MINI PROJECT FOR COMPUTER AND NETWORK SECURITY

PTS Encryption and Decryption Program

Submitted to

School of Information, Computer and Communication Technology
Sirindhorn International Institute of Technology
Thammasat University

2 May 2021

by

Paphana Yiwsiw 6222780379
Thanakit Lerttomolsakul 6222780569
Sirada Chaisawat 6222780668

Digital Engineering (DE) Curriculum
Semester 2 Academic Year 2021

Abstract

In this mini-project, we have implemented PTS (Paphana-Thanakit-Sirada) Encryption and Decryption Program. This system is similar to PGP encryption program which aims to protect messages on and end-to-end basis with focus on confidentiality and authentication of sender. In this system, we implemented both asymmetric cryptography using RSA public key encryption and symmetric cryptography using AES encryption algorithm. The length of 2 prime numbers used to generate key pairs in this system is 1024 bits. This system is demonstrated and implemented using Python programming language, and it is successfully implemented with confidentiality and authentication service simultaneously.

Contents

Chapter 1

Project Concept

In this project, we create the PTS (Paphana-Thanakit-Sirada) encryption program which is the implementation of PGP-like encryption system. The PTS encryption program provides the encryption service that focused on the confidentiality of the message and authentication of the sender.

1.1 Summary

In the program, we created the PGP-like service that used both asymmetric and symmetric encryption algorithm. The detail of the program is summarized as follows.

- Implementation of RSA algorithm as the asymmetric encryption algorithm for the program.
  - Public and private key generation from two 1024-bit prime numbers p and q.
  - RSA encryption and decryption from given input bits sequence.
- Implementation of AES from pycryptodome library as the symmetric encryption algorithm for the program.
- Implementation of SHA-1 hashing algorithm to generate the hash of the message that later used to create the digital signature of the sender.
- Using RSA and AES algorithm to provide encryption and decryption services for the message in the program for users.
  - The authentication of sender service which verify the received message actually send from the sender of the message.
  - The confidentiality of the message service which protect secrecy of the message and unauthorized access for those who are not the intended receiver.

1.2 Typical Usage

The PTS program will be used when the sender wants to send message to the receiver securely. At the receiver end, they can verify the authenticity of the message and ensure that the message will not be able to open and decrypt by unintended receiver.
Other key features in the delivered python program are as follows:

- Reading the message from and writing the encrypted message to corresponding files.
- Reading and writing the generated public and private key pair to the corresponding files for the user to use the key afterward.

1.3 Main Challenges

During the development of this mini-project, we encountered many challenges, including both technical and non-technical challenges. Major challenges are listed as follows:

- The first major challenge is the technical one which is the use of pycryptodome and hashlib library. We spent quite a lot of time on studying how to use both libraries in implementation of the encryption, which setback the project completion date.
- The second one is also about the implementation of the program based on what we learned in lectures. During the development, we encountered bugs which related to the leading zeros in conversion of a base number thus produce incorrect result, but this only occurs when the key size is small. Another problem is about the implementation of some algorithms given in class which used recursive method. It yields the recursive stack error when key size gets very large. Thus, we have to implement algorithm in the iterative version [2] which we will talk about that later on.
- The last challenge is on non-technical aspect which is the time management of this project. We have allocated the time for this mini-project short by a little. Other courses also have many assignments due at the same time which disabled us on working together on the project. However, we can still communicate to each other and deliver project on time.

Chapter 2

Project requirements

2.1 System description

The PTS (Paphana-Thanakit-Sirada) encryption program is the implementation of the PGP-like system which provide the authentication of the sender and confidentiality of the message. However, the PTS system will provide both services to the user simultaneously [1]. The steps of authentication and confidentiality are as follows:
To send and encrypt the message,

1. The sender creates the clear-text message. The sender can choose whether to use message as a text file (message.txt) or enter using the program prompt.

2. SHA-1 hash is generated from the clear-text message.

3. The digital signature is created using hash and encrypted by RSA algorithm using sender's private key. Then, append the digital signature to the message to create the signed message.

4. The session key is generated from random 128-bit (16-bytes) number, which will used for this session only.

5. The sender encrypts the signed message by AES symmetric encryption algorithm using the session key as AES encryption key, and produces encrypted signed message. The message tag is also generated from AES at this point.

