Security and Vulnerability in Networks Assignment 2

Abstract

This project presents the implementation of a secure communication framework that utilizes various cryptographic techniques to ensure confidentiality, integrity, and authenticity in message exchanges. The framework incorporates the Diffie-Hellman key exchange to establish a shared secret between parties, enabling secure communication without transmitting sensitive information directly. HMAC (Hash-based Message Authentication Code) is employed to verify message integrity, ensuring that messages remain untampered during transmission. AES (Advanced Encryption Standard) is utilized for encrypting messages, providing robust confidentiality through its efficient encryption algorithms. The project also explores the Single and Double Ratchet mechanisms to facilitate continuous key updates, enhancing security. The results demonstrated that these components work effectively together, with successful key exchanges and secure message transmissions validated through HMAC verification. A future improvement is implementing digital signatures for enhanced authentication. Overall, this project highlights the effectiveness of combining multiple cryptographic techniques to create a resilient communication system that can withstand modern security challenges.

1. Introduction

In today's digital age, secure communication is essential for protecting sensitive information exchanged between individuals and organizations. As cyber threats continue to evolve, cryptographic techniques play a critical role in safeguarding data against unauthorized access and tampering. This project focuses on implementing cryptographic primitives that establish a secure communication framework through the integration of the Diffie-Hellman key exchange, HMAC (Hash-based Message Authentication Code), and Advanced Encryption Standard (AES) encryption.

The Diffie-Hellman key exchange allows two parties to establish a shared secret over an insecure channel, enabling subsequent encrypted communication without transmitting the secret itself. This method relies on the difficulty of solving discrete logarithm problems, which adds a layer of security. However, while the Diffie-Hellman exchange ensures confidentiality, it does not provide authentication, making it susceptible to man-in-the-middle attacks . To counter this vulnerability, HMAC is utilized to ensure message integrity and authenticity. By appending an HMAC tag to each message, the receiving party can verify that the message has not been altered during transmission .

Furthermore, AES encryption is employed to ensure the confidentiality of the messages exchanged. AES is widely recognized for its efficiency and security, utilizing block ciphers to encrypt plaintext into ciphertext, which can only be decrypted with the appropriate key . The implementation of AES in CBC (Cipher Block Chaining) mode enhances security by preventing identical plaintext blocks from producing the same ciphertext.

Additionally, this project explores the Single and Double Ratchet mechanisms to provide forward secrecy and key management. The Single Ratchet allows for the continuous evolution of keys, ensuring that even if one key is compromised, previous keys remain secure. The Double Ratchet extends this concept by integrating Diffie-Hellman key exchanges alongside symmetric ratcheting, allowing for both secure messaging and key updates .

Overall, this project aims to demonstrate the practical implementation of these cryptographic techniques, ensuring secure communication while addressing the challenges posed by modern cyber threats.

2. Design and implementation

2.1. Implementing Diffie-Hellman

The **Diffie-Hellman (DH) key exchange** is a cryptographic protocol that allows two parties to securely establish a shared secret over an insecure communication channel. It is the foundation for many cryptographic systems, providing a mechanism to agree on a symmetric key that can be used for encryption without having to directly transmit the key itself.

In Diffie-Hellman, each party selects a private key and uses a shared public base (**g**) and a large prime number (**p**), both of which are publicly known. The private key is combined with these public parameters to generate a public key. The parties exchange their public keys, and then each party combines the received public key with their private key to compute the shared secret.

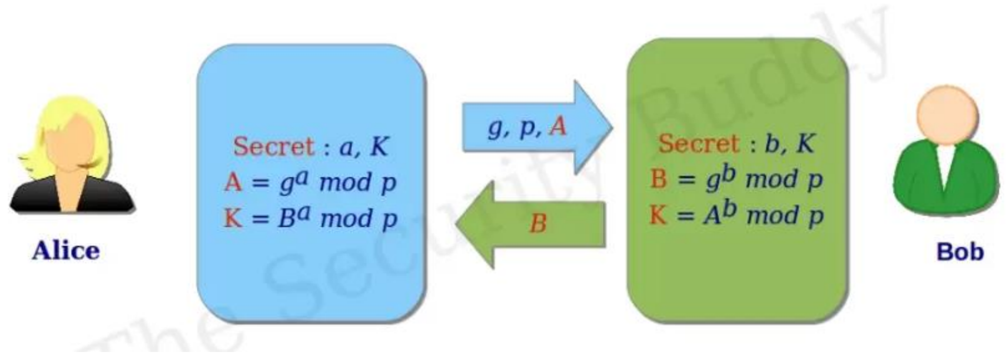


Image taken from:
https://www.thesecuritybuddy.com/encryption/how-does-diffie-hellman-key-exchange-protocol-work/

For **Task I**, the Diffie-Hellman key exchange serves as the mechanism for Alice and Bob to securely establish an initial **shared secret (K)**. In the implementation, Alice and Bob generate **private keys (a and b)** and compute their corresponding public keys by raising the shared base g to the power of their private keys, modulo p. They then exchange these public keys over the insecure channel. Upon receiving each other's public keys, both Alice and Bob can compute the shared secret by raising the received public key to the power of their respective private key, again modulo p.

The shared secret key computed by both parties is the same because of the mathematical property of the Diffie-Hellman protocol:

$$S = (g^b)^a \mod p = (g^a)^b \mod p$$

- Implementation

For the implementation of Diffie-Hellman, the program ‘**task1.py**’ has been developed:

```
# Step 1: Agree on public parameters
p = number
g = number

# Step 2: Alice and Bob generate private keys
a = number (1, p-1)
b = number (1, p-1)

# Step 3: Compute public keys
A = g^a mod p
B = g^b mod p

# Step 4: Exchange public keys and compute shared secret
shared_secret_Alice = B^a mod p
shared_secret_Bob = A^b mod p

# Step 5: Verify both parties have the same shared secret
if shared_secret_Alice == shared_secret_Bob -> “Shared secret successfully computed”
else -> "Error: Shared secrets do not match"
```

Pseudocode of task1.py (Diffie-Hellman)

The implementation begins with the selection of public parameters, specifically a large prime number (p) and a primitive root (g). These parameters are crucial as they form the basis for generating the public keys.

Then, both Alice and Bob generate their private keys. This is done by selecting random integers between 1 and p-1, ensuring that these keys remain confidential and are not disclosed to anyone else. Afterward, Alice and Bob compute their public keys using the formula **g^(private_key) mod p**. These public keys are then exchanged openly, allowing both parties to use them in subsequent calculations without fear of compromising their private keys.

In the next phase of the protocol, Alice and Bob compute the shared secret. Alice calculates this secret by taking Bob's public key and raising it to the power of her private key, modulo p. Similarly, Bob performs the calculation using Alice's public key and his private key. Due to the properties of modular arithmetic, both calculations yield the same shared secret, which can be verified through an assertion in the code.

Finally, the implementation outputs the public keys and the computed shared secret. This output illustrates that both parties have successfully derived the same shared secret, which is essential for subsequent symmetric encryption of their communications.

2.2. Implementing HMAC for Authentication

HMAC (Hash-based Message Authentication Code) combines a hash function with a secret key (Diffie-Hellman shared secret), ensuring that the message hasn't been tampered with and that the sender is authenticated. The primary use of HMAC is to generate a message authentication code (MAC), which is a fixed-length value derived from both the message and a secret key. This MAC can then be used to verify the integrity and authenticity of the message during transmission.

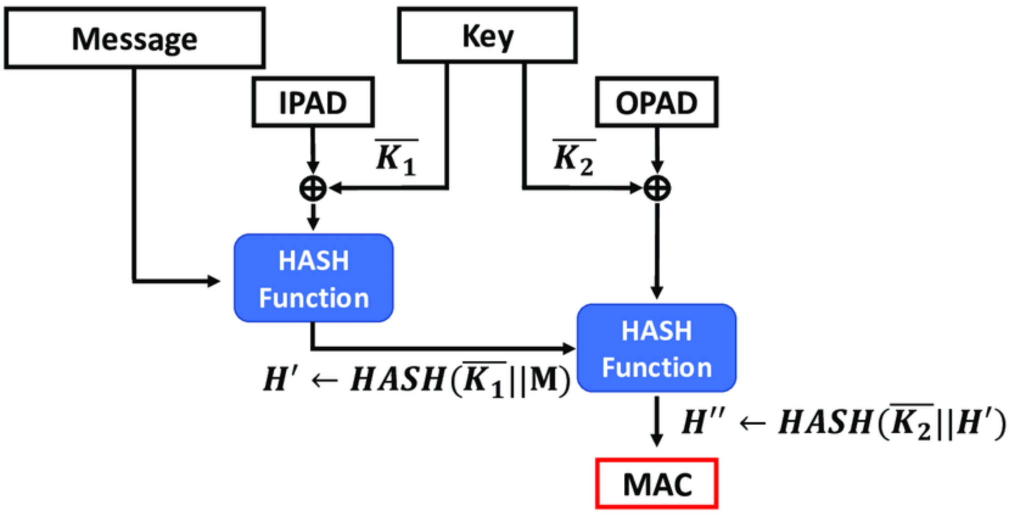

Message
Key
IPAD
OPAD
$\overline{K_1}$
$\overline{K_2}$
HASH Function
HASH Function
$H' \leftarrow HASH(\overline{K_1}||M)$
$H'' \leftarrow HASH(\overline{K_2}||H')$
MAC

Image taken from:
https://www.researchgate.net/figure/Hash-Message-Authentication-Code-HMAC-process-332-Overview-of-SHA-256-SHA-256_fig2_346634579

The HMAC algorithm involves two main steps. First, it pads the secret key to ensure that it is the same length as the block size of the hash function. If the key is longer than the block size, it is hashed to reduce its length. Next, two specific values, called the **inner pad (ipad)** and the **outer pad (opad)**, are generated by XORing the padded key with two different constants. The message is then combined with these pads, and the hash function is applied twice, first with the inner pad and message, and then with the outer pad and the result of the first hash.

$$\mathrm{HMAC}(K, m) = \mathrm{hash}\left(\left(K' \oplus \mathrm{opad}\right) \parallel \mathrm{hash}\left(\left(K' \oplus \mathrm{ipad}\right) \parallel m\right)\right)$$

In **Task II**, the HMAC method will be first implemented using an XOR-based hash function, which simplifies the HMAC process while still offering a basic level of security.

- Implementation

For the implementation of XOR-Based HMAC using Diffie-Hellman shared secret key, the program '**task2.py**' has been developed:

The implementation starts with the **Diffie-Hellman (DH) key exchange, the computation of public keys and the shared secret key** (like the 'task1.py'). This shared secret will later be used as the key in the HMAC process and that is why it is then converted to a byte string.

