ---

Bob sends message 4: Fine, I am doing the A2
HMAC tag: adec4f850649ce5faa10e7f2349d8cf94740ef2975da95e65706e4127d513555
Bob Send Chain Key: 3bbac6f1cf61b6ce2459817f2915c36768e71b7ac3fb04a2325b24967c9eca5d

Alice receives message 4
Alice Receive Chain Key: 3bbac6f1cf61b6ce2459817f2915c36768e71b7ac3fb04a2325b24967c9eca5d
Decrypted message from Bob: Fine, I am doing the A2adec4f850649ce5faa10e7f2349d8cf94740ef2975da95e65706e4127d513555
Original Message: Fine, I am doing the A2
Received HMAC tag: adec4f850649ce5faa10e7f2349d8cf94740ef2975da95e65706e4127d513555
```

In this case, Alice's send chain key is synchronized with Bob's receive chain key and vice versa. Every two rounds, new keys are generated.

# 4. Discussion

## Task 1

The results showed that for each iteration, Alice and Bob successfully derived identical shared secret keys, confirming the correct implementation of the Diffie-Hellman key exchange protocol.

Since the shared secret key relies on both the exchanged public values and the private values, which remain undisclosed, it becomes highly challenging for an attacker to derive the secret key. This is because solving for the secret involves computing the discrete logarithm, a process that is computationally difficult without access to the private keys. Additionally, when large prime numbers are used in the modulus operation, it becomes exceedingly difficult for an attacker to break the encryption. The use of large primes in Diffie-Hellman significantly complicates the computation of the discrete logarithm, making it a key component in secure cipher suites.

## Task 2

HMAC ensures that the message has not been modified during transmission and that it comes from a trusted source. The receiver can verify both the integrity and authenticity of the message, since only parties who know the secret key can generate or validate the HMAC.

HMAC with both XOR and SHA-256 hashing algorithms has been proven effective in authenticating and verifying messages. In Alice's case, the message "Hello, Bob!" was authenticated with HMAC and an XOR hash, while the message "Bye, Alice!" sent by Bob was authenticated with HMAC and SHA-256. In both scenarios, the authentication mechanism worked correctly, ensuring the integrity and authenticity of the messages.

We will now proceed to analyze both hashing algorithms separately, evaluating their characteristics, advantages and limitations when used in conjunction with HMAC.

### HMAC with XOR hash

- **Identical HMAC for the same message and different key (Test 1 and Test 2)**

  Despite using two different keys (805 and 10281), the HMAC generated for the message "Hello, Bob!" is identical, which could suggest an insufficiency in the entropy contributed by the key in the current XOR hashing scheme.

- **Different HMAC for a different message (Test 3 and Test 4):**

  In both cases, changing the message to "Bye, Alice!" generated a different HMAC than "Hello, Bob!", which is expected and demonstrates that the hash function responds to changes in the message content.

These results show that although XOR can generate different HMACs for different messages, the algorithm's ability to generate significantly different results by changing the key is limited, which represents a vulnerability in terms of cryptographic security. This makes XOR unsuitable for cryptographic hash functions, which rely on properties like diffusion (small changes in input producing significant changes in output) and collision resistance (difficulty in finding two inputs that produce the same output).

### HMAC with SHA-256 hash

When performing the 4 tests using SHA-256 as the hash function, it was observed that the HMACs generated for each combination of key and message were different. This result is a clear indicator of the improvements provided by SHA-256 compared to the use of XOR as the hash function.

SHA-256 provides collision resistance, guarantees the avalanche effect, and ensures that each generated HMAC is unique, even with small changes to the key or message. This makes SHA-256 the ideal choice for real-world cryptographic and security applications.

## Task 3

When Alice not only authenticates the message using HMAC, but also encrypts both the message and the HMAC tag, an extra layer of security is added to the communication.

- **Confidentiality**: By encrypting the message, Alice ensures that only Bob, who has the decryption key, can read the message's content. This protects confidential information from being intercepted by third parties, even if they manage to capture the encrypted message.

- **HMAC Protection**: By encrypting both the message and the HMAC tag, Alice prevents an attacker from accessing the HMAC in its original form. This ensures that an attacker cannot attempt to replicate the HMAC or perform brute force attacks or analysis on the HMAC to find weaknesses in the system.

It should be noted that although this approach is very secure, it increases computational complexity. Both encryption and HMAC calculation require additional processing, which may be a factor in systems with limited resources or requiring high efficiency.

## Task 4

In the implementation of the Single Ratchet mechanism, the results obtained show that the system guarantees both the integrity and authenticity of the transmitted messages. Verifying each message using the HMAC tag ensures that the messages have not been altered during transmission, fulfilling one of the main objectives of communications security. This verification process is carried out successfully in each round, indicating that both parties, Alice and Bob, are in sync with respect to the generated Chain Keys.

