



Summative Assessment Task TOR

Table of Contents

Submitted Assessment Task One	2
Submitted Assessment Task Two	3
Submitted Assessment Task Three	4
Submitted Assessment Task Four	5



AIN SHAMS UNIVERSITY
I-Credit Hours Engineering Programs
(i.CHEP)



University of
East London

Submitted Assessment Task One

Student name: Sohayla Ihab AbdelMawgoud Ahmed Hamed UEL ID: 2140646

Module name: Computer and Network Security

Module code: EG7643



CSE451: Computer Networks and Security

Spring 2024 Project Phase One

GROUP 6:

Mohamed Hesham Abouelenin 15P3090

Noorhan Hatem Ibrahim 19P5821

Sohayla Ihab Hamed 19P7343

Contents

Introduction	2
User Requirements	2
Deliverables Schedule and Meetings	2
General Description and Functional Requirements	3
Block Cipher Module	3
Module Working: Advanced Encryption Standard (AES) for Symmetric Encryption	3
Adaptability, Scalability, Reliability Requirements	4
Public Key Cryptosystem Module	4
Module Working	4
Adaptability, Scalability, Reliability Requirements	5
Hashing Module	5
Module Working	6
Key Management Module	7
Module Working	7
Authentication Module	8
Module Working	8
Adaptability, Scalability, Reliability Requirements	8
Internet Services Security Module	9
Module Working: Cryptographic Modules for Internet Service Security	9
Interaction Diagrams	10
Block Cipher Module	10

Public Key Cryptosystem Module	11
Hashing Module	12
Key Management Module	12
Authentication Module	13
Sample Architecture	16
Project Timeline	16

Introduction

This project aims to develop and experiment with different cryptographic techniques by a python developed Secure Communication Suite. The major applications that are planned to be tested are cloud drives and chat rooms.

User Requirements

(User most likely understands basic security concepts)

- As a user, I want to encrypt my messages using a block cipher so that they can be securely transmitted.
- As a user, I want to use public key cryptosystems to securely share keys with my communication partner.
- As a user, I want to verify the integrity of my received messages using hashing functions.
- As a user, I want to manage my cryptographic keys securely.
- As a user, I want to authenticate myself to the system to ensure secure access.
- As a user, I want to secure my internet services using the provided cryptographic modules.

Deliverables Schedule and Meetings

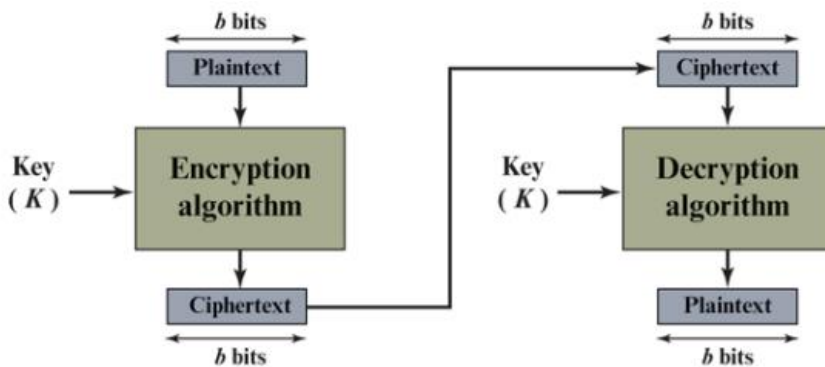
Phase	Deliverables	Tasks	Due
Phase One: Design and Planning	Project plan, software requirements specification, and design documents	1) Meeting One: Divide modules: Block Cipher: Sohayla PKC: Noorhan Hashing: Mohammed KMM: Mohammed Authentication: Noorhan Internet Services: Sohayla 2) Meeting Two: Finalize design diagrams	28/03/2024
Phase Two: Development of Cryptographic Modules	Working code for block cipher, public key cryptosystem, and hashing modules	Divide codes according to previous meetings	25/04/2024
Phase Three: Development of Key Management and Authentication Modules	Working code for key management and authentication modules	Divide codes according to previous meetings	09/05/2024
Phase Four: Integration and Testing	Fully integrated Secure Communication Suite, test cases, and test results	1) On campus meeting to fully integrate all codes 2) Testing	16/05/2024

General Description and Functional Requirements

Block Cipher Module

This module applies a **reversible (non-singular), symmetric** encryption algorithm in which a block of plaintext bits is transformed as a block of b bits into a ciphertext block of the same length. A major advantage of block ciphers is high diffusion meaning that if a single character of a plaintext is changed, then several characters of the ciphertext do change. This makes them fundamental for the building of secure communication protocols.

However, they are slow to function since an entire data block must be encrypted/decrypted. This can be altered through optimizing a block cipher algorithm or mode. Typically, a block cipher mode of operation is a technique to enhance the effect of a cryptographic algorithm, in this case a symmetric block cipher algorithm to encrypt files in a drive (i.e. define how exactly blocks are encrypted). Block ciphers are especially beneficial in cases of data-at-rest encrypted storage, such as data on an online drive, a private cloud server or data preserved as chat media.



Module Working: Advanced Encryption Standard (AES) for Symmetric Encryption

AES specifies a U.S. government- approved cryptographic algorithm that can be used to protect electronic data. This algorithm, as a specification the Rijndael algorithm, works by encrypting a block of 128 bits (or 16 bytes, 4x4 matrix) with a key size of 128, 192 or 256 bits. We will implement this algorithm in **CBC (Cipher Block Chaining)** mode in which each block of plaintext is XORed with the previous ciphertext block before being encrypted.

AES relies on **substitution-permutation network principle** meaning it is performed using a series of replacing and shuffling operations of input data.

The main features of this module are:

- 1) **Substitution-Permutation Network (SPN)**: It works by replacing and shuffling input data which is faster than other applicable algorithms, such as DES.
- 2) **Key Length**: The number of rounds can be varied according to the number of bits in the key. For instance, 128-bit key yields 10 rounds (and 11 keys), a 192-bit key yields 12 rounds (and 13 keys), and a 256-bit key yields 14 rounds (and 15 keys).
- 3) **Key Expansion**: Dependent on key management schemes, a single key is used at first then expanded to multiple keys per round (each call to this module prompts the encryption to run and with each run, there are several rounds)
- 4) **Byte Data**: The algorithm works on 16 bytes of data per plaintext block rather than 128 bits.

Adaptability, Scalability, Reliability Requirements

- **Adaptability Requirements:**
 - This module should be flexible to allow configuration of several scenarios
 - This module should be secure against brute-force attacks as well as possible quantum attacks, even with smaller key-sizes
 - This module should support modern processors and cryptographic hardware modules
 - This module should support cryptographic libraries and frameworks as well as internet service protocols
- **Scalability Requirements:**
 - This module should allow parallel operations for the concurrent processing and encryption
 - This module allows different key lengths in which each type can be used to separate different sizes of data files into equal blocks
- **Reliability Requirements:**
 - This module should allow an unfeasible amount of time per potential attack, especially when using correct key lengths and many permutations

Public Key Cryptosystem Module

This module applies an asymmetric encryption algorithm in which a pair of mathematically-related keys, a public key and a private key, are used to implement secure communication. A person's public key, also known as cryptographic key, is widely available to anyone who wants to send an encrypted message to its owner (the public key encrypts the message). On the other hand, a private key, also known as secret key, is available only to its owner who uses it to decrypt messages that he/she receives. This process ensures that only the intended recipient of the encrypted message is able to access its contents since only the intended recipient's private key can decrypt the message encrypted with his/her public key.

Besides increased data security, another main advantage of Public Key Cryptography is that it works with digital signatures that allow authentication of the sender of a message. If the sender encrypts a message using their private key, the receiver may try decrypting the message using the sender's known public key. If the public key is able to decrypt the message, the identity of the sender is authenticated.

The main drawbacks of Public Key Cryptography are its computational and key-management overheads. Managing private and public key pairs for a large number of users in a secure communication environment is computationally-expensive, as well as resource-expensive, since each user's keys are comprised of large random numbers paired together with mathematical algorithms. We cannot compromise on key length to reduce overhead since that would decrease security by increasing the potential for the key pairing algorithm to be cracked by a hacker. The creation, pairing, storing, and management overheads of these very large random number pairs for each user also cause asymmetric encryption to be typically slower than symmetric encryption.

Module Working

The two most common protocols used for asymmetric cryptography are the Rivest-Shamir-Adleman (RSA) algorithm and the Elliptical Curve Cryptography (ECC) algorithm. The RSA protocol uses prime factorization to encrypt messages while ECC relies on elliptic curves over finite fields for encryption. **We propose using the ECC algorithm** since it requires shorter key length to provide an equal strength of encryption as RSA,

meaning ECC is more efficient. Additionally, ECC is less likely to be compromised than RSA since there is no solution to the equation producing the elliptic curve's mathematical problem. The ECC protocol can only be attacked by brute-force or trial-and-error.

Adaptability, Scalability, Reliability Requirements

- **Adaptability Requirements:**
 - The module should be adaptable to various devices and platforms.
 - The module should be easy to adapt and alter to keep up with emerging customer needs and new technological changes.
- **Reliability Requirements:**
 - The module should be able to handle and deal with sudden errors and unexpected states.
 - The module should have safeguards against possible data loss or data corruption.
 - The module should ensure that the encrypted message is not tampered with during transfer and decryption.
- **Scalability Requirements:**
 - The module should have adequate storage and management capabilities to allow the number of key pairs to be scaled up easily to handle new users.
 - The module should have adequate computational power to be able to handle increased message frequency and bigger message size.
- **Security Requirements:**
 - The module should ensure that private keys remain confidential and untampered with.
 - The module should be resistant to known brute-force cryptographic attacks.
 - The module should comply with the cryptographic industry's standards, regulations, and best practices.

Hashing Module

Hashing is the process of generating size-fixed output from a variable sized input using mathematical function that is known as hashing function.

- Hash functions act like digital fingerprinting tools. They generate a fixed-size output (hash value) for any input data.
- By comparing the hash value of a received file with the original hash value (obtained from a trusted source), we can verify if the data has been tampered with during transmission.
- Hash function should be resistant to collisions. This means it's extremely unlikely for two different pieces of data to produce the same hash value.
- Hash functions are generally designed for efficient computation. They can generate hash values for data relatively quickly, making them suitable for various applications where real-time processing is needed.

- Regardless of the size of the input data, the hash function always produces a fixed-size output (e.g., 256 bits for SHA-256).

- There's always a small theoretical possibility of collisions, especially with shorter hash outputs.

- Hashing functions should be computationally infeasible to determine the original data from a given hash value.

- This constraint relates to the difficulty of finding another data input that produces the same hash output as a specific data input (different from the original data that generated the hash).

- There's a trade-off between processing speed and security with hashing functions. More complex algorithms with longer hash outputs generally offer better collision resistance and other security properties but may require more processing power.

- As computing power increases, even well-established hashing functions might become vulnerable to theoretical or practical attacks over time

Hashing applications:

- Password verification: Instead of encryptions, some websites use password hashing to ensure data integrity. When a user logs into a website that uses hashing, a string generated from the hash function using the user password as input is stored in the database. When a user logs in, the hash function checks the produced strings are similar.

- File downloads: Downloaded files often come with a hash value. Computers can calculate the hash of the downloaded file and compare it to the provided hash to ensure the file hasn't been corrupted during download.

- Maintaining data consistency: When two or more users are communicating, by calculating and comparing the hashed values, inconsistencies can be identified and data integrity can be maintained.

Popular hashing algorithms:

- **SHA-256 (Secure Hash Algorithm):** SHA256 is a widely used cryptographic hash function developed by the National Security Agency (NSA). SHA-256 takes any size data as input and generates a unique, fixed-size (256-bit) hash value as output. This hash value acts like a digital fingerprint for the data.
- **MD5 (Message-Digest Algorithm 5):** MD5 was a widely used hashing function designed by Ronald Rivest in 1991 to replace the MD4; however, nowadays MD5 is not recommended due to discovered vulnerabilities and it is **recommended to use SHA-256 algorithms**.