6. The session key is later encrypted using RSA with the receiver's public key. The encrypted session key is appended to the encrypted signed message, which yields the black-text message.

7. The sender also create nonce for indicating the freshness to the intended receiver and, along with the message tag, sends with the black-text message. Noted that final black-text message is in the binary form and written into the designated binary file.

To receive and decrypt the message,

1. The receiver read the received black-text message from a binary file, and split it into 4 sections: nonce, message tag, encrypted signed message, and encrypted session key.

2. The receiver then needs to decrypt and retrieve the session key using RSA with the receiver's private key.

3. After retrieved session key, the receiver use it to decrypt signed message using AES algorithm. In this step, nonce and message tag are used to verify and validate the freshness of the session.

4. Now, the decrypted signed message is received. The receiver splits the received message and digital signature apart from each other. This is to authenticate the message that it is actually sends from sender.

5. The digital signature is then decrypted using RSA with the sender's public key which will reveal the hash of plaintext message sent from the sender.

6. The hash of received message is generated using SHA-1.

7. To verify the authentication of sender, the generated SHA-1 hash from the received message and the hash of plaintext decrypted from the digital signature are compared. If both of them are match, then this message is authentic. Therefore, they can read the recovered plaintext message with knowledge that no one has tempered and read the message between them.

2.2 Computational tasks

1. Public and private key pair generation

   (a) 1024-bit prime numbers (p, q) generation by using Crypto.Util module from pycryptodome library.

   (b) Euclidean algorithm for finding greatest common divisor (gcd) in generating number e that is relatively prime to $\phi(n),\ gcd(e, \phi(n)) = 1$.

   (c) Extended euclidean algorithm in multiplicative inverse which uses for determining d such that $d = e^{-1}\, mod\, \phi(n)$

2. AES symmetric encryption algorithm, using from pycryptodome function.

   (a) Generating 128-bit session key to use for encrypting signed message and decrypting encrypted signed message.

   (b) Generate nonce and message tag to indicate the freshness.

3. RSA encryption

   (a) Calculate the plain bit block size and divide the bit sequence to blocks for using in the encryption process.

   (b) Padding of one and zeros to the end of the bit sequence to ensure that the last block is the same size.

   (c) Conversion of binary to decimal and decimal to binary.

   (d) Modular exponentiation ($C = M^e\, mod\, n$) to encrypt the message using the binary representation for better efficiency.

4. RSA decryption

   (a) Calculate the cipher bit block size and divide the bit sequence to blocks for using in the decryption process.

   (b) Conversion of binary to decimal and decimal to binary.

   (c) Modular exponentiation ($M = C^d\, mod\, n$) to decrypt the message.

   (d) Removal of padded one and zeros at the end of the decrypted bit sequence.

5. Miscellaneous tasks

   (a) Conversion of bytes to binary sequence and binary sequence to bytes, using in the encryption of the session key.

2.3 Use cases

The PTS encryption program can be used when the sender wants to send the message to the specific receiver with that message being encrypted and not readable by unintended receiver. The encrypted message package should be able to tell the receiver whether this message is authentic or not and protect the confidentiality of the message.

Chapter 3

Algorithm design and Implementation

3.1 Algorithm design

In the PTS encryption program, we have used algorithms as follows:

- Euclidean algorithm for finding greatest common divisor (gcd)
- Modular exponentiation ($a^m \, mod \, n$) using the binary representation of the exponent
- Extended euclidean algorithm ($ax + by = gcd(a, b)$)
- Multiplicative inverse algorithm ($d = e^{-1} \, mod \, \phi(n)$)
- Public key and private key pair generation
- RSA encryption and decryption

However, some of this algorithm may differ from what we learned in the class because it has impact on the performance of the program. Especially, when using the recursive function, we have change to use the iterative version of algorithms instead due to the recursive stack overflow and throw an exception to the program.