The '**xor_hash**' function is a simplified hash function based on the XOR operation. It works by iterating through the input data (message or key) and performing a bitwise XOR operation on each byte. This operation takes each byte from the input and XORs it with a corresponding byte from an initially zeroed-out byte array (hash_value), which is used to accumulate the hash result.

```
Function xor_hash(data, block_size):

  # Initialize a byte array of a block size, filled with zeros
  hash_value = byte_array(block_size)

  # For each byte in the input data:
  For i from 0 to length of data:
    # XOR the current byte of data with the corresponding byte in hash_value
    hash_value[i % block_size] = hash_value[i % block_size] XOR data[i]

  # Return the resulting byte array as a hash
  Return hash_value
```

Pseudocode of xor_hash function

Finally, we have the '**hmac_xor'** function, which follows several steps. First the secret key (the shared secret from Diffie-Hellman) is padded to the block size (64 bytes by default). If the key is longer than the block size, it is hashed using the XOR-based hash function. Then, two constants, **ipad** and **opad**, are created by XORing the padded key with 0x36 and 0x5c respectively.

The **inner hash** is computed by hashing the concatenation of ipad and the message, and the **outer hash** is then computed by hashing the concatenation of opad and the result of the inner hash, both using the 'xor_hash' function. The result is the HMAC tag generated based on the initial message and secret key.

```
Function hmac_xor(key, message, block_size):

  # Step 1: If the key is longer than the block size, hash it using XOR-based hash
  If length of key > block_size:
    key = xor_hash(key, block_size)

  # Step 2: Pad the key to the block size with zero bytes
  key = pad key to block_size using zero bytes

  # Step 3: Create opad and ipad using XOR
  For each byte in key:
    opad_byte = key_byte XOR 0x5c
    ipad_byte = key_byte XOR 0x36

  # Step 4: Perform the HMAC operation
  Inner hash: xor_hash((K' ⊕ ipad) || message)
  Outer hash: hash((K' ⊕ opad) || inner_hash)

  # Return the HMAC tag
  Return Outer hash
```

Pseudocode of hmac_xor function

In this way, by calling this last function with the shared secret key obtained from the Diffie-Hellman process and the original message, we can obtain the HMAC tag.

With the aim of improving the analysis and security of the communication system, an alternative to the XOR-based hash function has been considered: the **SHA-256** function (in the file '**task2.2.py**').

SHA-256 works by breaking the input data into blocks, processing each block through several rounds of mathematical transformations, and producing a final 256-bit hash. It is a secure and efficient cryptographic hash function that plays a vital role in many security protocols and applications. Its ability to generate a fixed-length output from input data of any size, combined with its strong resistance to pre-image and collision attacks, makes it a cornerstone of modern cryptography.

For its implementation, the 'hmac_xor' function has been replaced by the '**hmac_sha256**' function. This function adjusts the key and defines the constants ipad and opad similarly to the previous function. However, to carry out the hash operations, it uses the "sha256" function from the hashlib library.

```
Function hmac_sha256(key, message):
    block_size = 64  # Block size for SHA-256

    # Step 1: If the key is longer than the block size, hash it using SHA-256
    If length of key > block_size:
        key = sha256_hash(key)

    # Step 2: Pad the key to block size with zero bytes
    key = pad key to block_size using zero bytes

    # Step 3: Create opad and ipad using XOR
    For each byte in key:
        opad_byte = key_byte XOR 0x5c
        ipad_byte = key_byte XOR 0x36

    # Step 4: Perform the HMAC operation
    Inner_hash: sha256_hash((K' ⊕ ipad) || message)
    Outer_hash: sha256_hash((K' ⊕ opad) || inner_hash)

    # Return the HMAC tag
    Return Outer_hash
```

Pseudocode of hmac_sha256 function

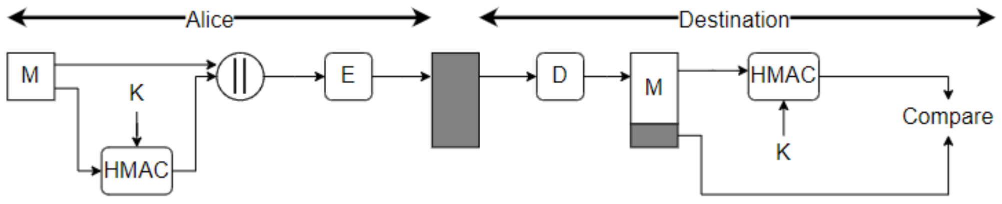
2.3. Applying Encryption-Decryption

For the encryption and decryption of the message during communication, the **AES algorithm in CBC mode** has been used, as it was the most efficient in the previous assignment.

The combination of HMAC followed by AES (Advanced Encryption Standard) encryption ensures that messages exchanged between Alice and Bob are both authenticated and confidential.

The process begins with HMAC, which provides data integrity and authenticity. Alice computes the HMAC using the shared secret derived from the Diffie-Hellman key exchange and the message she wants to send. Once the HMAC is generated, Alice concatenates the original message with the HMAC tag. This combined data is then encrypted using AES in CBC mode, which ensures that the message remains confidential during transmission.

When Bob receives the ciphertext, he decrypts it using the same AES key and IV. After decryption, Bob extracts the original message and the HMAC tag. He then computes a new HMAC for the received message using the shared secret and compares it to the received HMAC tag. If they match, it confirms the message's authenticity and integrity, assuring Bob that the message has not been altered in transit. This combined approach effectively secures the communication between Alice and Bob.



- Implementation

For the implementation of the encryption-decryption process, the program '**task3.py**' has been developed:

The programme begins by establishing a shared secret between Alice and Bob using the Diffie-Hellman (DH) key exchange protocol (as we have seen before).

Once the shared secret is established, it is converted to bytes to serve as the key for HMAC. The function hmac_sha256 is defined to compute the HMAC using the SHA-256 hashing algorithm (described in the task II). The result is an HMAC tag that uniquely ties the message to the shared secret, ensuring data integrity and authenticity.

Next, the implementation uses **AES encryption** to secure the message. A random 128-bit AES key and an Initialization Vector (IV) are generated. Before encryption, the original message is concatenated with the HMAC tag, creating a combined message that includes both the content and its integrity check.

```
# Step 1: Define the message and shared secret
message = "Hello, Bob!"
shared_secret = Derived_DH_shared_secret

# Step 2: Compute HMAC using the shared secret and the message
hmac_tag1 = hmac_sha256(shared_secret, message)

# Step 3: Generate a random 128-bit AES key and Initialization Vector (IV)
aes_key = generate_random_bytes(16 bytes)
iv = generate_random_bytes(16 bytes)

# Step 4: Concatenate the message and HMAC tag
concatenated_message = message + hmac_tag1

# Step 5: Encrypt the concatenated message using AES-CBC mode
ciphertext = aes_encrypt(concatenated_message, aes_key, iv)
```

Pseudocode of encryption process

The aes_encrypt function pads the plaintext to fit the AES block size and then encrypts the combined message using the AES key and IV in CBC mode. The resulting ciphertext is encoded in Base64 format for safe transmission.

```
Function aes_encrypt(plaintext, key, iv):

    # Step 1: Create an AES cipher object with the provided key and IV, in CBC mode
    cipher = initialize AES cipher with key, CBC mode, and iv

    # Step 2: Encode and pad the plaintext to match the block size
    padded_plaintext = pad(plaintext, AES block size)

    # Step 3: Encrypt the padded plaintext
    ciphertext = cipher.encrypt(padded_plaintext)

    # Step 4: Encode the ciphertext in base64 format and return as string
    Return base64_encode(ciphertext)
```

```
Function aes_decrypt(ciphertext, key, iv):

    # Step 1: Create an AES cipher object with the provided key and IV, in CBC mode
    cipher = initialize AES cipher with key, CBC mode, and iv

    # Step 2: Decode the base64-encoded ciphertext
    decoded_ciphertext = base64_decode(ciphertext)

    # Step 3: Decrypt the decoded ciphertext
    decrypted_data = cipher.decrypt(decoded_ciphertext)

    # Step 4: Unpad the decrypted data to remove padding added during encryption
    plaintext = unpad(decrypted_data, AES block size)

    # Step 5: Return the plaintext as a decoded string
    Return plaintext
```

Pseudocode of aes_encrypt and aes_decrypt functions

Upon receiving the ciphertext, Bob decrypts it using the same AES key and IV. The decrypted plaintext contains both the original message and the HMAC tag. Bob extracts these components and computes a new HMAC for the original message using the shared secret. This new HMAC is compared to the received HMAC tag to verify the message's authenticity. If they match, it confirms that the message has not been altered during transmission and verifies that it was indeed sent by Alice.

```
# Step 1: Decrypt the ciphertext using AES-CBC mode
decrypted_plaintext = aes_decrypt(ciphertext, aes_key, iv)

# Step 2: Extract the original message and the received HMAC tag
original_message = decrypted_plaintext[:-64]
received_hmac_tag = decrypted_plaintext[-64:]  # Extract the last 64 characters (which is the received HMAC tag)

# Step 3: Compute the HMAC of the original message using the shared secret
hmac_tag2 = hmac_sha256(shared_secret, original_message)

# Step 4: Verify the integrity of the message
If received_hmac_tag == hmac_tag2 -> "Message is authentic"
Else -> "Message authentication failed"
```