Module Working

1. **Choose a hashing function:** SHA-256 is more secure option in modern days
2. **Generate hash on the sender side:** When a users sends a message, the app calculates the hash of the message content on the sender device
3. **Send the hash and content:** The app then sends both the original message and the computed hash value.
4. **Verify hash on the receiver:** When message is received, the receiver recalculates the hash of the content

5. Compare and verify:

- a) If the hashes match, it indicates that the message arrived without any alteration during transmission.
- b) If the hashes do not match, it indicates a potential corruption. The app should notify the user about the discrepancy and offer to receive the message again.

Key Management Module

Data is encrypted and decrypted via the use of encryption keys, which means the loss or compromise of any encryption key would invalidate the data security measures put into place. Keys also ensure the safe transmission of data across an Internet connection. With authentication methods, like code signing, attackers could pretend to be a trusted service like Microsoft, while giving victim's computers malware, if they steal a poorly protected key.

- Key management eliminates the need for a pre-shared secret key for secure communication.
- Key management enables digital signatures, a mechanism that allows users to sign messages with their private key.
- Key management offers better resistance to man-in-the-middle attacks.
- Key management can be used to establish secure communication channels with a large number of users without the need to pre-distribute a unique key for each pair. This scalability makes it suitable for applications with many participants.
- Public keys can be widely distributed, while private keys are kept secret. This separation allows for functionalities like public key encryption and private key signing.
- Key management algorithms can be computationally more expensive. Encryption and decryption processes using public and private keys involve complex mathematical operations that require more processing power.
- Key management introduces the complexity of managing key pairs (public and private keys) for each user. Losing a private key can compromise the entire security mechanism.
- Some Key management implementations might be susceptible to side-channel attacks. These attacks exploit information leaked during the cryptographic operations.
- While PKC is great for secure key exchange and encrypting small amounts of data like messages, it's not ideal for encrypting large amounts of data due to its computational overhead.

Module Working

1. When a user wants to chat securely with another user, both users generate their own public-key pairs on their devices.
2. Users securely exchange their public keys through methods as out-of-band channel. Sharing the public key via a trusted channel like email or a secure chat window in another app.
3. Once users have each other's public keys, users can begin to send secure message where the sender sends a message using the receiver's public key and only the receiver's private key can decrypt the message, ensuring confidentiality

Authentication Module

This module applies a verification mechanism in which the system checks a user's login details and compares them with existing user data to allow a user to access their account if it exists in the system, and to ensure that no unauthorized third party can access a user's account. This module is also concerned with giving new users a mechanism to signup and create a new unique account. User authentication is necessary to protect user and data security from malicious actors.

Module Working

There are currently many forms of user verification in use, such as traditional username and password systems, certificate-based authentication, biometrics, token-based authentication, and multi-factor authentication. For the purposes of this project, we have decided to implement a traditional password-based authentication system due to its ease of implementation and integration with the broader project as well as its familiarity to the average user since our target customer is not expected to have high technical skills. Usernames and passwords are also relatively flexible and widely supported by almost all platforms and devices.

Adaptability, Scalability, Reliability Requirements

- **Adaptability Requirements:**
 - The module should be adaptable to various devices and platforms.
 - The module should be easy to adapt and alter to keep up with emerging customer needs and new technological changes.
- **Reliability Requirements:**
 - The module should be able to perform consistently and reliably without granting access to someone who is not authorized or blocking an authorized person from accessing their account.
 - The module should be able to handle and deal with sudden errors and unexpected states.
- **Scalability Requirements:**
 - The module be able to accommodate new accounts being registered.
 - The module should be able to accommodate periods of high login frequency.
- **Security Requirements:**
 - The module should be resistant to malicious attacks and attempts at unauthorized account access.
 - The module should safeguard against brute-force attacks like credential stuffing and phishing.
 - The module should comply with the industry's standards, regulations, and best practices.

Internet Services Security Module

This module applies a variety of cryptographic modules to secure internet services such as communication and cloud storage services. The techniques included in our suite are shown below.

Module Working: Cryptographic Modules for Internet Service Security

AES: This technique can be applied via APIs, such as Google Drive API, to encrypt files at rest. This ensures that even if unauthorized access to physical data is granted, the secrecy of the data isn't breached.

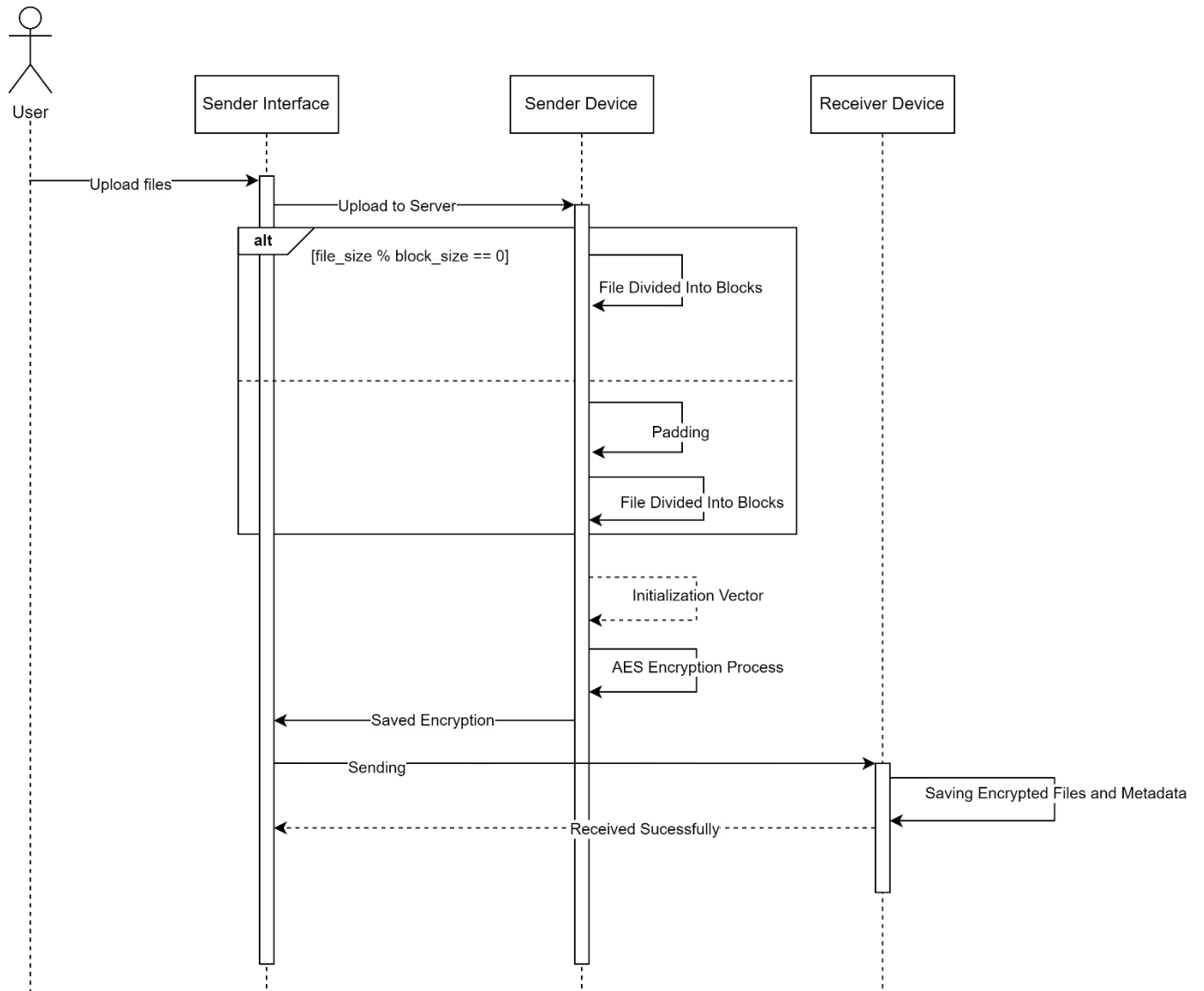
Key Management: This technique is used to safeguard encryption keys which are typically stored separate from files.

Authentication: Users must authenticate themselves in order to access and alter the data.

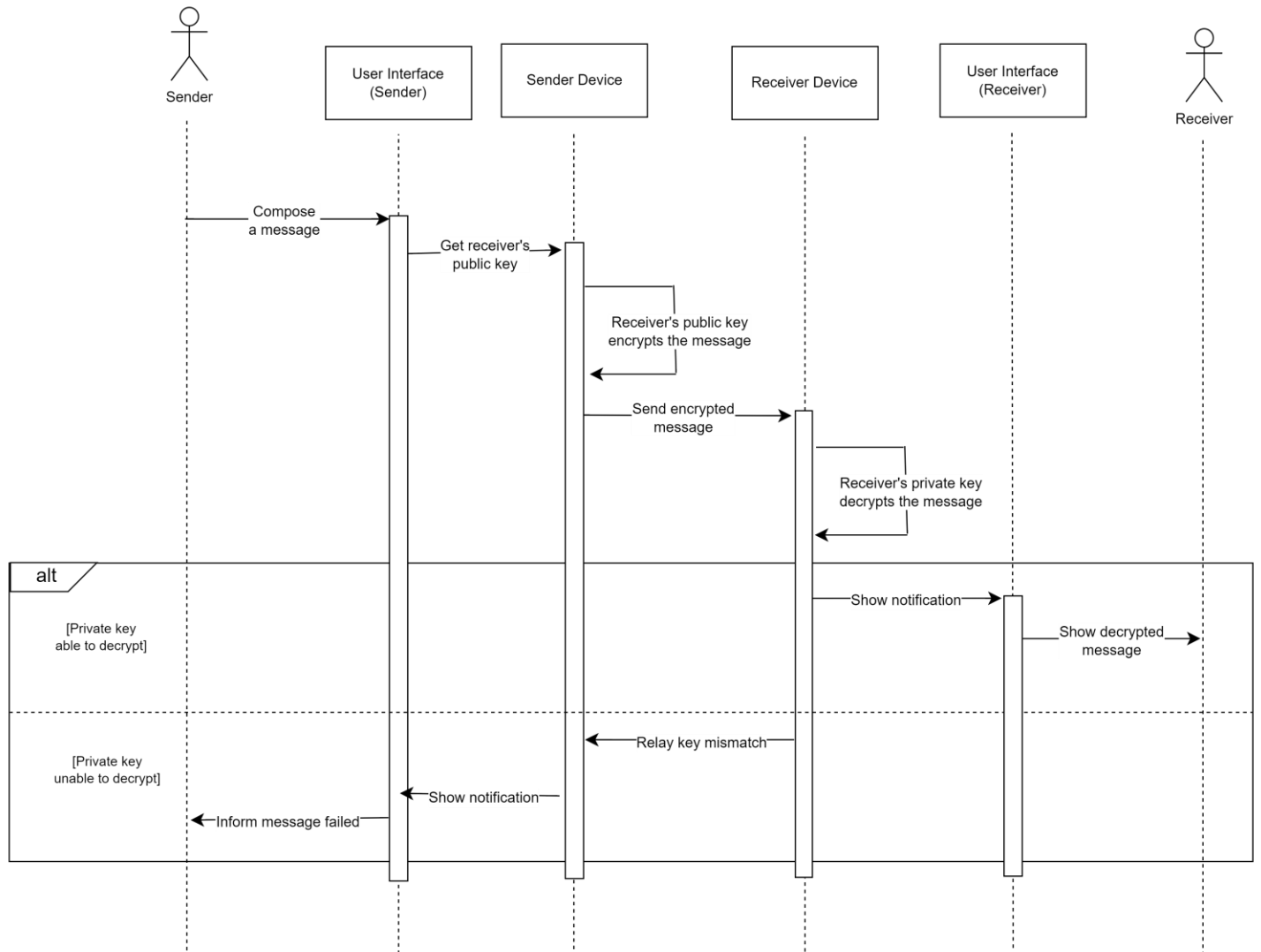
Data Integrity: Techniques such as hashing are applied to preserve sent and received data, such as data uploaded to a cloud storage drive or data sent over chat