3.1.1 Euclidean algorithm

Algorithm 1: Euclidean algorithm

Data: a, b
Result: $gcd(a, b)$
while $b \neq 0$ **do**
  $(a,\ b) \leftarrow (b,\ a \, mod \, b)$
end
return a

3.1.2 Modular exponentiation

Algorithm 2: Modular exponentiation; effModuloExp(a,m,n)

Data: a, m, n as in $a^m \, mod \, n$
Result: $a^m \, mod \, n$
$d \leftarrow 1$;
$b \leftarrow$ binary representation of m;
$k \leftarrow$ length of b;
for $i \leftarrow k$ **downto** 0 **do**
    $d \leftarrow (d \times d) \, mod \, n$;
    if $b_i \neq 0$ **then**
        $d \leftarrow (d \times a) \, mod \, n$;
    end
end
return d

3.1.3 Extended euclidean algorithm

Algorithm 3: Extended euclidean algorithm; extEuclidean(a,b)

Data: a, b
Result: returns a triple (c, x, y) such that $ax + by = c$ and $c = gcd(a, b)$
$r0, r1 \leftarrow a, b$;
$s0, s1 \leftarrow 1, 0$;
$t0, t1 \leftarrow 0, 1$;
while $r1 \neq 0$ **do**
    $q \leftarrow r0 \; div \; r1$;
    $r0, r1 \leftarrow r1, r0 - (q \times r1)$;
    $s0, t1 \leftarrow s1, s0 - (q \times s1)$;
    $t0, t1 \leftarrow t1, t0 - (q \times t1)$;
end
return $(r0, s0, t0)$

3.1.4 Multiplicative inverse

Algorithm 4: Multiplicative inverse; $mul\_inverse(e, \phi(n))$

Data: e such that $gcd(e, \phi(n)) = 1$
Result: returns $e^{-1} \; mod \; \phi(n)$
$c, x, y \leftarrow ExtEuclidean(e, \phi(n))$;
return x

3.1.5 Key generation

Algorithm 5: Public key generation; $get\_public\_key(p, q)$

Data: prime number p and q of size 1024 bits

Result: public key (e, n)

$phi \leftarrow (p-1) \times (q-1)$;

$e \leftarrow$ random integer between 1 and phi;

while $gcd(e, phi) \neq 1$ **do**

  $e \leftarrow$ random integer between 1 and phi;

end

$n \leftarrow p \times q$;

return (e, n)

Algorithm 6: Private key generation; $get\_private\_key(p, q, e)$

Data: prime number p and q of size 1024 bits, e from public key

Result: private key (d, n)

$phi \leftarrow (p-1) \times (q-1)$;

$d \leftarrow mul\_inverse(e, phi)$;

return (d, n)

3.1.6 RSA encryption and decryption

Algorithm 7: RSA encryption; $encrypt(plainBitSeq, key)$

Data: binary sequence $plainBitSeq$, encryption key as (e, n)

Result: cipher binary sequence $cipherBitSeq$

$plainBlockSize \leftarrow \lfloor \log_2(n) \rfloor$;

if $len(plainBitSeq)\ mod\ plainBlockSize \neq 0$, pad the one and zeros sequence (ex. $'100..0'$) at the end of $plainBitSeq$;

split $plainBitSeq$ into block of size $plainBlockSize$ and put into $plainBlockBin$;

for each block in $plainBlockBin$, convert from binary to decimal number to get $plainBlockDec$;

encrypt each block in $plainBlockDec$ using $effModuloExp(M, e, n)(C = M^e\ mod\ n)$ to get $cipherBlockDec$;

for each block in $cipherBlockDec$, convert from decimal to $(plainBlockSize + 1)$-bit binary, to get $cipherBlockBin$;

combine all blocks in $cipherBlockBin$ into one $cipherBitSeq$ binary sequence;

return $cipherBitSeq$

Algorithm 8: RSA decryption; $decrypt(cipherBitSeq, key)$

Data: binary sequence $cipherBitSeq$, decryption key as (d, n)

Result: plaintext binary sequence $plainBitSeq$

$cipherBlockSize \leftarrow \lfloor \log_2(n) \rfloor + 1$;

split $cipherBitSeq$ into block of size $cipherBlockSize$ and put into $cipherBlockBin$;

for each block in $cipherBlockBin$, convert from binary to decimal number to get $cipherBlockDec$;

decrypt each block in $cipherBlockDec$ using $effModuloExp(C, d, n)(M = C^d \ mod \ n)$ to get $plainBlockDec$;

for each block in $plainBlockDec$, convert from decimal to $(cipherBlockSize - 1)$-bit binary, to get $plainBlockBin$;

combine all blocks in $plainBlockBin$ into one $plainBitSeq$ binary sequence;

remove one and zeros padding at the end of $plainBitSeq$ binary sequence if any;

return $plainBitSeq$

3.2 Implementation

The PTS encryption and decryption program is implemented by following these pseudo codes:

Algorithm 9: PTS encryption

Data: message.txt file as the message, private key of sender, public key of receiver

Result: encrypted message binary encrypted_message.bin file

`E00` Read clean-text message from message.txt;

`E10` Create SHA-1 hash of the clean-text message;

`E20` Convert hash into binary sequence;

`E30` Create the digital signature of the message by encrypting hash using RSA with private key of the sender.;

`E40` Append the digital signature to create the signed message.;

`E50` Generate random 128-bit session shared secret key (SSSK);

`E60` Encrypt the signed message using AES encryption with SSSK as a key.;

`E70` Encrypt the SSSK using RSA with public key of the receiver.;

`E80` Append encrypted SSSK to the signed message.;

`E90` Create nonce which used to provide receiver the freshness.;

`E100` Concatenate nonce, tag, encrypted signed message, encrypted SSSK, to single binary sequence, write to *encrypted_message.bin* and send to the receiver.

return *encrypted_message.bin*

Algorithm 10: PTS decryption

Data: encrypted message binary encrypted_message.bin file

Result: decrypted message text file decrypted_message.txt file

`D00` Read black-text message from encrypted_message.bin;

`D10` Split black-text message into nonce, tag, encrypted signed message, encrypted SSSK, respectively.;

`D20` Decrypt the SSSK using RSA with the private key of the receiver.;

`D30` Decrypt the signed message using AES with the decrypted SSSK, use nonce and tag to validate and verify the freshness of the message. If it passes, then yields decrypted signed message. Otherwise, reject the message.;

`D40` Split the decrypted signed message into retrieved message and retrieved digital signature.;

`D50` Decrypt the retrieved digital signature with RSA with public key of the sender, yields the decrypted message hash.;

`D60` Create SHA-1 hash of the retrieved message, yields the generated message hash;

`D70` If the decrypted message hash (Step D50) and the generated message hash (Step D60) match, accept the message as authentic and write message to decrypted_message.txt. Otherwise, Reject the message;

return *decrypted_message.txt*

References

[1] @tsp2121999. *PGP - Authentication and Confidentiality: GeeksforGeeks*. URL: https://www.geeksforgeeks.org/pgp-authentication-and-confidentiality/ (visited on 04/30/2022).

[2] Aliyev Silap and Mariia Mykhailova. *Extended Euclidean Algorithm - Algorithm for Competitive Programming*. URL: https://cp-algorithms.com/algebra/extended-euclid-algorithm.html#iterative-version (visited on 04/30/2022).

Appendix A

Utility Function

In the utils.py file, the implementation of euclidean algorithm, modular exponentiation, iterative version of extended euclidean algorithm, multiplicative inverse are shown.

```
import random

def hex_to_bin(hex_str):
    binStr = bin(int(hex_str, 16))[2:]
    return binStr

def bin_to_hex(bin_str):
    return hex(int(bin_str, 2))[2:]

def dec_to_bin(dec_str):
    return bin(int(dec_str))[2:]

def bin_to_dec(bin_str):
    return int(bin_str, 2)

# All algorithms given in the mini-project requirement

# 0 Euclidean GCD Algorithm
def gcd(a, b):
    while b != 0:
        (a, b) = (b, a % b)
    return a

# 1 Molular Exponentiation to calculate a^m mod n
def effModuloExp(a, m, n):
    d = 1
    b = dec_to_bin(m)
    k = len(b)
    for i in range(k, 0, -1):
        d = (d * d) % n
        if int(b[0-i]) != 0:
```

```python
            d = (d * a) % n
    return d

# 4/1 extended euclidean algorithm recursive version
def extEuclidean_rec(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        c, x, y = extEuclidean_rec(b, a % b)
        return (c, y, x - (a // b) * y)

# 4/2 extended euclidean algorithm iterative version
def extEuclidean(a, b):
    r0, r1 = a, b
    s0, s1 = 1, 0
    t0, t1 = 0, 1
    while r1 != 0:
        q = r0 // r1
        r0, r1 = r1, r0 - q * r1
        s0, s1 = s1, s0 - q * s1
        t0, t1 = t1, t0 - q * t1
    return (r0, s0, t0)

# 5 Multiplicative Inverse of e modulo phi(p*q) without x being negative
def mul_inverse(e, phi):
    d = extEuclidean(e, phi)[1]
    if d < 0:
        d += phi
    return d

# 6 check if a number is prime
def is_prime(n):
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

# 7.1 convert bytes to binary
def bytes_to_bin(bytes_str):
    return ''.join(format(b, '08b') for b in bytes_str)

# 7.2 convert binary to bytes
def bin_to_bytes(bin_str):
    return bytes([int(bin_str[i:i+8], 2) for i in range(0, len(bin_str),
        8)])
```