One of the most notable features of the Single Ratchet is its ability to renew the keys after each message sent, which ensures that the system is resistant to key compromise attacks. This property, known as **Forward Secrecy**, guarantees that even if a current key is compromised, future messages remain secure, as new keys are generated continuously and automatically. Furthermore, the system implements Post-compromise Security, which ensures that an attacker cannot decrypt future messages even after gaining access to the keys from a previous round.

Using a new key in each message also mitigates the risks of replay attacks, since an intercepted message cannot be successfully resent, as the encryption key will have changed. Also, generating a new HMAC tag in each round guarantees the authenticity of the sender, by ensuring that only the person who possesses the chain key can generate a valid HMAC.

Overall, the results of the implementation validate the robustness of the Single Ratchet mechanism, which provides both confidentiality and authenticity of messages, even in scenarios where the security of a key is compromised.

## Task 5

Single Ratchet provides strong protection by allowing session keys to be updated progressively after each message sent. In this mechanism, the chain key is used to encrypt messages and is updated every time a new message is sent. However, a notable limitation of Single Ratchet is its vulnerability to post-compromise attacks. If an attacker manages to obtain the key for a compromised session, he can decrypt all previous messages, since all keys depend on the same initial session key.

To cover this limitation, Double Ratchet has been implemented. This mechanism combines two layers of ratchets: the symmetric key ratchet and the Diffie-Hellman (DH) key exchange ratchet, thus providing greater protection of both the confidentiality and integrity of transmitted messages.

A notable aspect of the Double Ratchet mechanism is its ability to continuously update keys. Every time a message exchange takes place, Alice and Bob generate new DH keys, which strengthens security and provides Forward Secrecy. This means that even if an attacker manages to compromise a session key, they will not be able to decrypt past messages, as each message uses a unique key that is dynamically generated.

Throughout the communication, the process of generating new DH keys and ratcheting ensures that each subsequent message is encrypted with a new set of keys, mitigating risks such as replay attack.. For example, when Bob sends "Hello, Alice!" in the third round, the use of a new chain key ensures that the communication remains secure, even in a potentially compromised environment.

The robustness of the Double Ratchet Mechanism is enhanced by its implementation of Post-compromise Security, which ensures that an attacker who manages to compromise the key for one round will not be able to access the keys for future rounds.

In conclusion, the findings of this implementation of the Double Ratchet Mechanism underline its effectiveness in providing a comprehensive security framework for communications, allowing Alice and Bob to exchange messages securely and reliably, even under adverse conditions. This ability to maintain confidentiality and authenticity highlights its relevance in modern messaging systems, where protecting user privacy is paramount.

# 5. Conclusion

In this project, we have implemented and tested several key mechanisms in secure communication protocols, such as the Diffie-Hellman, Single Ratchet, and Double Ratchet algorithms, as well as different methods for generating message authentication codes (HMAC). Throughout development, both forward secrecy and post-compromise security have been checked, ensuring that communications are secure even if some keys are compromised.

The use of XOR and SHA-256-based HMAC was evaluated, proving that while XOR can offer a lightweight alternative, it lacks the cryptographic robustness offered by SHA-256.

The results obtained validated that Single Ratchet is effective in maintaining forward secrecy, while Double Ratchet adds an additional layer of security by periodically exchanging Diffie-Hellman keys, ensuring that future communications remain secure after a compromise.

In summary, the project demonstrated the fundamental principles of cryptographic security in communication environments, providing a solid foundation for the implementation of secure messaging systems.

As a future improvement, optimizations in the efficiency of the algorithm could be explored, especially in resource-constrained scenarios. It would also be valuable to implement an Elliptic Curve Diffie-Hellman (ECDH) approach to improve the efficiency of the key exchange.

# References

[1] Postal, Caitlin. (2024). How Diffie-Hellman Key Exchange Provides Encrypted Communications. UpGuard.com. https://www.upguard.com/blog/diffie-hellman

[2] Volkov, Oleksandr. (2023). How HMAC Works, Step-by-Step Explanation with Examples. Medium.com. https://medium.com/@short_sparrow/how-hmac-works-step-by-step-explanation-with-examples-f4aff5efb40e

[3] Buchanan, William J (2024). *Hashing*. Asecuritysite.com. https://asecuritysite.com/mac/hmac

[4] Ch0pin, (2021). The Signal Protocol and the Double Ratchet algorithm. Medium.com. https://valsamaras.medium.com/the-signal-protocol-and-the-double-ratchet-algorithm-e3d01d1e403f

[5] ] T. Perrin and M. Marlinspike. (2016). The Double Ratchet Algorithm. Signal.org. https://signal.org/docs/specifications/doubleratchet/

[6] DAT510. Chapter 10 - Other Public-Key Cryptosystems.
[7] DAT510. Chapter 11 - Cryptographic Hash Functions
[8] DAT510. Chapter 12 - Message Authentication Codes