Interaction Diagrams

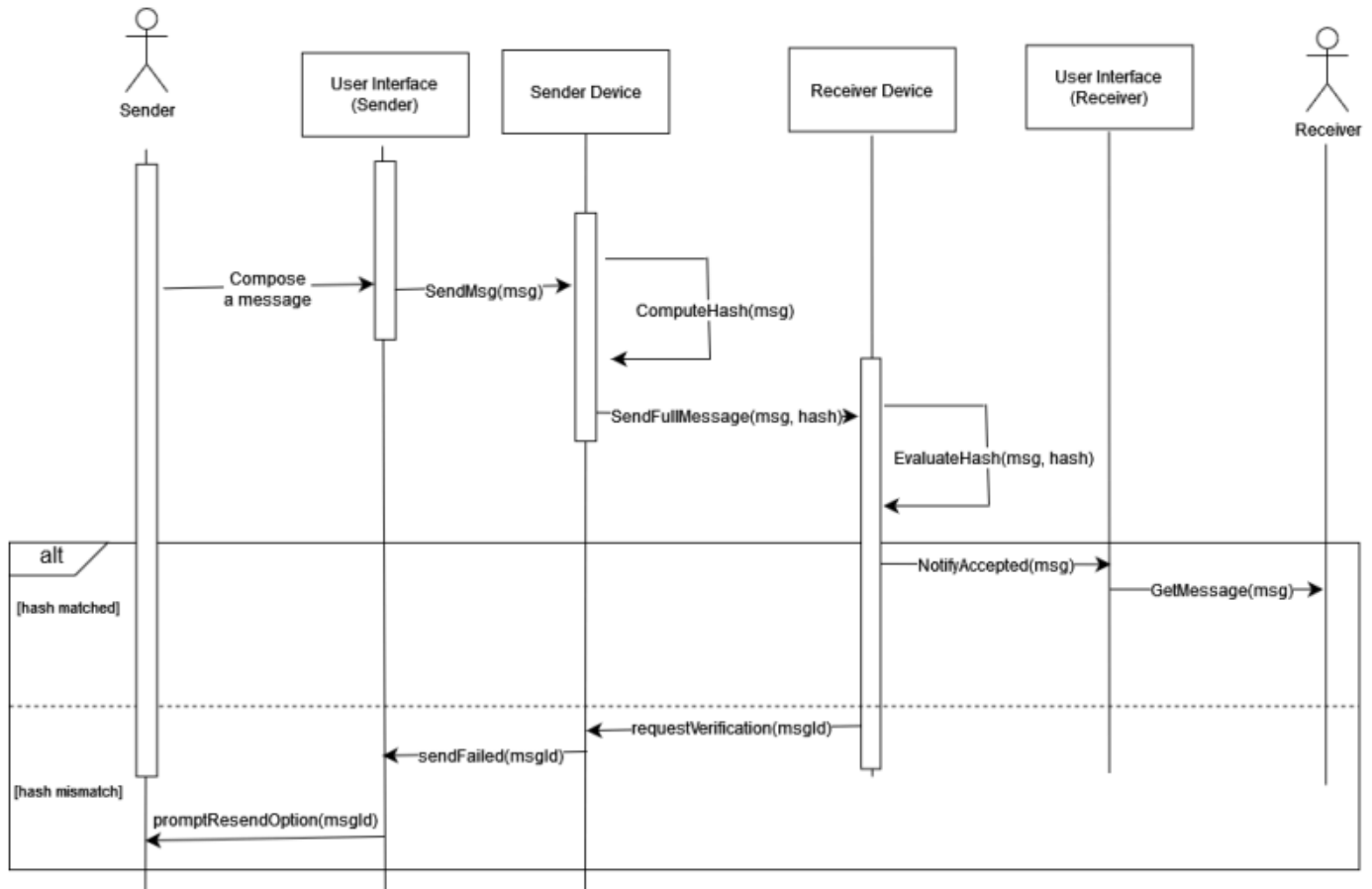
Block Cipher Module



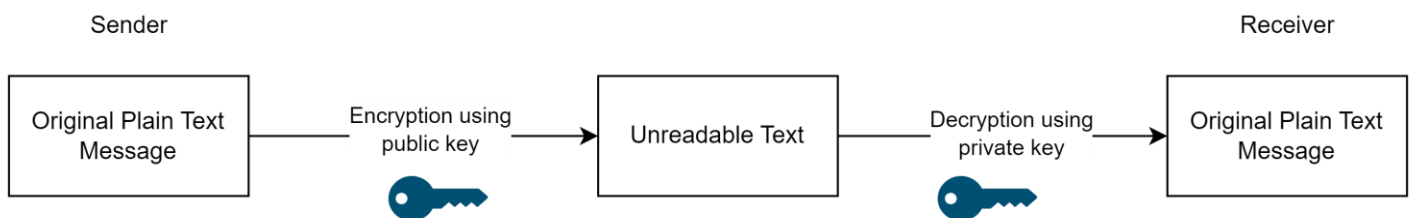
Public Key Cryptosystem Module



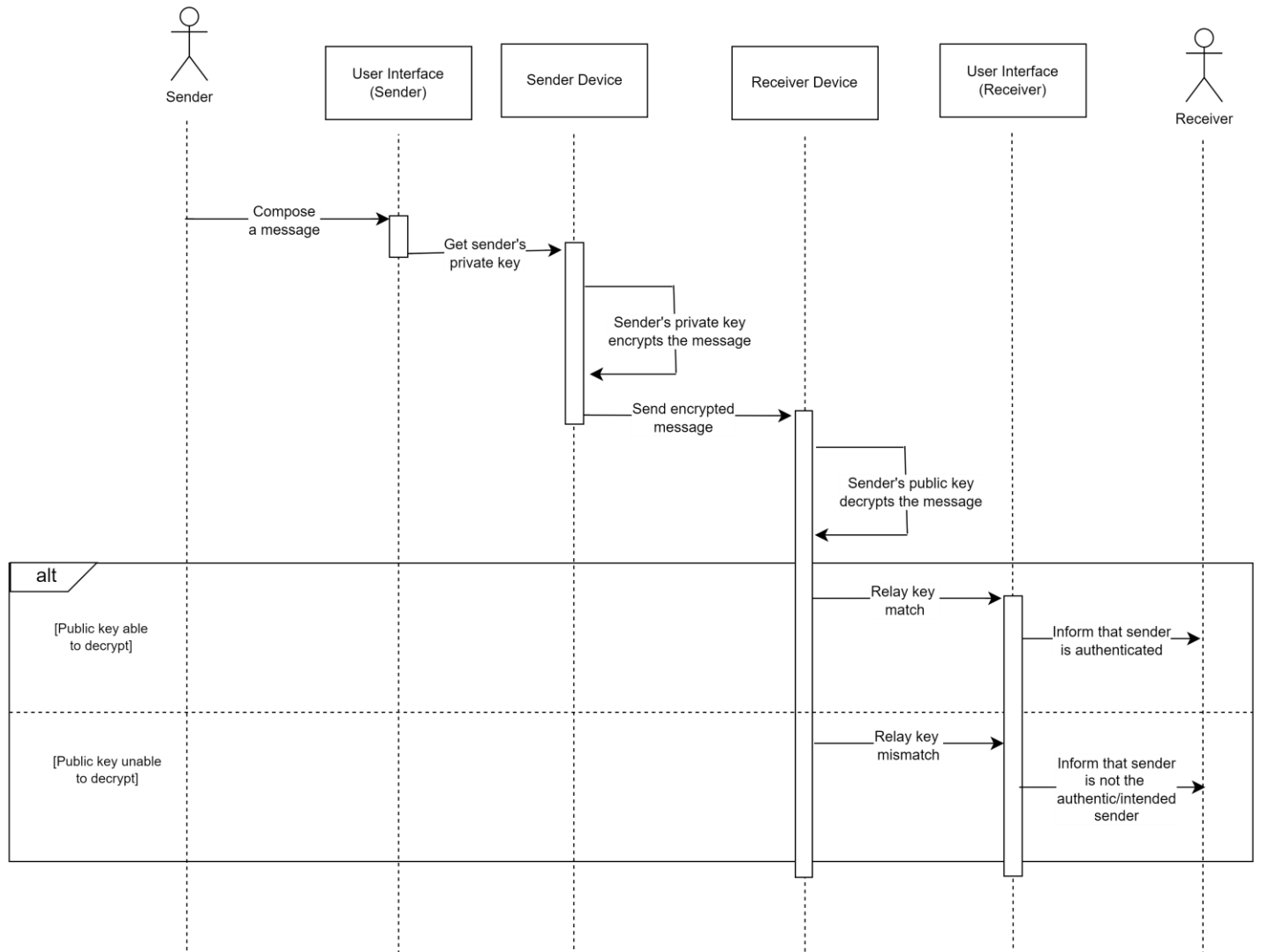
Hashing Module

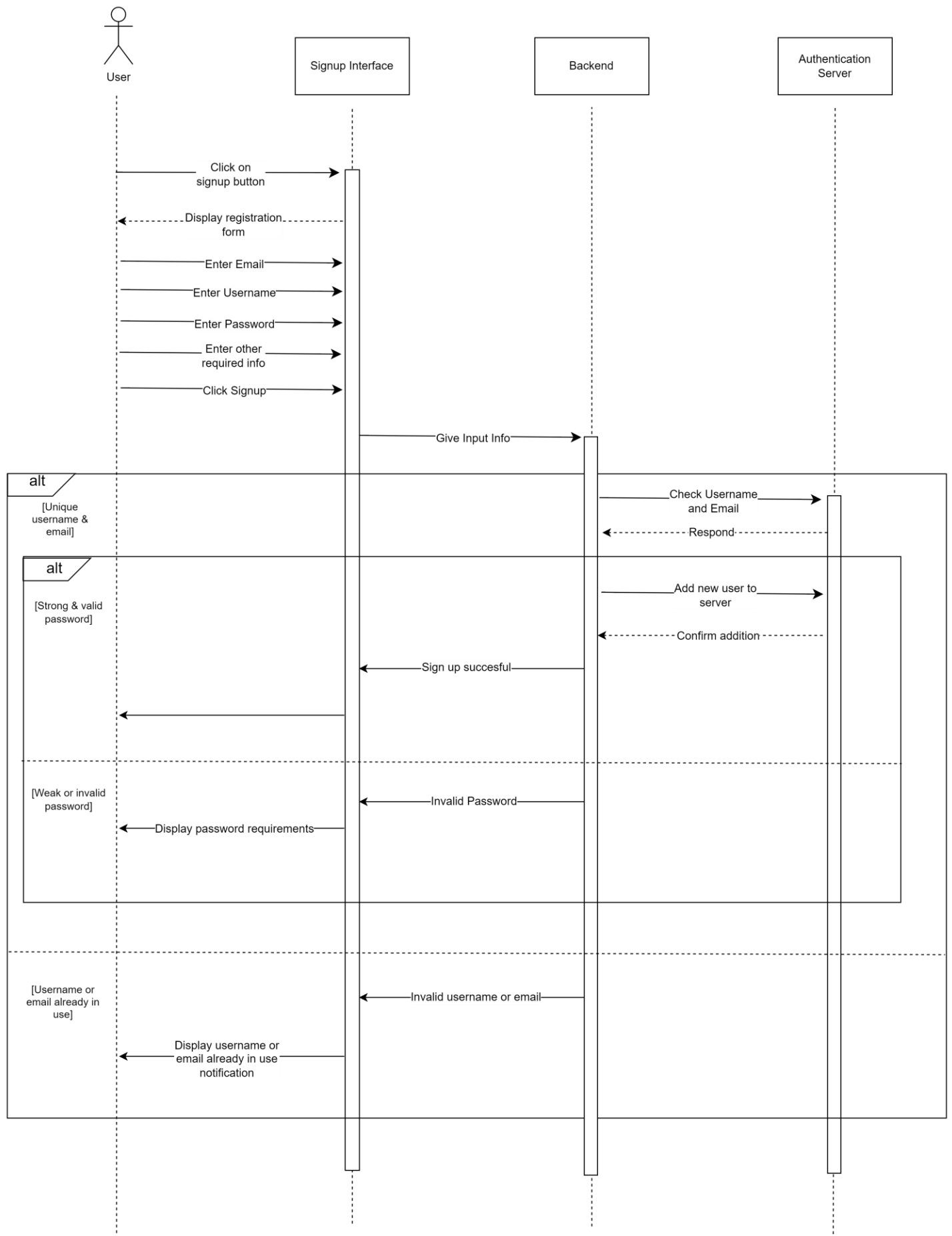


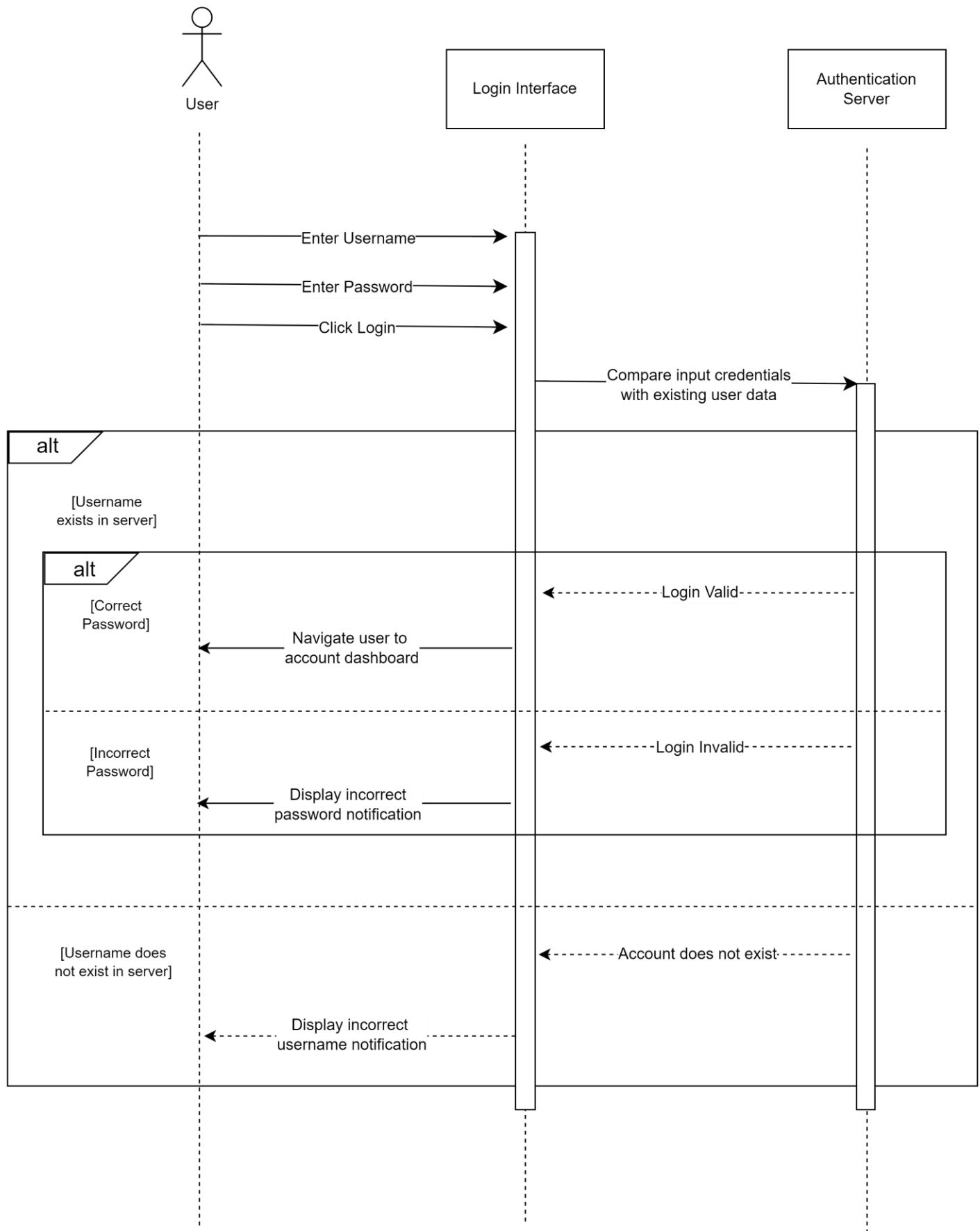
Key Management Module



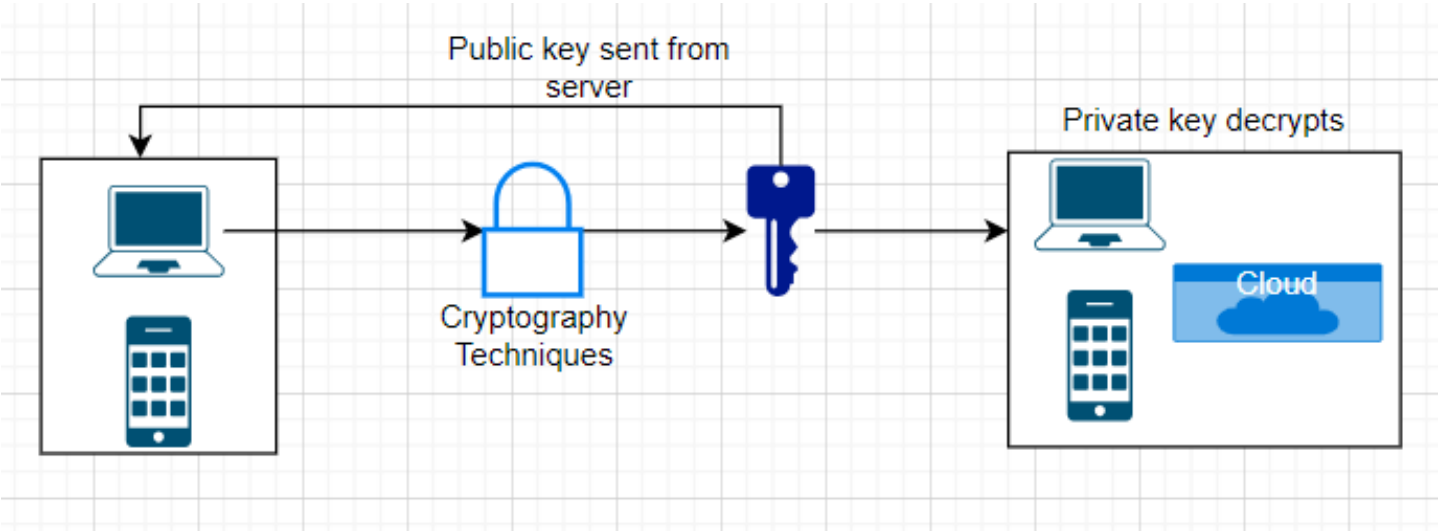
Authentication Module







Sample Architecture



Project Timeline

Task Name	Jan	Feb	Mar	Apr	May	June	July
Phase One							
Phase Two							
Phase Three							
Phase Four							



AIN SHAMS UNIVERSITY
I-Credit Hours Engineering Programs
(i.CHEP)



University of
East London

Submitted Assessment Task Two

Student name: Sohayla Ihab AbdelMawgoud Ahmed Hamed UEL ID: 2140646

Module name: Computer and Network Security

Module code: EG7643



CSE451: Computer Networks and Security

Spring 2024 Project Phase Two

GROUP 6:

Mohamed Hesham Abouelenin 15P3090

Noorhan Hatem Ibrahim 19P5821

Sohayla Ihab Hamed 19P7343

Contents

Introduction	2
Deliverables Schedule and Meetings	2
Block Cipher Module	3
Part One: AES (Advanced Encryption Standard), Mode of Operation Tested: CBC (Cipher Block changing)	3
Code: main_aes.py	3
Test Simulation	4
Test Simulation: Triple AES	4
Part Two: DES (Data Encryption Standard), Mode of Operation Tested: OFB (Output Feedback)	5
Code: main_des.py	5
Test Simulation: Main Operation	6
Test Simulation: Triple DES	6
Code	7
Hashing Module	10
Code	10
Demonstration	11
Download File check	11
Code	11
Running example	12
Conclusion	12

Introduction

This phase is the beginning of the full code development of this project, which mainly works in chatrooms. The main modules discussed here are block cipher, public key cryptography and hashing modules. Code is attached in the rest of this project deliverable. Version one of all code is given here. The rest of the version control will be later attached as a GitHub repository (https://github.com/PerfectionistAF/End_to_end_Chat)

Deliverables Schedule and Meetings

Phase	Deliverables	Tasks	Due	Status
Phase One: Design and Planning	Project plan, software requirements specification, and design documents	1) Meeting One: Divide modules: Block Cipher: Sohayla PKC: Noorhan Hashing: Mohammed KMM: Mohammed Authentication: Noorhan Internet Services: Sohayla 2) Meeting Two: Finalize design diagrams	28/03/2024	DONE
Phase Two: Development of Cryptographic Modules	Working code for block cipher, public key cryptosystem, and hashing modules	Divide codes according to previous meetings 1) Meeting One: Discuss integration concerns	25/04/2024	DONE
Phase Three: Development of Key Management and Authentication Modules	Working code for key management and authentication modules	Divide codes according to previous meetings	09/05/2024	
Phase Four: Integration and Testing	Fully integrated Secure Communication Suite, test cases, and test results	1) On campus meeting to fully integrate all codes 2) Testing	16/05/2024	