Pseudocode of decryption process

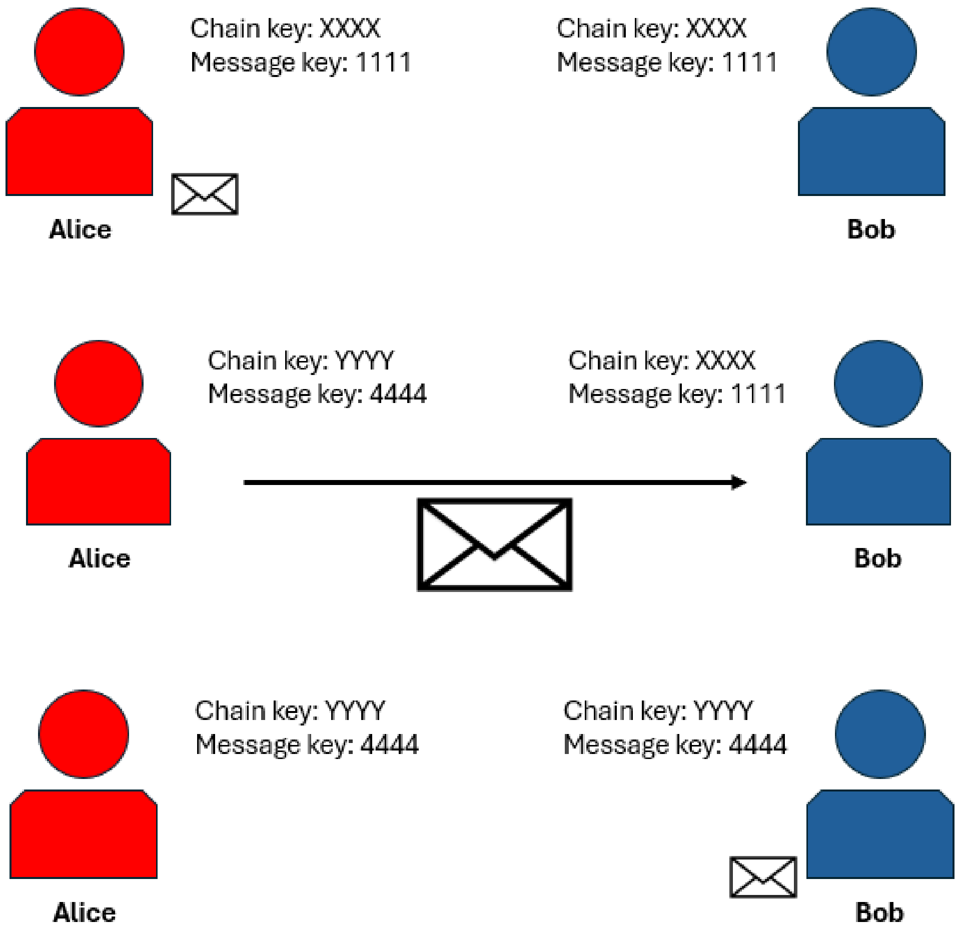
2.4. Implementing Single Ratchet

The **Single Ratchet** mechanism revolves around the idea of updating the **chain key** after each message exchange. Every time a message is sent or received, the chain key is ratcheted forward using a cryptographic hash function (HMAC), to derive new keys. This ensures that even if the current key is compromised, it cannot be used to decrypt previous messages (forward secrecy), nor can it be used to predict future keys (post-compromise security).

The process begins with a **Diffie-Hellman key exchange**, where Alice and Bob establish a shared secret. This shared secret serves as the initial **chain key** for both parties. The chain key is used to derive a new **message key** for encrypting messages, which is created by applying HMAC again to the chain key, ensuring that each message key is distinct.

After deriving the message key, the message is encrypted using **AES in CBC mode** with a randomly generated **Initialization Vector (IV)** for each message. This further increases security by ensuring that even identical plaintexts produce different ciphertexts when encrypted. On the receiver's side, the chain key is ratcheted forward, keeping both Alice and Bob in sync so they can decrypt the message using the same message key. This synchronized ratcheting of keys allows seamless communication, with both parties remaining securely synchronized.

As already said, after each message is exchanged, the **chain key is updated or "ratcheted"** forward using the **HMAC-SHA256** function. This ratcheting process ensures that every time a new message is sent or received, the chain key changes, and a new encryption key is derived.



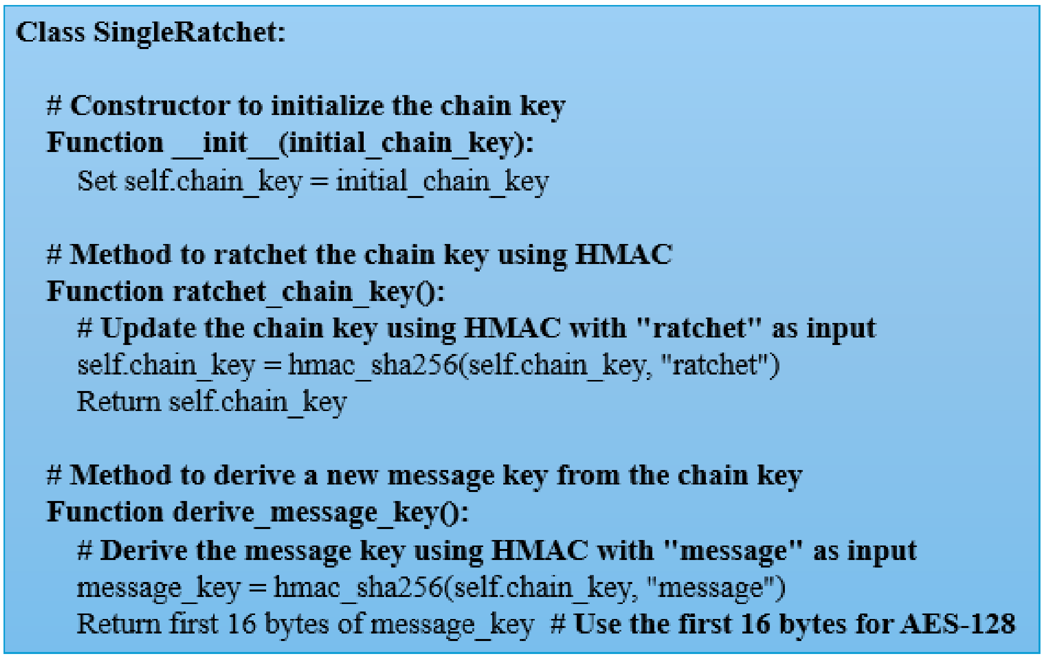
- Implementation

For the implementation of the Single Rachet mechanism, the program '**task4.py**' has been developed:

The first step in the code is to establish a shared secret using the **Diffie-Hellman key exchange**, in the same way as in the previous cases.

The functions **hmac_sha256**, **aes_encrypt**, and **aes_decrypt** remain the same.

The **Single Ratchet** is implemented as a class that handles the management of the chain key and the derivation of the message key. The **ratchet_chain_key** method updates the chain key using HMAC-SHA256, while the **derive_message_key** method derives a new message key for encrypting each message. The message key is derived from the chain key, ensuring that every message is encrypted with a unique key.



```
Class SingleRatchet:

    # Constructor to initialize the chain key
    Function __init__(initial_chain_key):
        Set self.chain_key = initial_chain_key

    # Method to ratchet the chain key using HMAC
    Function ratchet_chain_key():
        # Update the chain key using HMAC with "ratchet" as input
        self.chain_key = hmac_sha256(self.chain_key, "ratchet")
        Return self.chain_key

    # Method to derive a new message key from the chain key
    Function derive_message_key():
        # Derive the message key using HMAC with "message" as input
        message_key = hmac_sha256(self.chain_key, "message")
        Return first 16 bytes of message_key  # Use the first 16 bytes for AES-128
```

Pseudocode of SingleRatchet class

The Single Rachet instances of Bob and Alice are initialized with the shared secret of Diffie-Hellman.

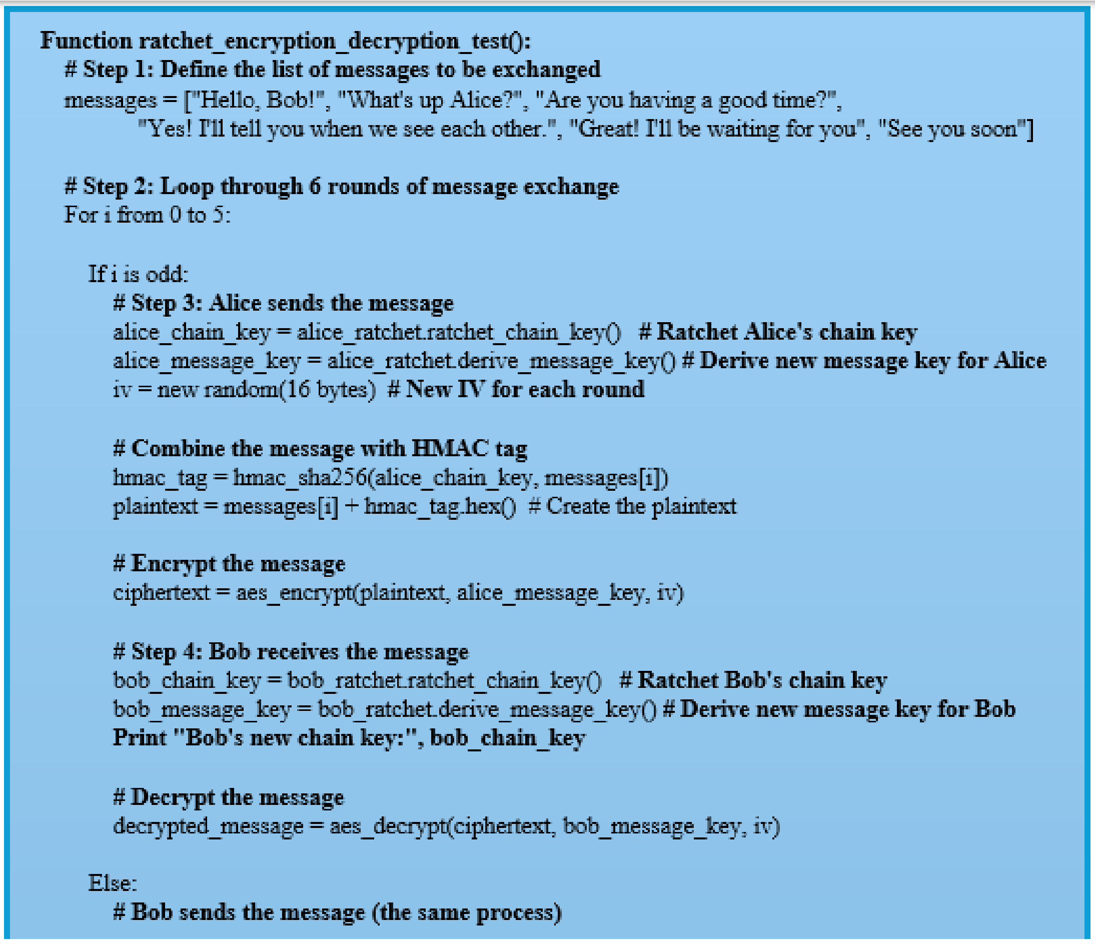
The **ratchet_encryption_decryption_test** function simulates a secure message exchange between Alice and Bob using the **Single Ratchet mechanism**. The purpose of this test is to ensure that the keys for encryption are updated (or "ratcheted") after each message, keeping Alice and Bob's keys synchronized.

The test runs over multiple rounds to simulate multiple messages being exchanged securely.

In each round, Alice starts by **ratcheting her chain key**, using **HMAC-SHA256**. From this updated chain key, Alice derives a **message key** which is used to encrypt the current plaintext (message + hmac tag) using **AES encryption** in **CBC mode**.

Once Alice has encrypted the message, Bob needs to decrypt it. To do this, Bob first ratchets his chain key in the same way that Alice did, ensuring their keys remain in sync. From his updated chain key, Bob derives a new **message key** that allows him to decrypt the ciphertext using AES in CBC mode.