Appendix B

Key Generation

In the keyGen.py file, the implementation of public and private key pair generation is shown.

```python
# Public Key Generation
from Crypto.Util import number
import utils
import random

def get_prime(bits):
    return number.getPrime(bits)

def get_prime_pair(bits):
    p = get_prime(bits)
    q = get_prime(bits)
    while (p == q):
        q = get_prime(bits)
    return (p, q)

def get_n(p, q):
    return p * q

def get_phi(p, q):
    return (p-1) * (q-1)

def get_e(p, q):
    phi = get_phi(p, q)
    e = random.randint(2, phi)
    while utils.gcd(e, phi) != 1:
        e = random.randint(2, phi)
    return e

def get_d(p, q, e):
    phi = get_phi(p, q)
    d = utils.mul_inverse(e, phi)
```

```python
    return d

def get_public_key(p, q):
    e = get_e(p, q)
    n = get_n(p, q)
    return (e, n)

def get_private_key(p, q, e):
    d = get_d(p, q, e)
    n = get_n(p, q)
    return (d, n)

def generate_key_pair(bits):
    p, q = get_prime_pair(bits)
    e, n = get_public_key(p, q)
    d, _n = get_private_key(p, q, e)
    if (e == None) or (d == None) or (n != _n) or (d <= 1):
        return generate_key_pair(bits)
    # avoid key collision
    while (d == e):
        d, _n = get_private_key(p, q, e)
    return (e, n), (d, n)
```

Appendix C

RSA Algorithm

In the RSA.py file, the implementation of RSA encryption and decryption algorithm is shown.

```
# RSA Encryption and Decryption
import utils
import hashlib
from math import floor, log

# divide blocks into blocks of length k
def plainBlock(plainSequence,size):
    # pad one and zero to the end of the sequence if necessary
    if len(plainSequence) % size != 0:
        plainSequence += '1'
        while len(plainSequence) % size != 0:
            plainSequence += '0'
    plainBlocks = []
    for i in range(0, len(plainSequence), size):
        plainBlocks.append(plainSequence[i:i+size])
    return plainBlocks

def encrypt(bit_seq, key):
    print("\tStarting RSA Encryption...")
    e, n = key
    plainBlockSize = floor(log(n, 2))
    plainBlockSeq = plainBlock(bit_seq, plainBlockSize)
    plainBlockDec = []
    for block in plainBlockSeq:
        plainBlockDec.append(int(block, 2))
    cipherBlockSeq = []
    for block in plainBlockDec:
        cipherBlockSeq.append(utils.effModuloExp(block, e, n))
    cipherBlockBin = []
    for block in cipherBlockSeq:
        blockBin = utils.dec_to_bin(block)
```

```python
        if len(blockBin) != plainBlockSize+1:
            blockBin = ('0'*(plainBlockSize-len(blockBin)+1)) + blockBin
        cipherBlockBin.append(blockBin)
    cipherText = ""
    for block in cipherBlockBin:
        cipherText += block
    print("\tEnding RSA Encryption...")
    return cipherText

def decrypt(cipherText, key):
    print("\tStarting RSA Decryption...")
    d, n = key
    cipherBlockSize = floor(log(n, 2)) + 1
    cipherBlock = [ cipherText[i:i+cipherBlockSize] for i in range(0,len
       (cipherText),cipherBlockSize)]
    cipherBlockDec = []
    for block in cipherBlock:
        cipherBlockDec.append(utils.bin_to_dec(block))
    plainBlockDec = []
    for block in cipherBlockDec:
        plainBlockDec.append(utils.effModuloExp(block, d, n))
    plainBlockBin = []
    for block in plainBlockDec:
        blockBin = utils.dec_to_bin(block)
        if len(blockBin) != cipherBlockSize - 1:
            blockBin = ('0'*((cipherBlockSize - 1) -len(blockBin))) +
                blockBin
        plainBlockBin.append(blockBin)
    plainTextPad = ""
    for block in plainBlockBin:
        plainTextPad += block
    idxLastOne = len(plainTextPad) - 1
    while plainTextPad[idxLastOne] == '0':
        idxLastOne -= 1
    plainText = plainTextPad[:idxLastOne]
    print("\tEnding RSA Decryption...")
    return plainText
```