Block Cipher Module

- As a user, I want to encrypt my messages using a block cipher so that they can be securely transmitted.

Part One: AES (Advanced Encryption Standard), Mode of Operation Tested: CBC (Cipher Block changing)

First, we import **threading**, **socket**, **pycryptodome (Crypto.AES)** to allow a user or a client (in the case of a chatroom) to send encrypted messages. This is a symmetric algorithm which means that the same key is used to decrypt a ciphertext. The program starts generating a 32 random byte salt and then uses a password, that can be set by the user, along with **PBKDF2 (Password-Based Key Derivation Function 2)** protocol to generate the key (in this example, a 32 bit key is generated for the CBC mode of operation, other modes of operation will be chosen by the user in later versions or tests of the code). This protocol is used to protect the password given by the user. In other versions of the code, a **nonce** is used to preserve the key generation process even if the password is constant. The ciphertext is output after the user message, in bytes, is padded according to the **AES.block_size**.

By the end of the code, the encrypted text is exported to a binary file to be saved for future implementation or simply user activity backup or history, and by that the date and time of the encryption is printed. For testing purposes, the salt and key are exported, which later aids in the regeneration of the same key even if it unknown (since a database and a test vector hasn't been established yet).

Code: main_aes.py

```
from Crypto.Random import get_random_bytes
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from datetime import datetime

salt = get_random_bytes(32) ##at first generate salt then get your key, export salt and password to
use for the same cipher generation
#salt = b'P.\xb8g\xdf\xdc\x87\xec\x9f\x84c\x8at\xb3T\xfc\xeb\xb7\xc5gI\xcc\xdd4\xaa\xa1\x14o\xe1Sq\x9f'
password = "mypassword" ##user can set password
#pass = input("Set your password: ")
#print(salt)

key = PBKDF2(password, salt, dkLen=32)

message = input("YOU: ")
cipher = AES.new(key, AES.MODE_CBC)

ciphertext = cipher.encrypt(pad(bytes(message, 'utf-8'), AES.block_size))
print("Ciphertext: ", ciphertext)
#export to aes_encrypted_test.bin ##to mimic history and backup
with open('aes_encrypted_test.bin', 'wb') as f:
    f.write(cipher.iv)
    f.write(ciphertext)
    print('\nTaken at: ', datetime.now(), '\n')

#time to decrypt
with open('aes_encrypted_test.bin', 'rb') as f:
    iv = f.read(16)
    decrypt_data = f.read()

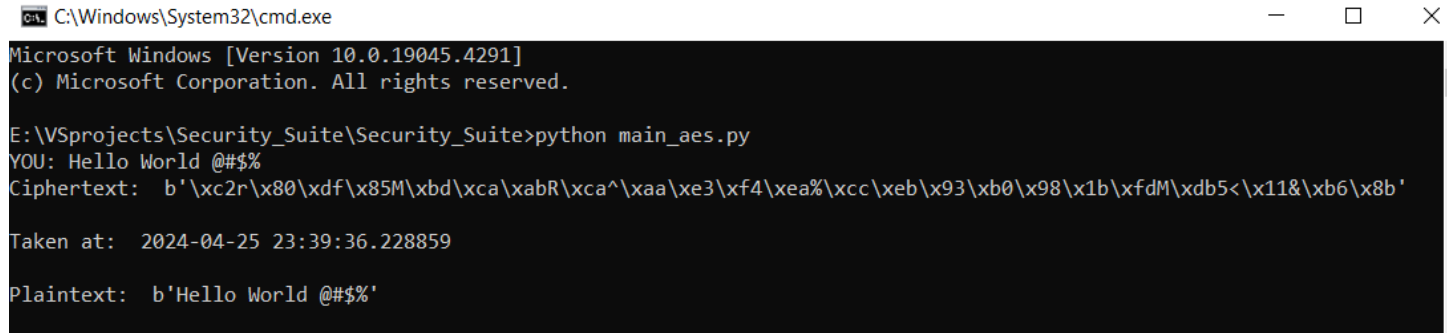
cipher = AES.new(key, AES.MODE_CBC, iv = iv)
plaintext = unpad(cipher.decrypt(decrypt_data), AES.block_size)
print("Plaintext: ", plaintext)

#export key and use on multiple files
```

```
#for testing purposes
with open('aes_key.bin', 'wb') as f:
    f.write(key)

with open('aes_salt.bin', 'wb') as f:
    f.write(salt)
```