The same is done when Bob sends a message to Alice.



```
Function ratchet_encryption_decryption_test():
  # Step 1: Define the list of messages to be exchanged
  messages = ["Hello, Bob!", "What's up Alice?", "Are you having a good time?",
        "Yes! I'll tell you when we see each other.", "Great! I'll be waiting for you", "See you soon"]

  # Step 2: Loop through 6 rounds of message exchange
  For i from 0 to 5:

    If i is odd:
      # Step 3: Alice sends the message
      alice_chain_key = alice_ratchet.ratchet_chain_key()  # Ratchet Alice's chain key
      alice_message_key = alice_ratchet.derive_message_key() # Derive new message key for Alice
      iv = new random(16 bytes)  # New IV for each round

      # Combine the message with HMAC tag
      hmac_tag = hmac_sha256(alice_chain_key, messages[i])
      plaintext = messages[i] + hmac_tag.hex()  # Create the plaintext

      # Encrypt the message
      ciphertext = aes_encrypt(plaintext, alice_message_key, iv)

      # Step 4: Bob receives the message
      bob_chain_key = bob_ratchet.ratchet_chain_key()  # Ratchet Bob's chain key
      bob_message_key = bob_ratchet.derive_message_key() # Derive new message key for Bob
      Print "Bob's new chain key:", bob_chain_key

      # Decrypt the message
      decrypted_message = aes_decrypt(ciphertext, bob_message_key, iv)

    Else:
      # Bob sends the message (the same process)
```

Afterward, the message is separated from the received HMAC tag. The test also includes checks to ensure the ratchet mechanism is functioning correctly. First, it checks that **the received HMAC tag is correct**, verifying the authenticity of the message. Second, it verifies that the **decrypted message matches the original message**. And third, it ensures that Alice and Bob’s **chain keys remain synchronized** after each round of encryption and decryption. If both checks pass, it confirms that the Single Ratchet mechanism is correctly updating the encryption keys, and that both parties are using the same keys to encrypt and decrypt messages.

```
# Step 5: Extract the original message and HMAC
original_message = decrypted_message[:-64]
received_hmac_tag = decrypted_message[-64:]  # Extract received HMAC tag

# Step 6: Compute HMAC for authenticity check
If i is odd:
   hmac_tag2 = hmac_sha256(bob_chain_key, original_message)
Else:
   hmac_tag2 = hmac_sha256(alice_chain_key, original_message)

# Step 7: Verify the authenticity of the message
if received_hmac_tag == hmac_tag2.hex() -> "The message is authentic"

# Step 8: Ensure decrypted message matches original
if original_message == messages[i] -> "Decrypted message match the original"

# Step 9: Ensure Alice and Bob's chain keys are synchronized
if alice_chain_key == bob_chain_key -> "Alice and Bob's chain keys are synchronized"
```

Pseudocode of ratchet_encryption_decryption_test function

To run the test, we will simply call the function once the Diffie-Hellman shared secret has been obtained and used to initialize the chain keys for Alice and Bob.

```
# Step 1: Obtain the shared secret key
shared_secret = Derived_DH_shared_secret

# Step 2: Initialize the Single Ratchet mechanism with the shared secret
alice_ratchet = SingleRatchet(shared_secret)
bob_ratchet = SingleRatchet(shared_secret)

# Step 3: Run the test for Single Ratchet encryption and decryption
ratchet_encryption_decryption_test()
```

Pseudocode of task4.py procedure

2.5. Implementing Double Rachet

The **Double Ratchet algorithm** is implemented to provide secure, evolving encryption in message exchanges between Alice and Bob. The algorithm combines **two ratcheting mechanisms**: the **Diffie-Hellman (DH) ratchet** and the **symmetric-key ratchet**. This combination ensures both **forward secrecy** (past messages remain secure if current keys are compromised) and **post-compromise security** (future messages are protected even if current keys are exposed).

The **Diffie-Hellman ratchet** is used to periodically update the key agreement between Alice and Bob by exchanging new DH key pairs. This process generates a new shared secret, which then feeds into a Key Derivation Function (KDF) to create new **sending and receiving chain keys**. These chain keys are used by the symmetric ratchet to derive individual **message keys** for each message.

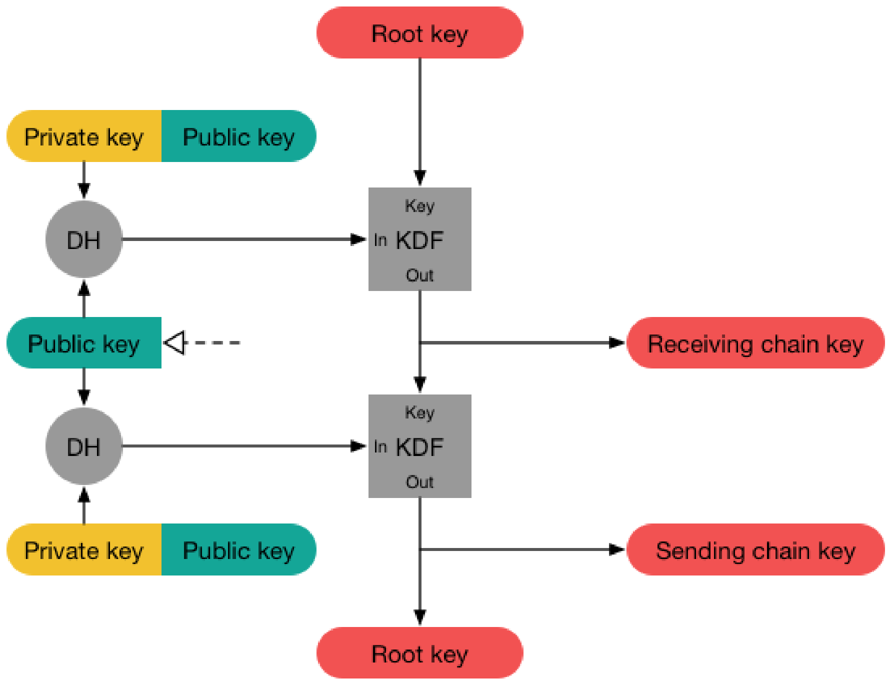


Image taken from: https://signal.org/docs/specifications/doubleratchet/

The **symmetric-key ratchet** is responsible for updating the encryption keys after each individual message. Once the new chain keys are established via the DH ratchet, the symmetric ratchet applies HMAC-SHA256 to the current chain key to derive the next message key. This ensures that every message is encrypted with a unique message key, even if no new DH key pairs are exchanged.

For each new message sent, Alice or Bob ratchets the corresponding chain key to derive a new message key, which is used to encrypt the message using **AES-CBC** encryption. This message key is discarded after use, ensuring it cannot be reused or compromised.

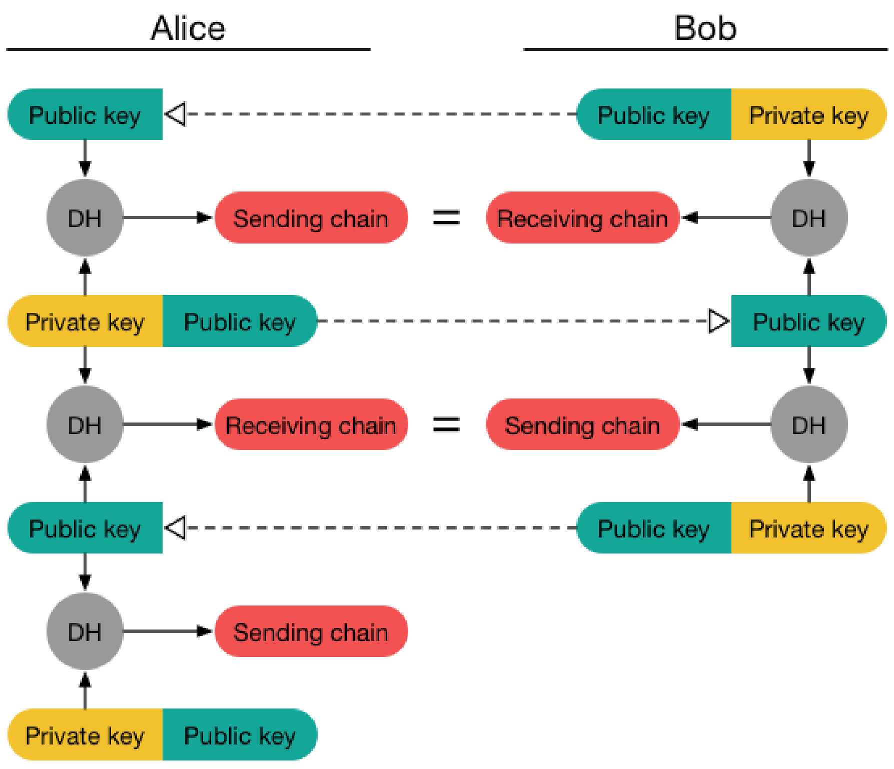


Image taken from:
https://signal.org/docs/specifications/doubleratchet/

- Implementation

For the implementation of the Double Rachet communication, the program '**task5.py**' has been developed:

In addition to the functions already defined earlier (dh_generate_key_pair, dh_compute_shared_secret, hmac_sha256, and aes encryption and decryption), the '**DoubleRatchet**' class has been implemented.

