Tutorial – 9

Exercise – 1

I decided to create a class User and keep values like x_i, P_i and so on in it along with partial keys calculations and such.

```python
# Define the User class with necessary functions
class User:
    def __init__(self, user_id, curve, generator, x_master):
        """
        Initialize a new user with a unique ID, elliptic curve, and generator point.
        """
        self.user_id = user_id  # Unique identifier (e.g., "Alice")
        self.curve = curve
        self.G = generator

        # Step 1: Generate the user's secret value x_i
        self.x_i = randint(1, q-1)

        # Step 2: Compute the user's partial public key P_i = x_i * G
        self.P_i = self.x_i * self.G

        # These will be set later by the KGC
        self.r_i = None
        self.R_i = None
        self.d_i = None

        # Random values for U and V
        self.lA = randint(1,q-1)
        self.U = self.lA * self.G

        self.hA = randint(1,q-1)
        self.V = self.hA * self.G
```

Here we have user_id as the unique ID for each user, curve and generators are the one already defined in the base file given to us.

Functions for partial key calculation

```python
    def generate_hash(self):
        # Generate a hash based on user's information
        hash_input = str(self.user_id) + str(self.R_i.x.value) + str(self.R_i.y.value) + str(self.P_i.x.value) + str(self.P_i.y.value)
        hash_output = int(hashlib.sha256(hash_input.encode()).hexdigest(), 16)
        return hash_output

    def generate_partial_key(self, x_master):
        # Generate partial key for the user
        self.r_i = randint(1,q-1)
        self.R_i = self.r_i * self.G
        H_val = self.generate_hash()
        self.d_i = (self.r_i + x_master * H_val) % q
```

Hash is calculated with user_id, R_i and P_i and then d_i is calculated.

```python
def compute_full_keys(self):
    # Compute full private and public keys for the user
    ski = (self.d_i, self.x_i)
    PKi = (self.R_i, self.P_i)
    return ski, PKi
```

Function to calculate full keys.

```python
# Set up the elliptic curve and the generator G (not shown here)
x_master = randint(1, q-1)
Ppub_master = x_master * G

# Create two users: Alice and Bob
alice = User("Alice", curve256k1, G, x_master)
bob = User("Bob", curve256k1, G, x_master)
```

Initialising x_master and Ppub_master and user alice and bob.

Condition to create full keys also in class User

```python
        if x_master:
            self.generate_partial_key(x_master)

        self.ski, self.PKi = self.compute_full_keys()

    def __repr__(self):
        # This is the string representation of the User class
        return (f"User({self.user_id}):\n"
                f"  x_i = {self.x_i}\n"
                f"  P_i = ({self.P_i.x.value}, {self.P_i.y.value})\n"
                f"  r_i = {self.r_i}\n"
                f"  R_i = ({self.R_i.x.value}, {self.R_i.y.value})\n"
                f"  d_i = {self.d_i}\n"
                f"  Full Private Key (s_ki) = {self.ski}\n"
                f"Full Public Key (P_Ki) = ({self.PKi[0].x.value}, {self.PKi[0].y.value}), "
                f"({self.PKi[1].x.value}, {self.PKi[1].y.value})")
```

Exercise – 2

```
# Random values for U and V
self.lA = randint(1,q-1)
self.U = self.lA * self.G

self.hA = randint(1,q-1)
self.V = self.hA * self.G
```

Calculation of lA, hA, U and V (also done in class User)

```
# Function to compute Y, T, and KAB
def compute_Y_and_T(user1, user2, Ppub_master):
    # Create a string for public key
    pk_str = (str(user2.PKi[0].x.value) + str(user2.PKi[0].y.value) +
              str(user2.PKi[1].x.value) + str(user2.PKi[1].y.value)).encode()

    # Create a hash input and compute the hash
    hash_input = (str(user2.user_id) + pk_str.decode()).encode()
    hash_output = int(hashlib.sha256(hash_input).hexdigest(), 16)

    # Compute Y and T values
    Y = user2.R_i + (hash_output * Ppub_master) + user2.P_i
    T = user1.hA * Y

    # Compute KAB (shared key)
    KAB_input = (str(Y) + str(user1.V) + str(T) + str(user2.user_id) + str(user2.P_i)).encode()
    KAB = int(hashlib.sha256(KAB_input).hexdigest(), 16)
    return Y, T, KAB

# Compute Y, T, and KAB using the above function
Y, T, KAB = compute_Y_and_T(alice, bob, Ppub_master)

# Print the KAB value (shared key)
print(KAB)
```

Function for computing Y and T, also printing KAB to check later.