Test Simulation






```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

E:\VSprojects\Security_Suite\Security_Suite>python main_aes.py
YOU: Hello World @#$$
Ciphertext:  b'\xc2r\x80\xdf\x85M\xbd\xca\xabR\xca^\xaa\xe3\xf4\xea%\xcc\xeb\x93\xb0\x98\x1b\xfdM\xdb5<\x11&\xb6\x8b '

Taken at:  2024-04-25 23:39:36.228859

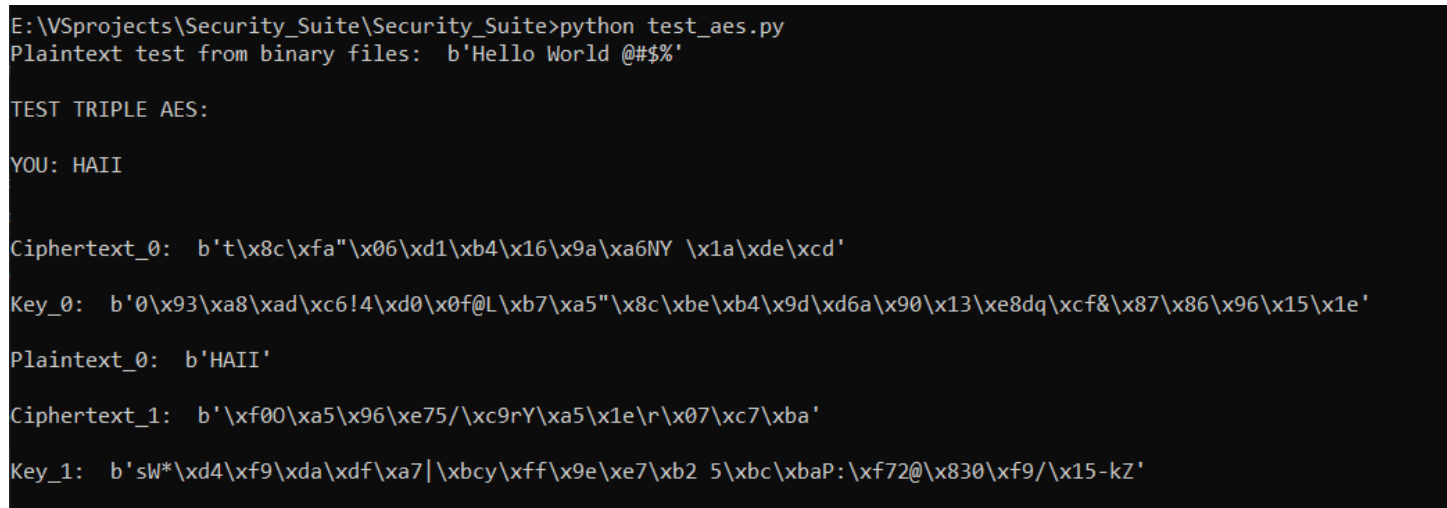
Plaintext:  b'Hello World @#$$'
```

-  aes_encrypted_test.bin
-  aes_key.bin
-  aes_salt.bin

Test Simulation: Triple AES

ENCRYPTION-----DECRYPTION-----ENCRYPTION

SHOULD LENGTHEN KEY TO STOP ITS POSSIBLE EXPLOITATION



```
E:\VSprojects\Security_Suite\Security_Suite>python test_aes.py
Plaintext test from binary files:  b'Hello World @#$$'

TEST TRIPLE AES:

YOU: HAIH

Ciphertext_0:  b't\x8c\xfa"\x06\xd1\xb4\x16\x9a\xa6NY \x1a\xde\xcd'

Key_0:  b'0\x93\xa8\xad\xc6!4\xd0\x0f@L\xb7\xa5"\x8c\xbe\xb4\x9d\xd6a\x90\x13\xe8dq\xcf&\x87\x86\x96\x15\x1e'

Plaintext_0:  b'HAIH'

Ciphertext_1:  b'\xf00\xa5\x96\xe75/\xc9rY\xa5\x1e\r\x07\xc7\xba'

Key_1:  b'sW*\xd4\xf9\xda\xdf\xa7|\xbcy\xff\x9e\xe7\xb2 5\xbc\xbaP:\xf72@\x830\xf9/\x15-kZ'
```

Part Two: DES (Data Encryption Standard), Mode of Operation Tested: OFB (Output Feedback)

Here, all the requirements were the same as Part One, with the exception to ***pycryptodome (Crypto.AES)*** where we will be using ***pycryptodome (Crypto.DES)*** instead. In the DES, we use the same 32 byte long salt and the, except that to perform the OFB mode of operation, an 8 byte long key is required. In addition, the input vector, ***iv***, should be 8 bytes.

Code: main_des.py

```
from Crypto.Random import get_random_bytes
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Cipher import DES
from datetime import datetime
##experiment without professional method
salt = get_random_bytes(32)
password = "mypassword" ##user can set password
#pass = input("Set your password: ")
#print(salt)

key = PBKDF2(password, salt, dkLen=8)

message = input("YOU: ")
cipher = DES.new(key[0:8], DES.MODE_OFB)

ciphertext = cipher.encrypt(bytes(message, 'utf-8'))
print("Ciphertext: ", ciphertext)
#export to aes_encrypted_test.bin ##to mimic history and backup
with open('des_encrypted_test.bin', 'wb') as f:
    f.write(cipher.iv)
    f.write(ciphertext)
    print('\nTaken at: ', datetime.now(), '\n')

#time to decrypt
with open('des_encrypted_test.bin', 'rb') as f:
    iv = f.read(8)
    decrypt_data = f.read()

cipher = DES.new(key, DES.MODE_OFB, iv = iv)
plaintext = cipher.decrypt(decrypt_data)
print("Plaintext: ", plaintext)

#export key and use on multiple files
#for testing purposes
with open('des_key.bin', 'wb') as f:
    f.write(key)

with open('des_salt.bin', 'wb') as f:
    f.write(salt)
```


Test Simulation: Main Operation


```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.


E:\VSprojects\Security_Suite\Security_Suite>python main_des.py
YOU: Another DES Test
Ciphertext:  b'\x89S\x12I|\xe66\xc4\xbe*\xaaf\x10\x0f\x1f\xf9'

Taken at:  2024-04-25 23:43:51.365664

Plaintext:  b'Another DES Test'
```

 des_encrypted_test.bin

 des_key.bin

 des_salt.bin

Test Simulation: Triple DES

ENCRYPTION-----DECRYPTION-----ENCRYPTION

SHOULD LENGTHEN KEY TO STOP ITS POSSIBLE EXPLOITATION

```
E:\VSprojects\Security_Suite\Security_Suite>python test_des.py
Plaintext test from binary files:  b'Another DES Test'

TEST TRIPLE DES:

YOU: HAIII

Ciphertext_0:  b'\xd9\xff\x9c\xbc1'
Key_0:  b"\xc0\xb4\x13\x0f'\xc7sP"
Plaintext_0:  b'HAIII'

Ciphertext_1:  b'\xe0\xa0\xeb/\x94'
Key_1:  b'\x1f4\x16\x08?\x81<;'
```

Public Key Cryptosystem Module

- As a user, I want to use public key cryptosystems to securely share keys with my communication partner.

Part One: Rivest–Shamir–Adleman (RSA) Cryptosystem

First, we start by importing the **threading**, **socket**, and **rsa** modules to create a python program that allows for two users (a host and a client) to send and receive RSA-encrypted messages. When a user runs the program, a public and private key pair of length 1024 bits are generated and assigned to them.

The program then asks the user to choose between being a host or a client to another host. If the user chooses to be a host, they will listen until a client connects to them. Once a connection is made, the host sends their public key to the client and then receives the client's public key. If the user chooses to be a client to an existing host, they first receive the host's public key and then send their own key.

After the keys are exchanged, the host and client utilize two functions **send_msg** and **receive_msg** to communicate securely together. The **send_msg** function continuously prompts the user for input (plaintext messages) which it encrypts using the public key received from the communication partner then sends. The **receive_msg** function continuously takes ciphertext sent by the communication partner and decrypts it using the recipient's (local) private key and displays the message.

The program creates two threads, one for sending messages and one for receiving messages.

Code

```
import threading
import socket
import rsa

public_key, private_key = rsa.newkeys(1024)
partner_public_key = None

choice = input("Enter (1) to become a host. Enter (2) or to connect to existing host: ")

if choice == "1":
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("192.168.220.1", 9999))
    server.listen()
    client, _ = server.accept()

    # Send public key
    client.send(public_key.save_pkcs1("PEM"))
    # Receive partner's public key
    partner_public_key = rsa.PublicKey.load_pkcs1(client.recv(1024))
elif choice == "2":
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(("192.168.220.1", 9999))

    # Receive partner's public key
    partner_public_key = rsa.PublicKey.load_pkcs1(client.recv(1024))
    # Send public key
    client.send(public_key.save_pkcs1("PEM"))
else:
    print("Invalid input.")
    exit()
```

```
def send_msg(client):
    while True:
        message = input("")
        client.send(rsa.encrypt(message.encode(), partner_public_key))
        print("You: " + message)

def receive_msg(client):
    while True:
        print("Client: " + rsa.decrypt(client.recv(1024), private_key).decode())

threading.Thread(target=send_msg, args=(client,)).start()
threading.Thread(target=receive_msg, args=(client,)).start()
```

Communication Suite Example

```
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\noorh>cd C:\Users\noorh\PycharmProjects\publickeycrypto

C:\Users\noorh\PycharmProjects\publickeycrypto>python main.py
Enter (1) to become a host. Enter (2) or to connect to existing host:
1
Hello
You: Hello
How are you?
You: How are you?
Partner: I'm fine. You?
I'm OK.
You: I'm OK.

Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\noorh>cd C:\Users\noorh\PycharmProjects\publickeycrypto

C:\Users\noorh\PycharmProjects\publickeycrypto>python main.py
Enter (1) to become a host. Enter (2) or to connect to existing host:
2
Partner: Hello
Partner: How are you?
I'm fine. You?
You: I'm fine. You?
Partner: I'm OK.
```

Part Two: Elliptic-Curve Cryptography (ECC)

Our ECC program is designed to establish a secure communication channel between two users (a host and client) like the aforementioned RSA program. The ECC program uses threading, sockets and allows the user to choose between being a host or client like in the RSA program. However, it differs from RSA by generating a private and public key pair for each user using the SECP256R1 elliptic curve.

After a connection between the host and client is established and their public keys are shared, the program derives a **shared key** using the ECDH protocol for each user using the user's local private key and the public key received from their partner. The derived key is further processed using HKDF to create a symmetric encryption key. This generated key is to be used for symmetric encryption.

The ECC system is incomplete as it is going to be integrated, in the coming milestones, with the AES system we created (integration is not required for this milestone). The ECC program will involve AES as an encryption/decryption protocol using the processed derived key. The following code is complete and functional except for the **send_msg** and **receive_msg** functions.