This class is the core of the Double Ratchet Algorithm implemented in Task V. It ensures that every message exchanged between Alice and Bob uses a fresh cryptographic key. Its **constructor** sets up the initial state of the ratchet mechanism. When a DoubleRatchet object is created, Alice and Bob use their respective Diffie-Hellman key pairs to establish an initial shared secret. The shared secret is then used to derive the **root key**, from which two chain keys are initialized: one for sending messages and one for receiving them:

```
Class DoubleRatchet:

  # Constructor to initialize the Double Ratchet mechanism
  Function __init__(p, g, initial_dh_private, initial_dh_public, ppublic_key):
    Set self.dh_private = initial_dh_private
    Set self.dh_public = initial_dh_public
    Set self.p = p
    Set self.g = g
    Set self.ppublic_key = ppublic_key # Public key of the other person

    # Step 1: Compute the shared secret
    self.shared_secret = dh_compute_shared_secret(self.ppublic_key, self.dh_private, self.p)

    # Step 2: Initialize the root key
    self.root_key = hmac_sha256(shared_secret, "root")

    # Step 3: Initialize chain keys for sending and receiving
    self.chain_key_send = hmac_sha256(self.root_key, "send")
    self.chain_key_reci = hmac_sha256(self.root_key, "receive")
```

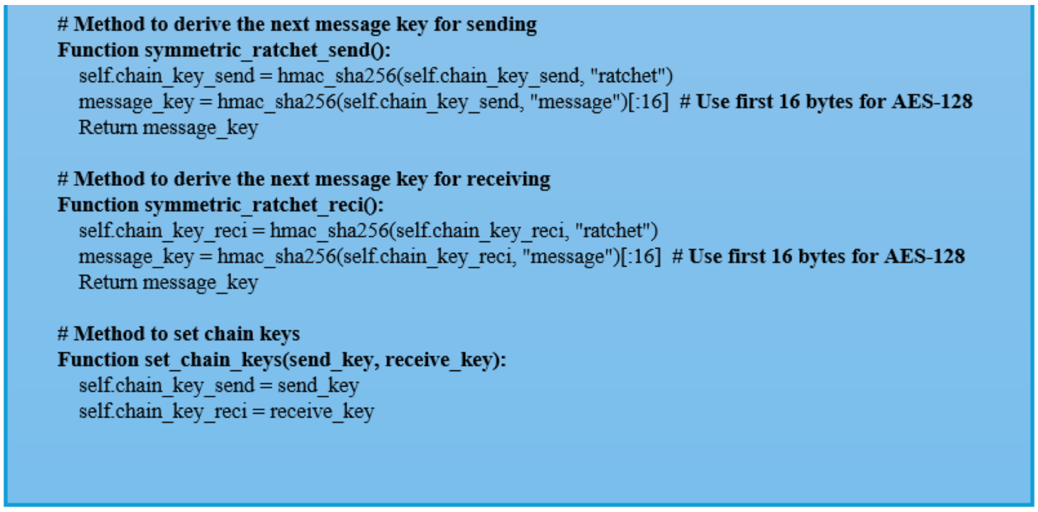
The method **ratchet_chain_key** handles the key update process by generating a new DH key pair, computing a new shared secret with the other party's public key, and deriving a new root key. This ensures that the system refreshes cryptographic material after every few messages, protecting future communications if a current key is compromised.

After the shared secret is computed, new sending and receiving chain keys are derived from the updated root key using HMAC-SHA256:

```
  # Method to ratchet the chain keys using HMAC
  Function ratchet_chain_key(private_key, public_key, ppublic_key):

    # Update the public keys and compute a new shared secret
    self.ppublic_key = ppublic_key
    self.dh_private = private_key
    self.dh_public = public_key

    self.shared_secret = dh_compute_shared_secret(self.ppublic_key, self.dh_private, self.p)

    # Update the root key
    self.root_key = hmac_sha256(self.root_key, shared_secret)

    # Update chain keys
    self.chain_key_send = hmac_sha256(self.root_key, "send")
    self.chain_key_reci = hmac_sha256(self.root_key, "receive")
```

The methods **symmetric_ratchet_send** and **symmetric_ratchet_reci** handle the process for deriving new message keys from the respective chain keys.

Finally, the **set_chain_keys** function in the DoubleRatchet class is responsible for manually setting the sending and receiving chain keys:



```
# Method to derive the next message key for sending
Function symmetric_ratchet_send():
    self.chain_key_send = hmac_sha256(self.chain_key_send, "ratchet")
    message_key = hmac_sha256(self.chain_key_send, "message")[:16]  # Use first 16 bytes for AES-128
    Return message_key

# Method to derive the next message key for receiving
Function symmetric_ratchet_reci():
    self.chain_key_reci = hmac_sha256(self.chain_key_reci, "ratchet")
    message_key = hmac_sha256(self.chain_key_reci, "message")[:16]  # Use first 16 bytes for AES-128
    Return message_key

# Method to set chain keys
Function set_chain_keys(send_key, receive_key):
    self.chain_key_send = send_key
    self.chain_key_reci = receive_key
```

Pseudocode of DoubleRatchet class

In addition, the **double_ratchet_test** function simulates a series of secure message exchanges between Alice and Bob using the Double Ratchet mechanism. The test function validates how the Double Ratchet evolves keys over multiple rounds, ensuring message confidentiality, integrity, and synchronization between Alice and Bob.

The function starts by defining a list of messages that Alice and Bob will exchange. Each message in this list represents a round of communication between Alice and Bob. The test will run for 6 rounds, with Alice and Bob alternating as the sender and receiver.

For each message, the function checks if it is Alice's or Bob's turn to send the message. If Alice is the sender, she performs the symmetric ratchet to generate a new message key and uses it to encrypt the message along with an HMAC tag. The same process is followed when Bob is the sender. The message is encrypted using AES in CBC mode with a new Initialization Vector (IV) for each round to ensure unique encryption.

Upon receiving the message, the receiver (either Alice or Bob) performs the symmetric ratchet to generate the correct decryption key based on their chain key. The function then decrypts the message using the derived message key:

```
Function double_ratchet_test():
    # Step 1: Define the list of messages to be exchanged
    messages = ["Hello, Bob!", "What's up Alice?", "Are you having a good time?",
              "Yes! I'll tell you when we see each other.", "Great! I'll be waiting for you", "See you soon"]

    # Step 2: Loop through 6 rounds of message exchange
    For i from 0 to 5:

        If i is odd:
            # Alice's turn to send the message

            # Step 3: Alice ratchets the chain key and derives a new message key
            alice_message_key = alice_ratchet.symmetric_ratchet_send()
            iv = generate_random_bytes(16)
            hmac_tag = hmac_sha256(alice_ratchet.chain_key_send, messages[i])

            plaintext = messages[i] + hmac_tag.hex()  # Combine message with HMAC

            # Step 4: Alice encrypts the message
            ciphertext = aes_encrypt(plaintext, alice_message_key, iv)

            # Step 5: Bob ratchets his chain key to stay in sync and derives a new message key
            bob_message_key = bob_ratchet.symmetric_ratchet_reci()

            # Step 6: Bob decrypts the message
            decrypted_message = aes_decrypt(ciphertext, bob_message_key, iv)

        Else:
            # Bob's turn to send the message (the same process)
```

After decryption, the message is separated from the HMAC tag. This last one is checked to ensure the message was not tampered with during transmission:

```
# Step 7: Extract the original message and HMAC
original_message = decrypted_message[:-64]
received_hmac_tag = decrypted_message[-64:]  # Extract received HMAC tag

# Step 8: Compute HMAC for authenticity check
If i is odd:
    hmac_tag2 = hmac_sha256(bob_ratchet.chain_key_reci, original_message)
Else:
    hmac_tag2 = hmac_sha256(alice_ratchet.chain_key_reci, original_message)

# Step 9: Verify the authenticity of the message
if received_hmac_tag == hmac_tag2.hex() -> "The message is authentic"

# Step 10: Ensure decrypted message matches original
if original_message == messages[i] -> "Decrypted message matches the original"
```

Every 3 rounds, the test performs a Diffie-Hellman ratchet. This involves Alice and Bob generating new Diffie-Hellman key pairs, exchanging public keys, and computing a new shared secret. This shared secret is then used to derive new chain keys for sending and receiving messages:

```
    # Step 11: Every 3 rounds, perform a DH ratchet (key exchange)
    If (i + 1) % 3 == 0:

        # Step 12: Alice and Bob generate new DH key pairs
        alice_private, alice_public = dh_generate_key_pair(p, g)
        bob_private, bob_public = dh_generate_key_pair(p, g)

        alice_ratchet.ratchet_chain_key(alice_private, alice_public, bob_public)
        bob_ratchet.ratchet_chain_key(bob_private, bob_public, alice_public)

        # Step 13: Set chain keys in Bob's instance
        bob_ratchet.set_chain_keys(bob_ratchet.chain_key_reci, bob_ratchet.chain_key_send)
```

Pseudocode of double_ratchet_test function

Once the functions are defined, the code **begins setting up the Double Ratchet mechanism.** Alice and Bob generate their Diffie-Hellman key pairs using a large prime p and a primitive root g. Alice shares her public key with Bob, and vice versa, enabling each party to compute a shared secret.

Then, they both **initialize their DoubleRatchet classes** using their own private and public keys and each other's public keys. Also, Bob's chain keys are synchronized with Alice's using the set_chain_keys function, ensuring both parties start from the same cryptographic state.

Finally, the **test function is called** to verify the functionality of the algorithm:

```
# Step 1: Initialize the Double Ratchet for Alice and Bob
alice_ratchet = DoubleRatchet(p, g, alice_private, alice_public, bob_public)
bob_ratchet = DoubleRatchet(p, g, bob_private, bob_public, alice_public)

# Step 2: Synchronize chain keys
bob_ratchet.set_chain_keys(alice_ratchet.chain_key_reci, alice_ratchet.chain_key_send)

# Step 3: Run the test for Double Ratchet encryption and decryption
double_ratchet_test()
```

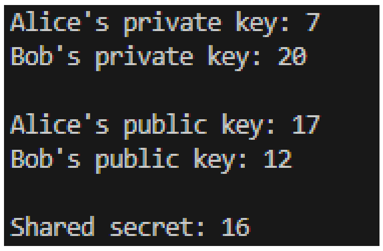
Pseudocode of task5.py procedure

3. Test Results

Now, we are going to run the programs to test the implementation. This process will allow us to evaluate whether the functionalities work as expected and provide us with information for further analysis.