Appendix D

PTS Main Program

In the main.py file, the implementation of PTS (Paphana-Thanakit-Sirada) program is shown. The result from this program will also contain the steps description on the command line screen as well. The submission code version may be slightly different from this report due to formatting into the report.

```
import utils
import keyGen
import RSA
import hashlib
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

print("----------------------- START OF PROGRAM ---------------")

NUMBER_OF_BITS = 1024

while True:
    choice = input("Do you want to generate new key pair or use existing key pair? (y/n) : ")
    choice = choice.lower()

    if choice == 'y':
        print("Generating public and private key of sender (A) and receiver (B)...")
        PU_A , PR_A = keyGen.generate_key_pair(NUMBER_OF_BITS)
        PU_B , PR_B = keyGen.generate_key_pair(NUMBER_OF_BITS)
        print("Public key and private key of sender (A) and receiver (B) are generated.\n")

        write_choice = input("Do you want to write public key of sender (A) and receiver (B) to file? (y/n) : ")
        write_choice = write_choice.lower()
        if write_choice == 'y':
```

```
            print("Writing public key and private key of sender (A) and receiver (B) to file...")
            try:
                with open('PUK_A.txt', 'w') as f:
                    f.write(str(PU_A[0]) + '\n')
                    f.write(str(PU_A[1]))
                with open('PUK_B.txt', 'w') as f:
                    f.write(str(PU_B[0]) + '\n')
                    f.write(str(PU_B[1]))
                with open('PRK_A.txt', 'w') as f:
                    f.write(str(PR_A[0]) + '\n')
                    f.write(str(PR_A[1]))
                with open('PRK_B.txt', 'w') as f:
                    f.write(str(PR_B[0]) + '\n')
                    f.write(str(PR_B[1]))
                print("Public key and private key of sender (A) and receiver (B) are written to file.\n")
            except Exception as e:
                print("Error cannot write key pairs to files.\n"+str(e))
        else:
            print("Public key and private key of sender (A) and receiver (B) are not written to file.\n")
        break

    elif choice == 'n':
        print("Reading public and private key of sender (A) and receiver (B) from file...")
        try:
            with open('PUK_A.txt', 'r') as f:
                puk_A = f.readlines()
            PU_A = (int(puk_A[0]), int(puk_A[1]))
            with open('PUK_B.txt', 'r') as f:
                puk_B = f.readlines()
            PU_B = (int(puk_B[0]), int(puk_B[1]))
            with open('PRK_A.txt', 'r') as f:
                prk_A = f.readlines()
            PR_A = (int(prk_A[0]), int(prk_A[1]))
            with open('PRK_B.txt', 'r') as f:
                prk_B = f.readlines()
            PR_B = (int(prk_B[0]), int(prk_B[1]))
            print("Public key and private key of sender (A) and receiver (B) are read from file.\n")
            break
        except Exception as e:
            print("Error: " + str(e))
            print("Please generate new key pair or read key pair from file.\n")
```

```
            continue
    else:
        print("Invalid choice: Please enter y or n.")

while True:
    message_choice = input("Do you want to read message from message.txt
         file or enter message manually?(y/n)  : ")
    message_choice = message_choice.lower()
    if message_choice == 'y':
        try:
            input_message = ""
            with open('message.txt', 'r') as f:
                message_file = f.readlines()
            for msg in message_file:
                input_message += msg + '\n'
            print("Message is read from file.\n")
            break
        except Exception as e:
            print("Error: " + str(e))
            print("Please enter message manually.\n")
            continue
    elif message_choice == 'n':
        input_message = input("Enter message: ")
        break
    else:
        print("Invalid choice: Please enter y or n.\n")

orginal_message = input_message
orginal_message += '\n'
orginal_message.encode('utf-8')