Code

```

import threading
import socket
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.backends import default_backend

# Generating ECC key pair
private_key = ec.generate_private_key(ec.SECP256R1(), default_backend())
public_key = private_key.public_key()
public_partner = None

choice = input("Enter (1) to become a host. Enter (2) or to connect to existing host: ")

if choice == "1":
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("192.168.220.1", 9999))
    server.listen()
    client, _ = server.accept()

    # Send public key
    client.send(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ))
    # Receive partner's public key
    public_partner = serialization.load_pem_public_key(client.recv(1024),
backend=default_backend())

elif choice == "2":
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(("192.168.220.1", 9999))

    # Receive partner's public key
    public_partner = serialization.load_pem_public_key(client.recv(1024),
backend=default_backend())
    # Send public key
    client.send(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ))

else:
    exit()

def derive_shared_key(private_key, public_partner):
    shared_key = private_key.exchange(ec.ECDH(), public_partner)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)
    return derived_key

def send_msg(client, derived_key):
    while True:
        message = input("")
        # Encryption to be implemented using derived key and AES
        print("You: " + message)

```

```
def receive_msg(client, derived_key):
    while True:
        # Decryption to be implemented using derived key and AES
        print("Partner: " + message)

threading.Thread(target=send_msg, args=(client,)).start()
threading.Thread(target=receive_msg, args=(client,)).start()
```

Hashing Module

- As a user, I want to verify the integrity of my received messages using hashing functions.

Our Hashing Module uses Python's **hashlib** library to hash and verify data integrity or passwords.

Code

```
import hashlib
def hash_data(data):
    # Insure that the data is a string or byte array
    if isinstance(data, str):
        data = data.encode("utf-8")
    # Use SHA256 Algorithm
    hash = hashlib.sha256()
    # Set data to be hashed
    hash.update(data)
    # Return hexadecimal digest as a string
    return hash.hexdigest()
```

Using **hashlib**, We define a function called **hash_data** to return a string of 64 hexadecimal characters.

We can use the hash to verify the data received or hash password strings so that database compromise does not leak passwords to any attackers; however, hashing passwords using this snippet is not advisable because:

- **Vulnerability to Brute-Force Attacks:** Although hashing is strong against brute-force attacks, attackers can use brute-force attacks to try different passwords and hash them with the same algorithm. If they find a hash that matches the stored password hash, they gain access to the account. Without a cryptographic key, it's easier to perform such attacks.
- **Rainbow Table Attacks:** Rainbow tables are pre-computed tables that map possible plaintexts (passwords) to their corresponding hashes. Attackers can use these tables to quickly reverse a hash and obtain the original password. The effectiveness of rainbow tables depends on the hashing algorithm and its output size.

A more suitable approach is to use HMAC (keyed-hash message authentication code or hash-based message authentication code), a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key.

Using **hashlib** for data integrity checks is more suitable than for password protection.

When needed to verify data, we can use the hashing algorithm to produce a digest which can be used to compare with other hashed values.

When we try the function with:

```
print(hash_data("Some data"))
```

We produced the following hash

```
PS E:\Projects\Security\Hashing> python .\hash.py
1fe638b478f8f0b2c2aab3dbfd3f05d6dfe2191cd7b4482241fe58567e37aef6
```

Trying it with a minimal difference produced, (for example changing 'd' to 'D')

```
print(hash_data("Some Data"))
```

```
PS E:\Projects\Security\Hashing> python .\hash.py
2d27ec8437ec76ec2db484c98ed89f7793f0575e271518dd1d62a18fde6e202d
```

Notice that we get a completely different hash, which makes the hashing algorithm almost impossible to predict or guess.

Demonstration

Download File check

Here is how hashing can be used for data integrity:

For this example, I will use the [Ubuntu OS](#) (ubuntu-24.04-desktop-amd64.iso image) and compare it with its hash which is located at the [official site](#).

The official team of Ubuntu provided this hash:

"81fae9cc21e2b1e3a9a4526c7dad3131b668e346c580702235ad4d02645d9455"

The purpose is to check if the downloaded file is the same exact piece of software (**data integrity**).

Code

```
CORRECT_HASH =
"81fae9cc21e2b1e3a9a4526c7dad3131b668e346c580702235ad4d02645d9455"
with open("ubuntu-24.04-desktop-amd64.iso", "rb") as f:
    file_bytes = f.read()
    file_hash = hash_data(file_bytes)
    if file_hash == CORRECT_HASH:
        print("File is verify")
    else:
        print("File does not match the hashed value. Please make sure the file
is downloaded safely")
```

We first store the hash given from the source and then hash the downloaded file's bytes using the function we defined earlier.

Running example

```
PS E:\Projects\Security\Hashing> python .\hash_example.py
File is verified
```

When running the script, we find that the file downloaded is secure and we can safely assume that there is no corruption over the file.

If we use a different file, we get an error and then we know that the file downloaded is not the one expected.

Conclusion

1) How does block cipher work?

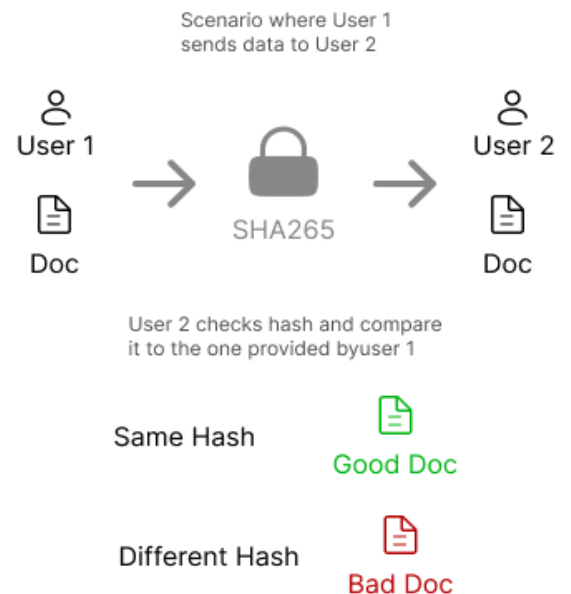
Start with a password, a salt, and a generated key. A ciphertext is output using those 3. To decrypt this text, the input vector is required as well as an unpadding if necessary. This module aims to encrypt the messages between clients in an end to end encrypted chat.

2) How does hashing protect data integrity?

We can ensure data integrity by sending the data's hash along with the data, the receiver checks if the data is the same as the one provided by the sender thus confirming the data integrity. **(As shown in the diagram)**

3) What is the role of the public key cryptosystem module?

As inferred from the user story: "As a user, I want to use public key cryptosystems to securely share keys with my communication partner", the public key cryptosystem module will be used to derive and share an encryption key that will be used for symmetric encryption. This is explained further in the ECC module in this document.





AIN SHAMS UNIVERSITY
I-Credit Hours Engineering Programs
(i.CHEP)



University of
East London

Submitted Assessment Task Three

Student name: Sohayla Ihab AbdelMawgoud Ahmed Hamed UEL ID: 2140646

Module name: Computer and Network Security

Module code: EG7643



CSE451: Computer Networks and Security

Spring 2024 Project Phase Three

GROUP 6:
Mohamed Hesham Abouelenin 15P3090
Noorhan Hatem Ibrahim 19P5821
Sohayla Ihab Hamed 19P7343

Contents

Introduction.....	2
Deliverables Schedule and Meetings.....	2
User Authentication Module	3
Part One: User Authentication via Password.....	3
Code	3
Test Simulation	4
Key Management Module	5
Part One: Implementation	5
Code	5
Data Integrity Update	7
Code	7
Server actions.....	7
Client Action.....	8
Test Simulation	9
Conclusion.....	10

Introduction

This phase is the beginning of the full code development of this project, which mainly works in chatrooms. The main modules discussed here are user authentication and key management modules. Code is attached in the rest of this project deliverable. Version one of all code is given here. The rest of the version control will be later attached as a GitHub repository ([PerfectionistAF/End to end Chat: Security suite to encrypt an end to end chat, using cryptography techniques, socket programming and threading. \(github.com\)](#))

Deliverables Schedule and Meetings

Phase	Deliverables	Tasks	Due	Status
Phase One: Design and Planning	Project plan, software requirements specification, and design documents	1) Meeting One: Divide modules: Block Cipher: Sohayla PKC: Noorhan Hashing: Mohammed KMM: Mohammed Authentication: Noorhan Internet Services: Sohayla 2) Meeting Two: Finalize design diagrams	28/03/2024	DONE
Phase Two: Development of Cryptographic Modules	Working code for block cipher, public key cryptosystem, and hashing modules	Divide codes according to previous meetings 1) Meeting One: Discuss integration concerns	25/04/2024	DONE
Phase Three: Development of Key Management and Authentication Modules	Working code for key management and authentication modules	Divide codes according to previous meetings	09/05/2024	DONE
Phase Four: Integration and Testing	Fully integrated Secure Communication Suite, test cases, and test results	1) On campus meeting to fully integrate all codes 2) Testing	16/05/2024	

User Authentication Module

As a user, I want to authenticate myself to the system to ensure secure access.

Part One: User Authentication via Password

To implement email/password authentication for our communication suite, we decided to use Firebase Authentication because it provides reliable and secure services while being easy to incorporate into a python project using the pyrebase library. The following code is for a simple sign-up/sign-in python program connected to a firebase project (id: securityproject-7a510). In the next phase, this authentication module will be incorporated with the rest of the communication suite.

Code

```
import pyrebase

firebaseConfig = {'apiKey': "AIzaSyD93v3JJazqqg95RZA3raqTu3qNFetYYOI",
                  'authDomain': "securityproject-7a510.firebaseio.com",
                  'projectId': "securityproject-7a510",
                  'storageBucket': "securityproject-7a510.appspot.com",
                  'messagingSenderId': "688231036937",
                  'appId': "1:688231036937:web:6ec8712f306b21819f20d8",
                  'measurementId': "G-WN8NKB7LKH"
                  }

firebase = pyrebase.initialize_app(firebaseConfig)
auth = firebase.auth()

def signin():
    print("SIGN-IN")
    email = input("Email: ")
    password = input("Password: ")
    try:
        signin = auth.sign_in_with_email_and_password(email, password)
        print("Successfully signed in!")
    except:
        print("Incorrect email and/or password.")
    return

def signup():
    print("SIGN-UP")
    email = input("Email: ")
    password = input("Password: ")
    try:
        user = auth.create_user_with_email_and_password(email, password)
        ask = input("Do you want to login? [y/n]")
        if ask == 'y':
            signin()
    except:
        print("Email already used.")
    return

answer = input("Are you a new user? [y/n]")


if answer == 'n':
    signin()
elif answer == 'y':
    signup()
```

Test Simulation

I. The following displays a test sign-up user we created using the program and how it was registered in firebase.

```
Are you a new user? [y/n]y
SIGN-UP
Email: test@xxx.com
Password: test123
Successfully signed up!

Process finished with exit code 0
|
```

Search by email address, phone number, or user UID					Add user	↺	⋮
Identifier	Providers	Created ↓	Signed In	User UID			
test@xxx.com		May 9, 2024		KMuLht6NciQ8CoVXmk82g6h...	...		
Rows per page: 50 ▼ 1 – 1 of 1 < >							

II. The following displays a valid test sign-in.

```
Are you a new user? [y/n]n
SIGN-IN
Email: test@xxx.com
Password: test123
Successfully signed in!

Process finished with exit code 0
|
```

III. The following displays an invalid test sign-in [incorrect password].

```
Are you a new user? [y/n]n
SIGN-IN
Email: test@xxx.com
Password: abc
Incorrect email and/or password.

Process finished with exit code 0
|
```

III. The following displays an invalid test sign-up [reusing an already registered email].

```
Are you a new user? [y/n]y
SIGN-UP
Email: test@xxx.com
Password: password
Email already used.

Process finished with exit code 0
```

Key Management Module

As a user, I want to manage my cryptographic keys securely.

Part One: Implementation

Key Management is a cornerstone of modern cryptography, providing a robust way to ensure the confidentiality and authenticity of data transmissions. One of the popular algorithms is RSA (Rivest–Shamir–Adleman), where a pair of keys are generated for encryption and decryption.