- Diffie-Hellman Implementation (task1.py):

Using the Diffie-Hellman key exchange algorithm, the program has calculated the following values based on the public parameters p (a large prime number) and g (a primitive root modulo p):

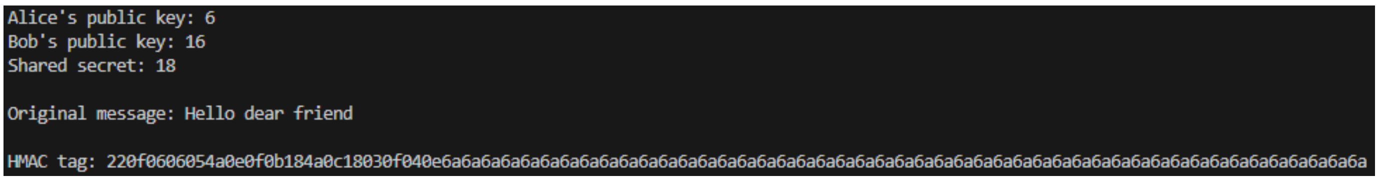


```
Alice's private key: 7
Bob's private key: 20

Alice's public key: 17
Bob's public key: 12

Shared secret: 16
```

- HMAC Implementation:

1. <u>HMAC using XOR-based hash (task2.py):</u>

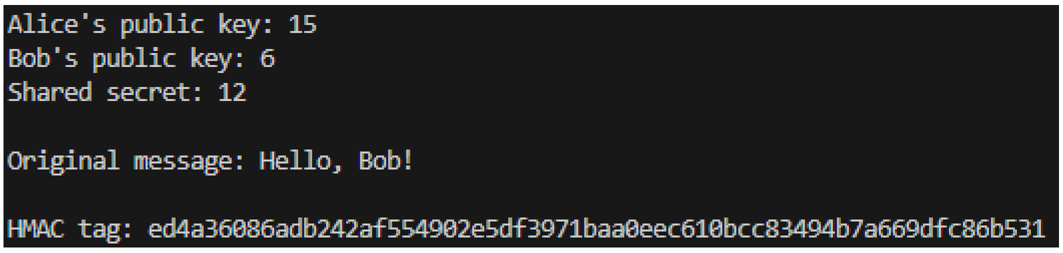
Using the HMAC with XOR hash program, the following values were obtained:



```
Alice's public key: 6
Bob's public key: 16
Shared secret: 18

Original message: Hello dear friend

HMAC tag: 220f0606054a0e0f0b184a0c18030f040e6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a6a
```

2. <u>HMAC using SHA-256 (task2.2.py):</u>

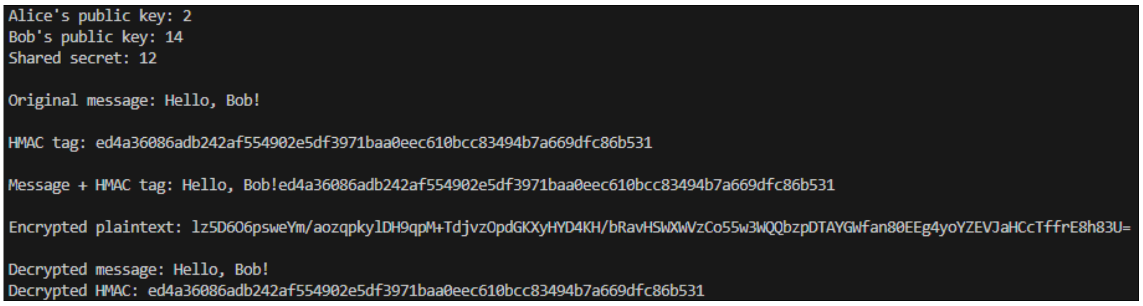
On the other hand, while using the HMAC with SHA-256 hash program, these are the results:



```
Alice's public key: 15
Bob's public key: 6
Shared secret: 12

Original message: Hello, Bob!

HMAC tag: ed4a36086adb242af554902e5df3971baa0eec610bcc83494b7a669dfc86b531
```

- Encryption-Decryption Implementation (task3.py):

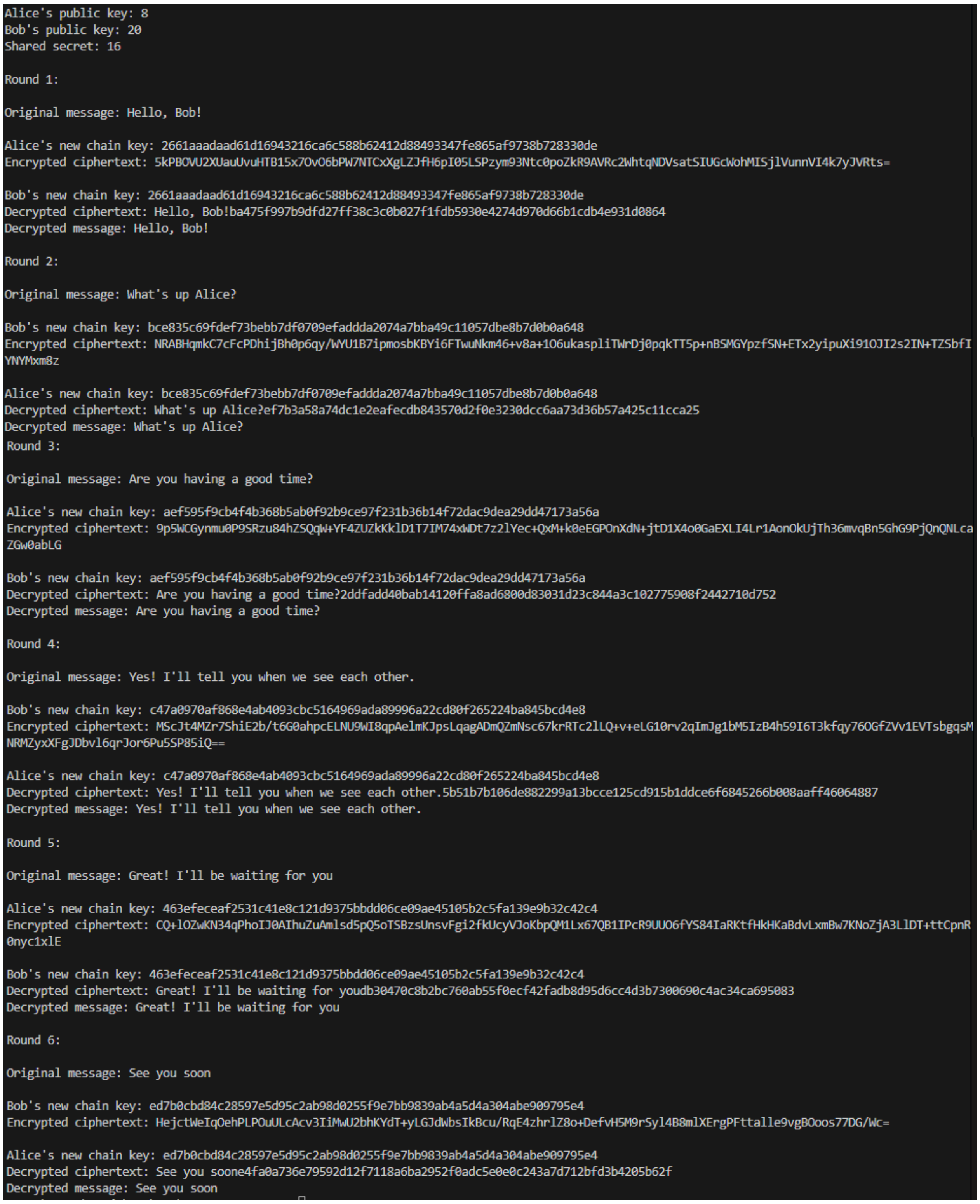
When the encryption-decryption program is executed, we see how the original message is combined with the HMAC tag to be encrypted (**plaintext**). Then, the receiver decrypts the plaintext and separates the message from the HMAC (which is used to authenticate the message).



```
Alice's public key: 2
Bob's public key: 14
Shared secret: 12

Original message: Hello, Bob!

HMAC tag: ed4a36086adb242af554902e5df3971baa0eec610bcc83494b7a669dfc86b531

Message + HMAC tag: Hello, Bob!ed4a36086adb242af554902e5df3971baa0eec610bcc83494b7a669dfc86b531

Encrypted plaintext: 1z5D6O6psweYm/aozqpkylDH9qpM+TdjvzOpdGKXyHYD4KH/bRavHSWXWVzCo55w3WQQbzpDTAYGWfan80EEg4yoYZEVJaHCcTffrE8h83U=

Decrypted message: Hello, Bob!
Decrypted HMAC: ed4a36086adb242af554902e5df3971baa0eec610bcc83494b7a669dfc86b531
```

- Single-Ratchet Implementation (task4.py):

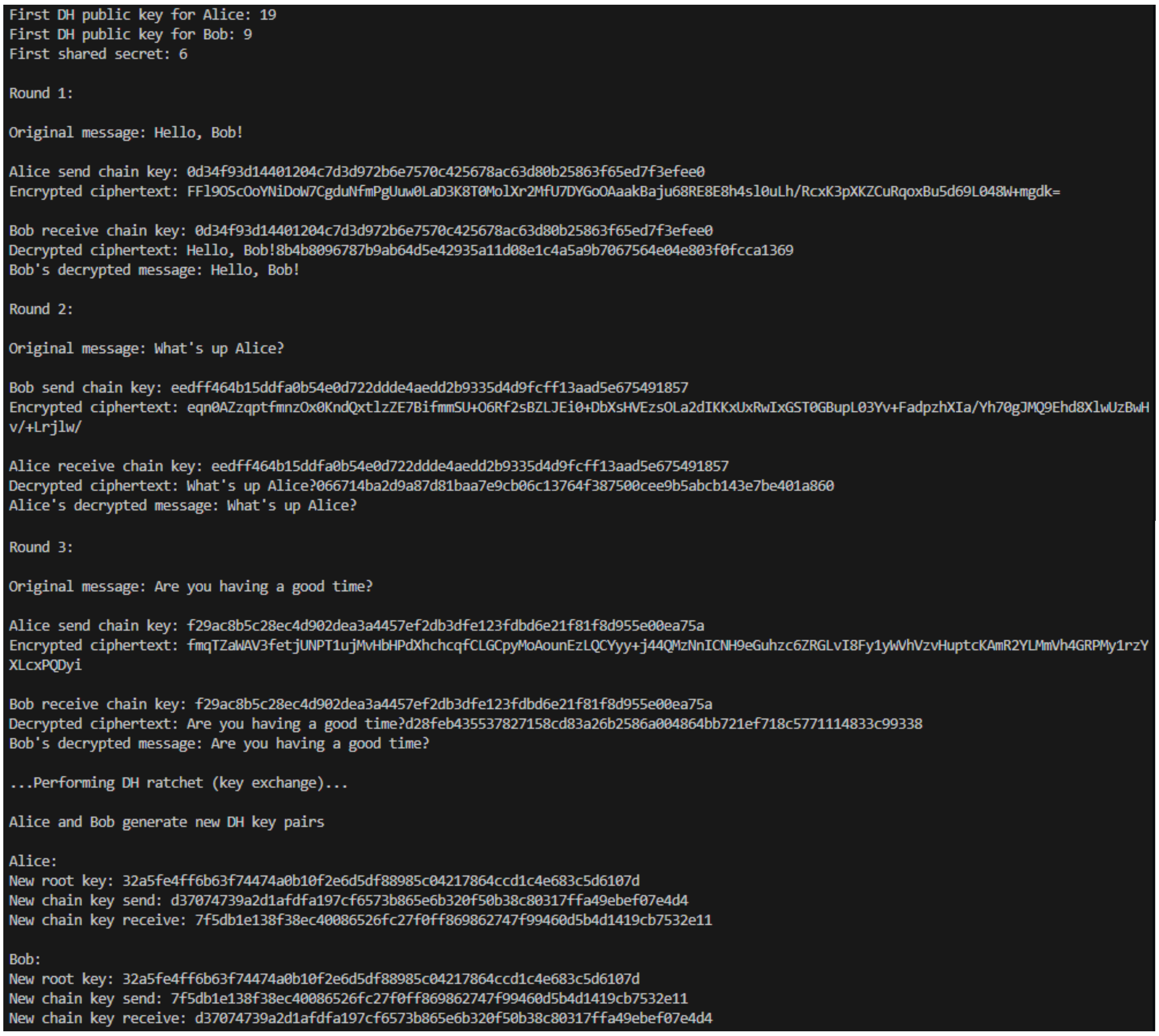
When the single ratchet program is executed, it will carry out several rounds, and in each one, we will see the exchange of a message from Alice to Bob or vice versa. On the screen, the new chain key derived from the sender is displayed, the encrypted ciphertext (message + HMAC), the new chain key derived from the receiver when the message is received (it must be the same as the sender's), how the ciphertext is decrypted, and finally how the message is separated from the HMAC.