# Step Notation:
# Exy or Exxy - Encryption      Dxy or Dxxy - Decryption
#  x  or  xx  - Step number     (E1x - E10x / D1x - D7x)
#          y  - Substep number  (0 indicates no substep)

# Encryption
print("Start Encryption Process ----------------------\n")

# E10 create hash of message
hash_original_message = hashlib.sha1(orginal_message.encode('utf-8')).
    hexdigest()
print("E10 create hash of message")

# E20 convert hash to binary
hash_original_message_binary = utils.hex_to_bin(hash_original_message)
print("E20 convert hash to binary")
```

```python

# E30 create digital signature of message using private key of sender
print("E30 create digital signature of message using private key of sender")
signature_binary = RSA.encrypt(hash_original_message_binary, PR_A)
signature_hex = utils.bin_to_hex(signature_binary)

# E40 add digital signature to message
message_with_signature = orginal_message + signature_hex
print("E40 add digital signature to message")

# E50 generate random 128-bit AES key
key = get_random_bytes(16)
print("E50 generate random 128-bit AES key")

# E60 encrypt message with signature using AES key
print("E60 encrypt message with signature using AES key")
print("\tStarting AES Encryption...")
cipher = AES.new(key, AES.MODE_EAX)
ciphertext, tag = cipher.encrypt_and_digest(message_with_signature.encode('utf-8'))
print("\tEnding AES Encryption...")

# E70 encrypt AES key with public key of receiver
print("E70 encrypt AES key with public key of receiver")
# --E71 convert AES key to binary
key_binary = utils.bytes_to_bin(key)
print("--E71 convert AES key to binary")
# --E72 encrypt AES key with public key of receiver
print("--E72 encrypt AES key with public key of receiver")
key_encrypted_binary = RSA.encrypt(key_binary, PU_B)

# E80 append encrypted AES key to encrypted message
print("E80 append encrypted AES key to encrypted message")
# --E81 convert ciphertext to binary
ciphertext_binary = utils.bytes_to_bin(ciphertext)
print("--E81 convert ciphertext to binary")
# --E82 append encrypted AES key to encrypted message
ciphertext_with_key_binary = ciphertext_binary + "\n" + key_encrypted_binary
print("--E82 append encrypted AES key to encrypted message")

# E90 create nonce
nonce = cipher.nonce
print("E90 create nonce")

# E100 write encrypted message to encrypted_message.bin
```

```
print("E100 write encrypted message to encrypted_message.bin")
# --E101 convert nonce from bytes to binary
nonce_binary = utils.bytes_to_bin(nonce)
print("--E101 convert nonce from bytes to binary")
# --E102 convert tag from bytes to binary
tag_binary = utils.bytes_to_bin(tag)
print("--E102 convert tag from bytes to binary")
# --E103 append encrypted message to nonce and tag
ciphertext_with_key_nonce_tag_binary = nonce_binary + "\n" + tag_binary
    + "\n" + ciphertext_with_key_binary
print("--E103 append encrypted message to nonce and tag")
# --E104 write encrypted message to encrypted_message.bin
# in encrypted_message.bin has 4 binary lines as follows:
# line 1: nonce                  line 2: tag
# line 3: ciphertext             line 4: shared session secret key (SSSK)
print("--E104 write encrypted message to encrypted_message.bin")
try:
    with open("encrypted_message.bin", "w") as f:
        f.write(ciphertext_with_key_nonce_tag_binary)
    print("Encrypted message successfully written to encrypted_message.
        bin")
except Exception as e:
    print("Error writing encrypted message to encrypted_message.bin")
    print(e)
print()
print("End Encryption Process --------------------------")
print()

# Decryption
print("Start Decryption Process ----------------------\n")

# D10 read encrypted message from encrypted_message.bin
print("D10 read encrypted message from encrypted_message.bin")
try:
    with open("encrypted_message.bin", "r") as f:
        received_encrypted_message = f.read()
    print("Encrypted message successfully read from encrypted_message.
        bin")
except Exception as e:
    print("Error reading encrypted message from encrypted_message.bin")
    print(e)
    exit(0)

# D20 split encrypted message into nonce, tag, and encrypted message
received_encrypted_message_split = received_encrypted_message.split("\n"
    )
```

```
print("D20 split encrypted message into nonce, tag, and encrypted
    message")