Code

Key Generation:

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)

public_key = private_key.public_key()

pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption(),
)

with open('server_private.pem', 'wb') as f:
    f.write(pem)
```

Decryption:

```
def decrypt_data(data):  
    return private_key.decrypt(  
        data,  
        serialization.Padding.OAEP(  
            mgf=serialization.MGF1(algorithm=serialization.SHA256()),  
            algorithm=serialization.SHA256(),  
            label=None  
        )  
    )
```

Data Integrity Update

As a user, I want to verify the integrity of my received messages using hashing functions.

In this phase, we used our previous hash function to verify files between a server and a host

Code

```
def host_server(IP:str):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((IP, 8876))
    server.listen()
    print("Server is listening....")
    client, _ = server.accept()
    return client, server

def connect_to_host(IP:str):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((IP, 8876))
    print("Client connected!")
    return client
```

Server actions

```
client, server = host_server(IP)
filename = client.recv(1024).decode()
verification_hash = client.recv(1024).decode()

os.makedirs("./storage", exist_ok=True)
file_path = os.path.join("./storage", filename)

if not os.path.exists(file_path):
    with open(file_path, 'x') as file:
        pass
with open(file_path, 'wb') as file:
    while True:
        data = client.recv(1024)
        if not data:
            break
        file.write(data)
with open(filename, 'rb') as file:
    data = file.read().decode()
```

```
    if verify_hash(verification_hash, data):
        print("File transfer integrity is confirmed")
    else:
        print("WARNING: File transfer integrity is not confirmed")
print("File received successfully!")
client.close()
server.close()
```

Client Action

```
client = connect_to_host(IP)
filename = input("Enter filename to send: ")
with open(filename, 'rb') as file:
    data = file.read(1024)
    client.send((filename).encode())
    time.sleep(1) # we sleep to preview data
    verification_hash = hash_data(data.decode())

    time.sleep(1)

    client.send(verification_hash.encode())
    while data:
        client.send(data)
        data = file.read(1024)

print("File sent successfully")
client.close()
```


Test Simulation

Data verified case:

```
● etsh@Cipher End_to_end_Chat % cd DIM
● etsh@Cipher DIM % ls
DataIntegrity.py      storage      text.jpeg      text.txt
● etsh@Cipher DIM % python3 DataIntegrity.py
Do you want to host(1) or do you want to connect(2): 1
Server is listening...
File transfer integrity is confirmed
File received successfully!
○ etsh@Cipher DIM %
```

```
● etsh@Cipher DIM % python3 main.py
/Library/Frameworks/Python.framework/Versions/3.12/Resources/Python.app/Contents/MacOS/Python: can't open file '/Users/etsh/Educational/Security/Project/End_to_end_Chat/DIM/main.py': [Errno 2] No such file or directory
● etsh@Cipher DIM % ls
DataIntegrity.py      text.jpeg
storage               text.txt
● etsh@Cipher DIM % python3 DataIntegrity.py
Do you want to host(1) or do you want to connect(2): 2
Client connected!
Enter filename to send: text.txt
File sent successfully
○ etsh@Cipher DIM %
```

Data verification failure case:

```
● etsh@Cipher DIM % python3 DataIntegrity.py
Do you want to host(1) or do you want to connect(2): 1
Server is listening...
WARNING: File transfer integrity is not confirmed
File received successfully!
○ etsh@Cipher DIM %
```

```
● etsh@Cipher DIM % python3 DataIntegrity.py
Do you want to host(1) or do you want to connect(2): 2
Client connected!
Enter filename to send: text.txt
File sent successfully
○ etsh@Cipher DIM %
```

Conclusion

1) How does the key management module secure key distribution and storage?

We use the RSA algorithm to create a pair of public and private. Both parties create a key of their own and send the public key to the other party. As a server, the clients public key of each client is stored to be used for encrypted communication between the two parties. The client needs only to store its private key for decryption and the server's (host) public key for encryption. The process is similar to Github's SSH Keys.

2) What authentication mechanisms are implemented in the authentication module?

We used Firebase Authentication via the pyrebase python library. Firebase Authentication is a service, provided by Google Firebase, that offers several authentication methods and handles user sign-up, user sign-in, and other account management tasks. We implemented the Firebase email and password authentication method.

3) How does the authentication module verify user identities?

It verifies the validity of a user's email and password during sign-up and sign-in. When signing-up, the module ensures that a new user is not registering with an email already used by another user. When signing-in, the module ensures that the correct email and password are entered.



AIN SHAMS UNIVERSITY
I-Credit Hours Engineering Programs
(i.CHEP)



University of
East London

Submitted Assessment Task Four

Student name: Sohayla Ihab AbdelMawgoud Ahmed Hamed UEL ID: 2140646

Module name: Computer and Network Security

Module code: EG7643



CSE451: Computer Networks and Security

Spring 2024 Project Phase Four

GROUP 6:

Mohamed Hesham Abouelenin 15P3090

Noorhan Hatem Ibrahim 19P5821

Sohayla Ihab Hamed 19P7343

Contents

Introduction.....	2
Deliverables Schedule and Meetings.....	2
Architecture of Internet Services Module.....	3
Steps of Integration.....	3
Authenticate the User	3
Generate Public and Private Keys for the Secure Key Exchange	3
Start Block Cipher to Prepare for Message Exchange.....	4
Key Management and Data Integrity	5
Overall Implementations	6
Prerequisites	6
1) Hybrid Encryption.py.....	7
2) Key Message Passing.py.....	8
Test Suite on Version 1	9
Conclusion.....	9

Introduction

This phase is the integration and testing phase of this project. The main module discussed in this document is the internet security module. A detailed view of the tested architectures, techniques and tests is provided. Code snippets is attached in the rest of this project deliverable. Version one of all code is given here. The rest of the version control will be later attached as a GitHub repository ([PerfectionistAF/End to end Chat: Security suite to encrypt an end to end chat, using cryptography techniques, socket programming and threading. \(github.com\)](#))

Deliverables Schedule and Meetings

Phase	Deliverables	Tasks	Due	Status
Phase One: Design and Planning	Project plan, software requirements specification, and design documents	1) Meeting One: Divide modules: Block Cipher: Sohayla PKC: Noorhan Hashing: Mohammed KMM: Mohammed Authentication: Noorhan Internet Services: Sohayla 2) Meeting Two: Finalize design diagrams	28/03/2024	DONE
Phase Two: Development of Cryptographic Modules	Working code for block cipher, public key cryptosystem, and hashing modules	Divide codes according to previous meetings 1) Meeting One: Discuss integration concerns	25/04/2024	DONE
Phase Three: Development of Key Management and Authentication Modules	Working code for key management and authentication modules	Divide codes according to previous meetings	09/05/2024	DONE
Phase Four: Integration and Testing	Fully integrated Secure Communication Suite, test cases, and test results	1) On campus meeting to fully integrate all codes 2) Testing	16/05/2024	DONE

Architecture of Internet Services Module

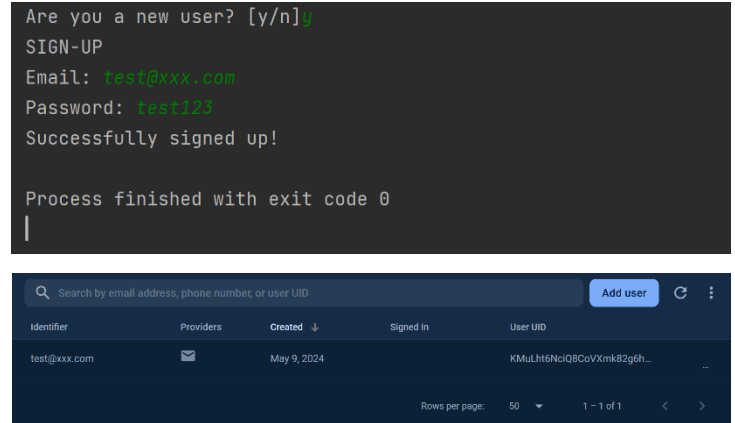
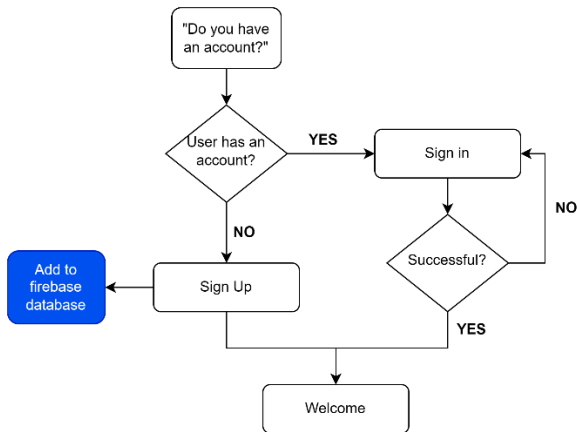
As a user, I want to secure my internet services using the provided cryptographic modules.

Steps of Integration

This module combines all previous modules, authentication, block cipher, public key cryptosystems, hashing and key management. The main objective of this module is to:

- 1) Authenticate users into the suite.
- 2) Securely send encryption and decryption keys.
- 3) Use those keys to encrypt messages.
- 4) Check the integrity of the keys.
- 5) Check users or messages hashes.

Authenticate the User

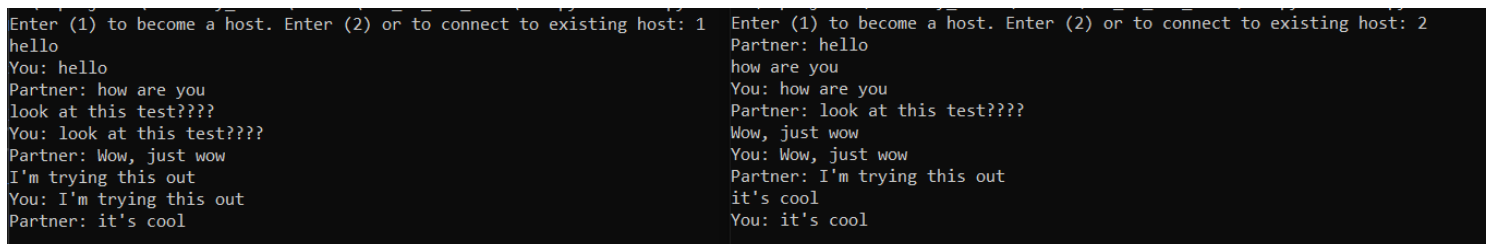
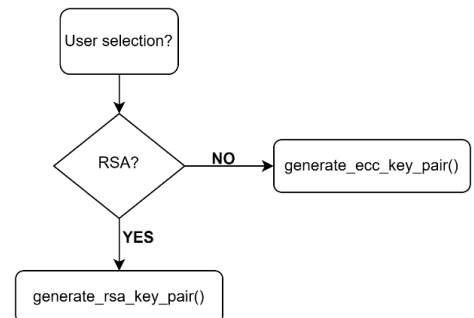


The user is first prompted to enter their credentials, then either a sign in or a sign up occurs. Firebase real time database has the advantage of automatically hashing user credentials, so an additional has is thought to be unnecessary. After correct operation, a welcome screen shows.

Generate Public and Private Keys for the Secure Key Exchange

This module ensures message encryption keys are exchanged securely. Basic algorithms used in this module are RSA and ECC. At first, the user chooses how they want to secure their keys. This method is then coupled with algorithms from block cipher module to securely encrypt data in an end-to-end encrypted chat. The following tests were carried out before integration:

- 1) RSA end to end encrypted chat.
- 2) ECC end to end encrypted chat.



Start Block Cipher to Prepare for Message Exchange

Code Snippet: RSA + AES – SENDING MESSAGES

```
if choice2 == "1":
    client.send('1'.encode())
    # Receive partner's public key
    partner_public_key = rsa.PublicKey.load_pkcs1(client.recv(1024))
    print("Public key received for encryption...")#, partner_public_key)
    #####
    salt = get_random_bytes(16) # AES-128
    # auth.signin.password # use logged in password to create a relationship between confidentiality
    and authentication
    password = "12345"
    symmetric_key = PBKDF2(password, salt, dklen=16)
    cipher = AES.new(symmetric_key, AES.MODE_CBC)
    # Send the public key encoded AES symmetric key
    symmetric_key = binascii.b2a_hex(symmetric_key).decode("utf-8").strip() ## to string
    client.send(rsa.encrypt(symmetric_key.encode(), partner_public_key))
    print("RSA encoded message AES encryption key sent...")#, symmetric_key)
    print("START SENDING MESSAGES NOW...")
    while True:
        message = input("")
        if message == 'exit':
            break
        print("You:", message)
        # Encrypt using symmetric key
        padded_message = pad(bytes(message, 'utf-8'), AES.block_size)
        ciphertext = cipher.encrypt(padded_message)
        cprint("Encrypted YOU: " + str(ciphertext), "yellow")
        iv = cipher.iv
        encrypted_message = iv + ciphertext
        client.send(encrypted_message)
```

Code Snippet: RSA + AES – RECEIVING MESSAGES