AES encryption

```python
# AES encryption setup
# Convert KAB to 32 bytes (AES-256 key size)
key = KAB.to_bytes(32, byteorder='big')

# Generate a random plaintext message
plaintext = "This is a test message for encryption.".encode()

# Pad the plaintext to make it a multiple of AES block size (16 bytes)
padded_plaintext = pad(plaintext, AES.block_size)

# Encrypt the message using AES-256 in CBC mode
cipher = AES.new(key, AES.MODE_CBC)  # CBC mode
ciphertext = cipher.encrypt(padded_plaintext)

# To make the ciphertext and the IV (initialization vector) easily usable
iv = cipher.iv
```

Function for encapsulation

```python
# Function to encapsulate message
def encpsulate_message(user1, user2, message, T):
    input_data = [user1.U, message, T, user1.user_id, user2.user_id, user1.P_i, user2.P_i]
    hash_input = ''.join(map(str, input_data)).encode()
    hash_output = int(hashlib.sha256(hash_input).hexdigest(), 16)

    H = hash_output

    W = user1.d_i + user1.lA * H + user1.x_i * H
    phi = (user1.U, user1.V, W)
    return phi

# Encapsulate the message and return the phi
phi = encpsulate_message(alice, bob, ciphertext.hex(), T)
```

Exercise – 3

```python
# Function to reverse Y and T
def reverse_Y_and_T(user2, phi):
    Y = (user2.d_i + user2.x_i) * user2.G
    T = (user2.d_i + user2.x_i) * phi[1]
    return Y, T

# Reverse Y and T to check if the values match
Y_return, T_return = reverse_Y_and_T(bob, phi)

# Check if reversed Y and T match the original ones
if Y_return == Y and T_return == T:
    print("Successfully got Y and T back")

# Function to reverse KAB
def reverse_KAB(Y, user1, T, user2):
    input_data = [Y, user1.V, T, user2.user_id, user2.P_i]
    hash_input = ''.join(map(str, input_data)).encode()
    KAB_reverse = int(hashlib.sha256(hash_input).hexdigest(), 16)
    return KAB_reverse

# Reverse KAB to check if we get the same shared key back
KAB_reverse = reverse_KAB(Y_return, alice, T_return, bob)

# Print the reversed KAB
print(KAB_reverse)
```

Function to calculate Y and T again and check if they are the same.

Also, function to calculate KAB again as well and check if they match. Not sure why the H calculation needed to be done again, I skipped it.

Decryption using KAB_reverse

```python
# Check if KAB matches the reversed KAB
if KAB == KAB_reverse:
    print("Successfully got KAB back")

# AES decryption setup
# Convert KAB_reverse to 32 bytes (AES-256 key size)
key = KAB_reverse.to_bytes(32, byteorder='big')

# Decrypt the message using AES-256
cipher = AES.new(key, AES.MODE_CBC, iv)  # Use the same IV as for encryption
decrypted_padded_plaintext = cipher.decrypt(ciphertext)

# Unpad the decrypted plaintext
decrypted_plaintext = unpad(decrypted_padded_plaintext, AES.block_size)

# Convert the decrypted plaintext back to a string
print(f"Decrypted Message: {decrypted_plaintext.decode()}")
```

Working output

```
aroraan@HP-Elitebook:/mnt/c/TAU/aliceAndBob/tutorial9 (main)$ python3 base_ECC_impl.py
22181142094228353831485477292165227175344187249738095346204365481987029098924
Successfully got Y and T back
22181142094228353831485477292165227175344187249738095346204365481987029098924
Successfully got KAB back
Decrypted Message: This is a test message for encryption.
aroraan@HP-Elitebook:/mnt/c/TAU/aliceAndBob/tutorial9 (main)$
```