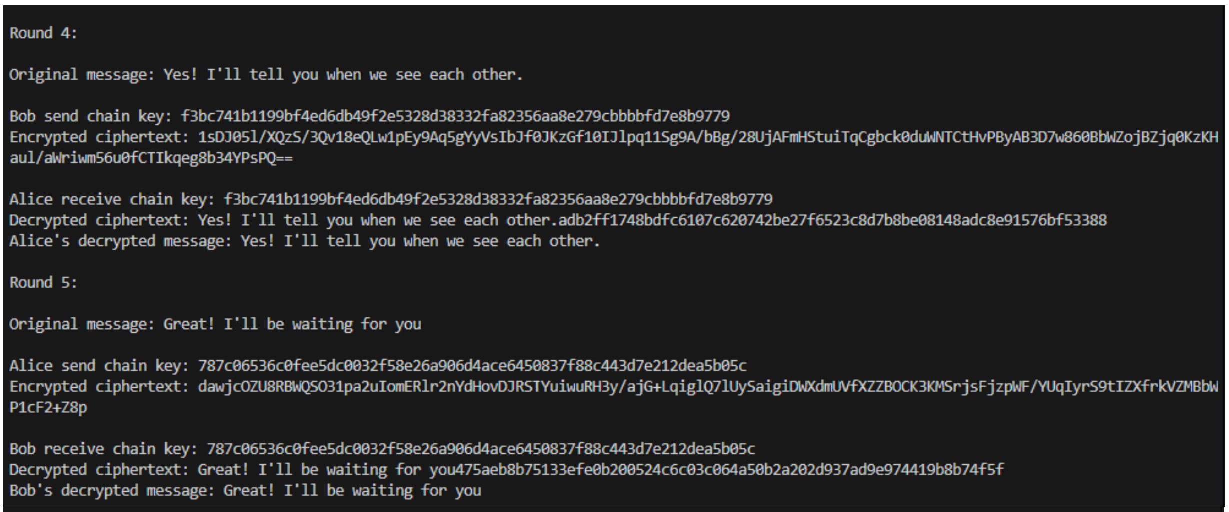
```
Alice's public key: 8
Bob's public key: 20
Shared secret: 16

Round 1:

Original message: Hello, Bob!

Alice's new chain key: 2661aaadaad61d16943216ca6c588b62412d88493347fe865af9738b728330de
Encrypted ciphertext: 5kPBOVU2XUauUvuHTB15x7OvO6bPW7NTCxXgLZJfH6pI05LSPzym93Ntc0poZkR9AVRc2WhtqNDVsatSIUGcWohMISjlVunnVI4k7yJVRts=

Bob's new chain key: 2661aaadaad61d16943216ca6c588b62412d88493347fe865af9738b728330de
Decrypted ciphertext: Hello, Bob!ba475f997b9dfd27ff38c3c0b027f1fdb5930e4274d970d66b1cdb4e931d0864
Decrypted message: Hello, Bob!

Round 2:

Original message: What's up Alice?

Bob's new chain key: bce835c69fdef73bebb7df0709efaddda2074a7bba49c11057dbe8b7d0b0a648
Encrypted ciphertext: NRABHqmkC7cFcPDhijBh0p6qy/WYU1B7ipmosbKBYi6FTwuNkm46+v8a+1O6ukaspliTWrDj0pqkTT5p+nBSMGYpzfSN+ETx2yipuXi91OJI2s2IN+TZSbfI
YNYMxm8z

Alice's new chain key: bce835c69fdef73bebb7df0709efaddda2074a7bba49c11057dbe8b7d0b0a648
Decrypted ciphertext: What's up Alice?ef7b3a58a74dc1e2eafecdb843570d2f0e3230dcc6aa73d36b57a425c11cca25
Decrypted message: What's up Alice?
Round 3:

Original message: Are you having a good time?

Alice's new chain key: aef595f9cb4f4b368b5ab0f92b9ce97f231b36b14f72dac9dea29dd47173a56a
Encrypted ciphertext: 9p5WCGynmu0P9SRzu84hZSQqW+YF4ZUZkKklD1T7IM74xWDt7z2lYec+QxM+k0eEGPOnXdN+jtD1X4o0GaEXLI4Lr1AonOkUjTh36mvqBn5GhG9PjQnQNLca
ZGw0abLG

Bob's new chain key: aef595f9cb4f4b368b5ab0f92b9ce97f231b36b14f72dac9dea29dd47173a56a
Decrypted ciphertext: Are you having a good time?2ddfadd40bab14120ffa8ad6800d83031d23c844a3c102775908f2442710d752
Decrypted message: Are you having a good time?

Round 4:

Original message: Yes! I'll tell you when we see each other.

Bob's new chain key: c47a0970af868e4ab4093cbc5164969ada89996a22cd80f265224ba845bcd4e8
Encrypted ciphertext: MScJt4MZr7ShiE2b/t6G0ahpcELNU9WI8qpAelmKJpsLqagADmQZmNsc67krRTc2lLQ+v+eLG10rv2qImJg1bM5IzB4h59I6T3kfqy76OGfZVv1EVTsbgqsM
NRMZyxXFgJDbvl6qrJor6Pu5SP85iQ==

Alice's new chain key: c47a0970af868e4ab4093cbc5164969ada89996a22cd80f265224ba845bcd4e8
Decrypted ciphertext: Yes! I'll tell you when we see each other.5b51b7b106de882299a13bcce125cd915b1ddce6f6845266b008aaff46064887
Decrypted message: Yes! I'll tell you when we see each other.

Round 5:

Original message: Great! I'll be waiting for you

Alice's new chain key: 463efeceaf2531c41e8c121d9375bbdd06ce09ae45105b2c5fa139e9b32c42c4
Encrypted ciphertext: CQ+lOZwKN34qPhoIJ0AIhuZuAmlsd5pQ5oTSBzsUnsvFgi2fkUcyVJoKbpQM1Lx67QB1IPcR9UUO6fYS84IaRKtfHkHKaBdvLxmBw7KNoZjA3LlDT+ttCpnR
0nyc1xlE

Bob's new chain key: 463efeceaf2531c41e8c121d9375bbdd06ce09ae45105b2c5fa139e9b32c42c4
Decrypted ciphertext: Great! I'll be waiting for youdb30470c8b2bc760ab55f0ecf42fadb8d95d6cc4d3b7300690c4ac34ca695083
Decrypted message: Great! I'll be waiting for you

Round 6:

Original message: See you soon

Bob's new chain key: ed7b0cbd84c28597e5d95c2ab98d0255f9e7bb9839ab4a5d4a304abe909795e4
Encrypted ciphertext: HejctWeIqOehPLPOuULcAcv3IiMwU2bhKYdT+yLGJdWbsIkBcu/RqE4zhrlZ8o+DefvH5M9rSyl4B8mlXErgPFttalle9vgBOoos77DG/Wc=

Alice's new chain key: ed7b0cbd84c28597e5d95c2ab98d0255f9e7bb9839ab4a5d4a304abe909795e4
Decrypted ciphertext: See you soone4fa0a736e79592d12f7118a6ba2952f0adc5e0e0c243a7d712bfd3b4205b62f
Decrypted message: See you soon
```

- Double-Ratchet Implementation (task5.py):