```
public_key, private_key = rsa.newkeys(1024)
print("Public key to connections...")#, public_key)
# Send public key
client.send(public_key.save_pkcs1())
# Receive encoded Symmetric Key and decrypt it using private key
encrypted_symmetric_key = client.recv(1024)
symmetric_key = rsa.decrypt(bytes(encrypted_symmetric_key), private_key).decode()
print("RSA decoded AES symmetric key...")#, symmetric_key)
print("START RECEIVING MESSAGES NOW...")
while True:
    encrypted_message = client.recv(1024)
    if not encrypted_message:
        break
    iv = encrypted_message[:16]
    #print("IV: ", iv)
    ciphertext = encrypted_message[16:] ##AES_block_size
    cprint("CIPHERTEXT: " + str(ciphertext), "yellow")
    cipher_aes = AES.new(symmetric_key.encode(), AES.MODE_CBC, iv=iv)
    #print("CIPHER AES: ", cipher_aes)
    #print("BLOCK_SIZE: ", AES.block_size)
    #print(type(unpad(cipher_aes.decrypt(ciphertext), AES.block_size)))
    #print(unpad(cipher_aes.decrypt(ciphertext), AES.block_size).decode())
    #plaintext = unpad(cipher_aes.decrypt(ciphertext), AES.block_size).decode()
    decrypt_plaintext = cipher_aes.decrypt(ciphertext)
    plaintext = binascii.b2a_hex(cipher_aes.decrypt(decrypt_plaintext)).decode("utf-8").strip()
    #decrypt_plaintext = cipher_aes.decrypt(ciphertext)
    cprint("Partner: " + plaintext, "yellow")
```

```
E:\VSpjects\Security Suite\GitHub\End_to_end_Chat\End2End_Chat>python hybrid_enc.py
****WELCOME TO SECURITY SUITE 1.0****
```

WELCOME

```
Enter (1) to become a host. Enter (2) to connect to an existing host: 1
```

```
RSA + AES
```

```
Public key to connections...
```

```
RSA decoded AES symmetric key...
```

```
START RECEIVING MESSAGES NOW...
```

```
CIPHERTEXT: b'\xef\xa4\x9d\xf0\x04\xdc\xab\x12'\t\n\x89'
```

```
Partner: c9e4a7a97f0f805db6f00ca86f13805a
```

```
CIPHERTEXT: b'5\xa1\x13\xf0\xb0\x8e\x1d\x01\xe,\xc7oh\xe4s\xe9'
```

```
Partner: 8ea938ccc99a60f34120507904ef2c0f
```

WELCOME

```
Enter (1) to become a host. Enter (2) to connect to an existing host: 2
```

```
Enter (1) for RSA + AES | (2) for ECC + AES | (3) for RSA + DES | (4) for ECC + DES:1
```

```
Public key received for encryption...
```

```
RSA encoded message AES encryption key sent...
```

```
START SENDING MESSAGES NOW...
```

```
hello
```

```
You: hello
```

```
Encrypted YOU: b'\xef\xa4\x9d\xf0\x04\xdc\xab\x12'\t\n\x89'
```

```
again
```

```
You: again
```

```
Encrypted YOU: b'5\xa1\x13\xf0\xb0\x8e\x1d\x01\xe,\xc7oh\xe4s\xe9'
```

```
exit
```

Key Management and Data Integrity

The required functions do two main things, first is hashing the password salt for the aes and des keys and second checking sent message hashes. The protocol used for password hashing avoids hash collision which means that even if 2 users login with the same password, the generated aes or des keys will not be alike.

Code Snippet: HASHING, KEY MANAGEMENT

```
load_dotenv()
```

```
def hash_data_secretly(data):
    if isinstance(data, str):
        data = data.encode("utf-8")
    key = os.environ["CRYPTO_SECRET_KEY"]
    hash = HMAC(key.encode("utf-8"), data, "SHA256")
    hashed_data = hash.hexdigest()
    return hashed_data
```

```
def verify_secret_hash(hashed_data, data):
    return (hashed_data == hash_data_secretly(data))
```

```
def send_message_with_integrity(plaintext, cipher):
    return json.dumps({
        'message': cipher,
        'hash': hash_data_secretly(plaintext)
    })
```

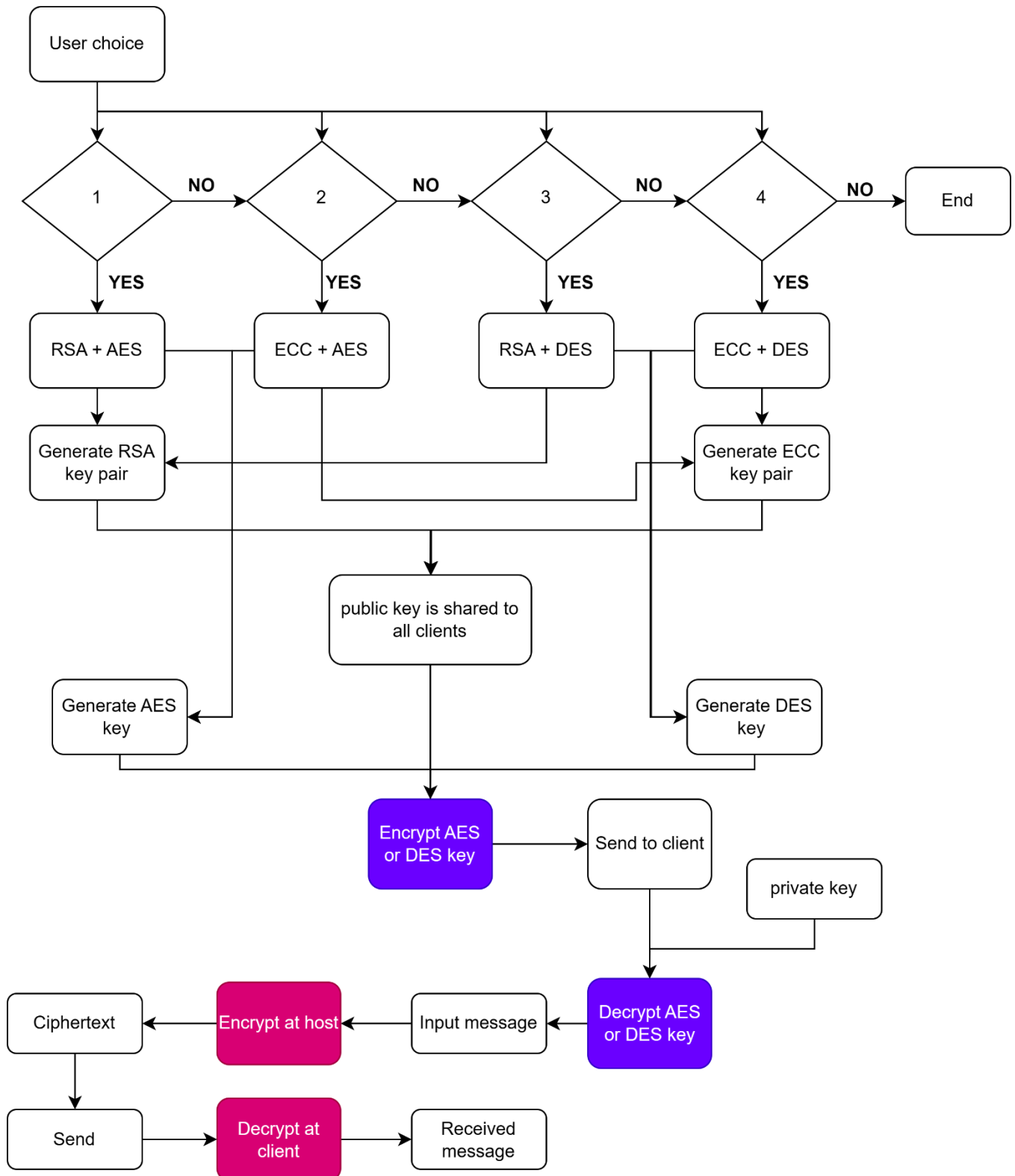
```
def get_message_with_integrity(plaintext: str):
    res = json.loads(plaintext)
    return res['message'], res['hash']
```


Overall Implementations

Prerequisites

```
from Crypto.Cipher import AES, PKCS1_OAEP, DES
from Crypto.PublicKey import RSA, ECC
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
from Crypto.Protocol.KDF import PBKDF2
import threading
import socket
import rsa
import binascii
from unidecode import unidecode
from ecies.utils import generate_eth_key
from ecies import encrypt, decrypt
from colorama import init, Fore, Back, Style
from termcolor import cprint
#import Authentication.auth
import welcome
from hmac import HMAC
import json
from dotenv import load_dotenv
import os
import pyrebase
```

1) Hybrid Encryption.py



2) Key Message Passing.py

```
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
##using aes and rsa
###same as hybrid encryption functions
def generate_rsa_key_pair():
    key = RSA.generate(2048)
    private_key = key.export_key()
    public_key = key.publickey().export_key()
    return private_key, public_key

def encrypt_symmetric_key(sym_key, rsa_public_key):
    rsa_cipher = PKCS1_OAEP.new(RSA.import_key(rsa_public_key))
    encrypted_sym_key = rsa_cipher.encrypt(sym_key)
    return encrypted_sym_key

def decrypt_symmetric_key(encrypted_sym_key, rsa_private_key):
    rsa_cipher = PKCS1_OAEP.new(RSA.import_key(rsa_private_key))
    sym_key = rsa_cipher.decrypt(encrypted_sym_key)
    return sym_key

def encrypt_message(message, sym_key):
    cipher = AES.new(sym_key, AES.MODE_EAX)
    nonce = cipher.nonce
    ciphertext, tag = cipher.encrypt_and_digest(message)
    return nonce, ciphertext, tag

def decrypt_message(nonce, ciphertext, tag, sym_key):
    cipher = AES.new(sym_key, AES.MODE_EAX, nonce=nonce)
    decrypted_message = cipher.decrypt_and_verify(ciphertext, tag)
    return decrypted_message

#Alice
alice_private_key, alice_public_key = generate_rsa_key_pair()
sym_key = get_random_bytes(16) # AES-128
#Use Alice rsa public key to encrypt sym key
encrypted_sym_key = encrypt_symmetric_key(sym_key, alice_public_key)
#User message example
message = b"Hello. Bob. Please receive this secret message secretly."
#encrypt alice msg with aes
nonce, ciphertext, tag = encrypt_message(message, sym_key)

#Bob
#Use Bob private key to decrypt sym key
decrypted_sym_key = decrypt_symmetric_key(encrypted_sym_key, alice_private_key)

#decrypt alice msg to bob with aes
decrypted_message = decrypt_message(nonce, ciphertext, tag, decrypted_sym_key)

print("Original message:", message)
print("Encrypted message:", ciphertext)
print("Decrypted message:", decrypted_message)
```

Test	Description	Result	Issue
User Authentication	Using Pyrebase	Success	But problems with pyrebase because of version incompatibility with module Collections
Hybrid encryption AES, RSA	Without socket programming	Success	
Hybrid encryption AES, RSA	With socket programming	Fail	Unpadding in AES library despite correct keys, ciphers, block size, ciphertext
Hybrid encryption AES, ECC	With socket programming and ecies	Success	
Hybrid encryption DES, RSA	With socket programming	Fail	Unpadding in DES
Key Message Passing: RSA, AES	Without socket programming	Success	
Hashing	With socket programming	Fail	Due to issues with AES and DES
Key Management of different files	With socket programming	Success	

Conclusion

1) How are the different modules integrated into the Secure Communication Suite?

They were integrated using 2 basic architectures, hybrid encryption which combines both symmetric and symmetric encryption, and key message passing which combines the possibility of either encrypting host AES key with client public key or with client private key.

2) What types of tests were conducted on the suite?

The tests conducted on this version are manual integration tests. Unit tests we conducted in previous modules. This document focuses on integration tests. All unit tests were successful.

3) How does the suite secure internet services?

It provides a means of communication using an anti-meet in the middle attack key exchange technique, given in the public key cryptosystems module in addition to reliable encryption and decryption given in the block cipher module. As for communication integrity, a one-way encryption salt in the hashing module is added to sent messages and compared with received messages. A key management module is also added to ensure confidentiality. Finally, each user has their own unique password hashes, thus guaranteeing authenticability.