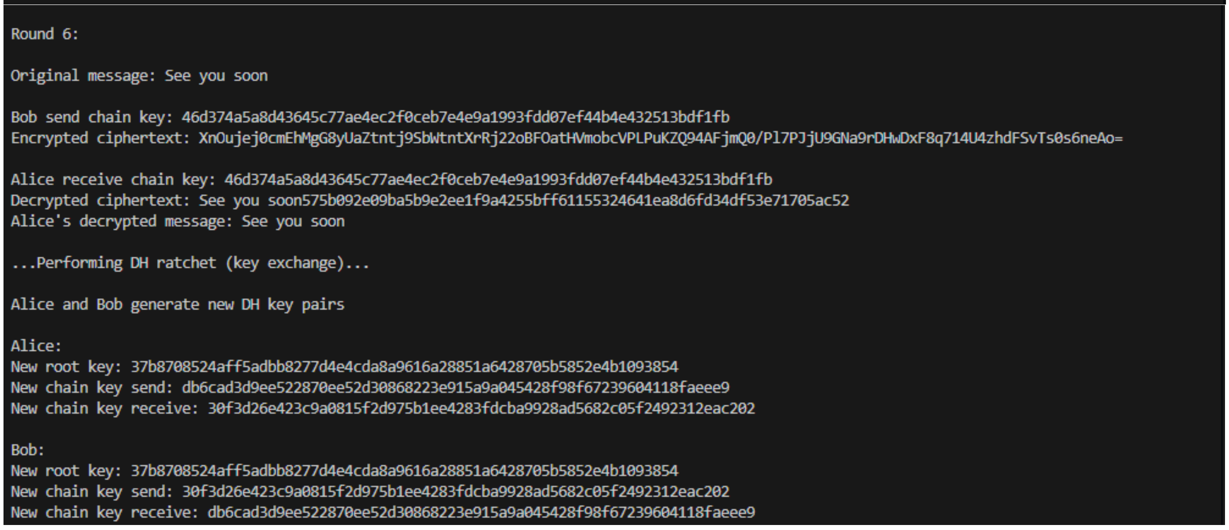
Lastly, the double ratchet program carries out 6 rounds of message exchanges, just like the previous one. In each round, we can see the update of the sender's send chain key, the encrypted ciphertext (message + HMAC), the update of the receiver's receive chain key, how the ciphertext is decrypted, and how the message is separated from the HMAC tag. Additionally, every 3 rounds, the Diffie-Hellman ratchet is performed, which generates a new root key and thus new chain keys (the Bob's send chain key must match the Alice's receive chain key, and vice versa).



```
First DH public key for Alice: 19
First DH public key for Bob: 9
First shared secret: 6

Round 1:

Original message: Hello, Bob!

Alice send chain key: 0d34f93d14401204c7d3d972b6e7570c425678ac63d80b25863f65ed7f3efee0
Encrypted ciphertext: FFl9OScOoYNiDoW7CgduNfmPgUuw0LaD3K8T0MolXr2MfU7DYGoOAaakBaju68RE8E8h4sl0uLh/RcxK3pXKZCuRqoxBu5d69L048W+mgdk=

Bob receive chain key: 0d34f93d14401204c7d3d972b6e7570c425678ac63d80b25863f65ed7f3efee0
Decrypted ciphertext: Hello, Bob!8b4b8096787b9ab64d5e42935a11d08e1c4a5a9b7067564e04e803f0fcca1369
Bob's decrypted message: Hello, Bob!

Round 2:

Original message: What's up Alice?

Bob send chain key: eedff464b15ddfa0b54e0d722ddde4aedd2b9335d4d9fcff13aad5e675491857
Encrypted ciphertext: eqn0AZzqptfmnzOx0KndQxtlzZE7BifmmSU+O6Rf2sBZLJEi0+DbXsHVEzsOLa2dIKKxUxRwIxGST0GBupL03Yv+FadpzhXIa/Yh70gJMQ9Ehd8XlwUzBwH
v/+Lrjlw/

Alice receive chain key: eedff464b15ddfa0b54e0d722ddde4aedd2b9335d4d9fcff13aad5e675491857
Decrypted ciphertext: What's up Alice?066714ba2d9a87d81baa7e9cb06c13764f387500cee9b5abcb143e7be401a860
Alice's decrypted message: What's up Alice?

Round 3:

Original message: Are you having a good time?

Alice send chain key: f29ac8b5c28ec4d902dea3a4457ef2db3dfe123fdbd6e21f81f8d955e00ea75a
Encrypted ciphertext: fmqTZaWAV3fetjUNPT1ujMvHbHPdXhchcqfCLGCpyMoAounEzLQCYyy+j44QMzNnICNH9eGuhzc6ZRGLvI8Fy1yWVhVzvHuptcKAmR2YLMmVh4GRPMy1rzY
XLcxPQDyi

Bob receive chain key: f29ac8b5c28ec4d902dea3a4457ef2db3dfe123fdbd6e21f81f8d955e00ea75a
Decrypted ciphertext: Are you having a good time?d28feb435537827158cd83a26b2586a004864bb721ef718c5771114833c99338
Bob's decrypted message: Are you having a good time?

...Performing DH ratchet (key exchange)...

Alice and Bob generate new DH key pairs

Alice:
New root key: 32a5fe4ff6b63f74474a0b10f2e6d5df88985c04217864ccd1c4e683c5d6107d
New chain key send: d37074739a2d1afdfa197cf6573b865e6b320f50b38c80317ffa49ebef07e4d4
New chain key receive: 7f5db1e138f38ec40086526fc27f0ff869862747f99460d5b4d1419cb7532e11

Bob:
New root key: 32a5fe4ff6b63f74474a0b10f2e6d5df88985c04217864ccd1c4e683c5d6107d
New chain key send: 7f5db1e138f38ec40086526fc27f0ff869862747f99460d5b4d1419cb7532e11
New chain key receive: d37074739a2d1afdfa197cf6573b865e6b320f50b38c80317ffa49ebef07e4d4
```



```
Round 4:

Original message: Yes! I'll tell you when we see each other.

Bob send chain key: f3bc741b1199bf4ed6db49f2e5328d38332fa82356aa8e279cbbbbfd7e8b9779
Encrypted ciphertext: 1sDJ05l/XQzS/3Qv18eQLw1pEy9Aq5gYyVsIbJf0JKzGf10IJlpq11Sg9A/bBg/28UjAFmHStuiTqCgbck0duWNTCtHvPByAB3D7w860BbWZojBZjq0KzKH
aul/aWriwm56u0fCTIkqeg8b34YPsPQ==

Alice receive chain key: f3bc741b1199bf4ed6db49f2e5328d38332fa82356aa8e279cbbbbfd7e8b9779
Decrypted ciphertext: Yes! I'll tell you when we see each other.adb2ff1748bdfc6107c620742be27f6523c8d7b8be08148adc8e91576bf53388
Alice's decrypted message: Yes! I'll tell you when we see each other.

Round 5:

Original message: Great! I'll be waiting for you

Alice send chain key: 787c06536c0fee5dc0032f58e26a906d4ace6450837f88c443d7e212dea5b05c
Encrypted ciphertext: dawjcOZU8RBWQSO31pa2uIomERlr2nYdHovDJRSTYuiwuRH3y/ajG+LqiglQ7lUySaigiDWXdmUVfXZZBOCK3KMSrjsFjzpWF/YUqIyrS9tIZXfrkVZMBbW
P1cF2+Z8p

Bob receive chain key: 787c06536c0fee5dc0032f58e26a906d4ace6450837f88c443d7e212dea5b05c
Decrypted ciphertext: Great! I'll be waiting for you475aeb8b75133efe0b200524c6c03c064a50b2a202d937ad9e974419b8b74f5f
Bob's decrypted message: Great! I'll be waiting for you
```



```
Round 6:

Original message: See you soon

Bob send chain key: 46d374a5a8d43645c77ae4ec2f0ceb7e4e9a1993fdd07ef44b4e432513bdf1fb
Encrypted ciphertext: XnOujej0cmEhMgG8yUaZtntj9SbWtntXrRj22oBFOatHVmobcVPLPuKZQ94AFjmQ0/Pl7PJjU9GNa9rDHwDxF8q714U4zhdFSvTs0s6neAo=

Alice receive chain key: 46d374a5a8d43645c77ae4ec2f0ceb7e4e9a1993fdd07ef44b4e432513bdf1fb
Decrypted ciphertext: See you soon575b092e09ba5b9e2ee1f9a4255bff61155324641ea8d6fd34df53e71705ac52
Alice's decrypted message: See you soon

...Performing DH ratchet (key exchange)...

Alice and Bob generate new DH key pairs

Alice:
New root key: 37b8708524aff5adbb8277d4e4cda8a9616a28851a6428705b5852e4b1093854
New chain key send: db6cad3d9ee522870ee52d30868223e915a9a045428f98f67239604118faeee9
New chain key receive: 30f3d26e423c9a0815f2d975b1ee4283fdcba9928ad5682c05f2492312eac202

Bob:
New root key: 37b8708524aff5adbb8277d4e4cda8a9616a28851a6428705b5852e4b1093854
New chain key send: 30f3d26e423c9a0815f2d975b1ee4283fdcba9928ad5682c05f2492312eac202
New chain key receive: db6cad3d9ee522870ee52d30868223e915a9a045428f98f67239604118faeee9
```

4. Discussion

In this project, we implemented and tested various cryptographic mechanisms, focusing on the Diffie-Hellman key exchange, HMAC authentication, AES encryption, and ratcheting mechanisms such as Single Ratchet and Double Ratchet. Each of these components plays a crucial role in ensuring secure communication, and their integration forms a robust system for both confidentiality and integrity.

The **Diffie-Hellman key exchange** was successfully implemented in Task 1 to establish a shared secret between Alice and Bob, even when communicating over an insecure channel. By exchanging public keys derived from their private keys, both parties were able to compute the same shared secret independently.

However, one limitation of the Diffie-Hellman key exchange is that it does not inherently provide authentication. This makes the system vulnerable to man-in-the-middle attacks, where an adversary could intercept the public keys. To mitigate this, we integrated **HMAC-based message authentication**.

HMAC was used to ensure the integrity and authenticity of messages exchanged between Alice and Bob. In Task 2, we demonstrated how the shared secret, derived from Diffie-Hellman, was used as a key for HMAC. This mechanism ensured that both the sender and receiver could verify that the message was not tampered with during transmission.

The implementation of **AES in CBC (Cipher Block Chaining) mode** for the Task 3, provided encryption for the messages exchanged between Alice and Bob. AES ensures confidentiality by encrypting the plaintext (message + HMAC tag) into ciphertext that can only be decrypted with the correct key. In our implementation, a 128-bit AES key was generated for each encryption, and a unique IV (Initialization Vector) was used to prevent the same plaintext from producing identical ciphertexts. This added a layer of confidentiality to the communication, ensuring that even if an adversary intercepted the encrypted messages, they would not be able to read the content without the encryption key. During decryption, the program successfully restored the original message by reversing the encryption process and removing any padding.

The **Single Ratchet mechanism** was implemented in Task 4 to provide forward secrecy, ensuring that the encryption keys are continuously updated with each message exchange. This mechanism is crucial for minimizing the damage in case a key is compromised, as it ensures that even if an attacker gains access to a message key, they cannot decrypt previous messages. However, they will be able to predict future keys (post-compromise insecurity). To avoid this, the implementation of the Double Ratchet has been carried out.

The **Double Ratchet mechanism**, implemented in Task 5, extends the concept of the Single Ratchet by incorporating a Diffie-Hellman ratchet alongside the symmetric ratchet. The Double Ratchet ensures both forward secrecy (by updating the chain key) and asynchronous messaging support (by enabling key exchanges through the Diffie-Hellman mechanism), ensuring post-compromise security. This mechanism combined with the Diffie-Hellman shared secret key, the HMAC and the encryption and decryption process, provides **strong encryption, forward secrecy, and message authentication**.

5. Conclusion

The objective of this project was to implement a secure communication framework that combines various cryptographic techniques, including the Diffie-Hellman key exchange, HMAC for message authentication, and AES encryption, along with ratcheting mechanisms to ensure confidentiality, integrity, and authenticity of messages exchanged between parties. The results confirmed that these mechanisms work effectively in tandem; the Diffie-Hellman key exchange established a shared secret key, HMAC ensured message authentication, and AES provided strong encryption for confidentiality. Additionally, the integration of Single and especially Double Ratchet mechanisms enabled continuous key updates, reinforcing security against potential key compromises. For future improvements, I would recommend exploring the implementation of digital signatures for stronger authentication, which could provide an additional layer of protection against man-in-the-middle attacks. This enhancement would make the communication framework even more robust in the face of evolving cyber threats.

References

Lectures 10, 11, 12 and 15 of the course material.

- Implementing Diffie-Hellman:

https://www.thesecuritybuddy.com/encryption/how-does-diffie-hellman-key-exchange-protocol-work/

- Implementing HMAC for Authentication:

https://www.researchgate.net/figure/Hash-Message-Authentication-Code-HMAC-process-332-Overview-of-SHA-256-SHA-256_fig2_346634579

https://www.freeformatter.com/hmac-generator.html#before-output

- Implementing Double Ratchet:

https://signal.org/docs/specifications/doubleratchet/