# D30 convert nonce and tag from binary to bytes
received_nonce_bytes = utils.bin_to_bytes(
    received_encrypted_message_split[0])
received_tag_bytes = utils.bin_to_bytes(received_encrypted_message_split
    [1])
print("D30 convert nonce and tag from binary to bytes")

# D40 decrypt AES key with private key of receiver
print("D40 decrypt AES key with private key of receiver")
# --D41 decrypt key using private key of receiver
print("--D41 decrypt key using private key of receiver")
decrypted_AESkey_binary = RSA.decrypt(received_encrypted_message_split
    [3], PR_B)
# --D42 convert decrypted binary key to bytes
decrypted_AESkey_bytes = utils.bin_to_bytes(decrypted_AESkey_binary)
print("--D42 convert decrypted binary key to bytes")

# D50 decrypt message with nonce and tag using AES key
print("D50 decrypt message with nonce and tag using AES key")
print("\tStarting AES Decryption...")
cipher = AES.new(decrypted_AESkey_bytes, AES.MODE_EAX, nonce=
    received_nonce_bytes)
received_message_bytes = utils.bin_to_bytes(
    received_encrypted_message_split[2])
decrypted_message_bytes = (cipher.decrypt_and_verify(
    received_message_bytes, received_tag_bytes)).decode('utf-8')
print("\tEnding AES Decryption...")

# D60 split decrypted message into original message and digital
    signature
received_decrypted_signature = decrypted_message_bytes.split("\n")[-1]
received_message = decrypted_message_bytes.replace("\n"+
    received_decrypted_signature, "")
received_message_split = [received_message, received_decrypted_signature
    ]
# append new line to received_message
received_message_split[0] = received_message_split[0] + "\n"
print("D60 split decrypted message into original message and digital
    signature")

# D70 verify digital signature of original message
print("D70 verify digital signature of original message")
# --D71 create SHA-1 hash of received message
```

```python
received_message_hash = hashlib.sha1(received_message_split[0].encode('
    utf-8')).hexdigest()
print("--D71 create SHA-1 hash of received message")
# --D72 convert received digital signature to binary
received_digital_signature_binary = utils.hex_to_bin(
    received_message_split[1])
print("--D72 convert received digital signature to binary")
# --D73 decrypt digital signature with public key of sender
print("--D73 decrypt digital signature with public key of sender")
received_digital_signature_decrypted_binary = RSA.decrypt(
    received_digital_signature_binary, PU_A)
# --D74 convert decrypted digital signature to hex
received_digital_signature_decrypted_hex = utils.bin_to_hex(
    received_digital_signature_decrypted_binary)
print("--D74 convert decrypted digital signature to hex")
# --D75 compare received digital signature with hash of received message
print("--D75 compare received digital signature with hash of received
    message\n")

print("Received digital signature: ",
    received_digital_signature_decrypted_hex)
print("  Hash of received message: ", received_message_hash)
print()
if received_digital_signature_decrypted_hex == received_message_hash:
    print("Sender digital signature verified.")
    print("Authentication successful!")
    print("\nDecrypted message ------------------\n")
    lines = [ line for line in received_message_split[0].split("\n\n")]
    for line in lines:
        print(line)
    print("--------------------------------------\n")
    while True:
        output_choice = input("Would you like to save the decrypted
            message to a file? (y/n): ")
        if output_choice == "y":
            print("Saving decrypted message to decrypted_message.txt")
            try:
                with open("decrypted_message.txt", "w") as f:
                    for line in lines:
                        f.write(line + "\n")
                print("Decrypted message saved to decrypted_message.txt"
                    )
            except Exception as e:
                print("Error saving decrypted message to
                    decrypted_message.txt")
                print(e)
            break
```

```
        elif output_choice == "n":
            print("Not saving decrypted message")
            break
        else:
            print("Invalid input. Please try again.")
else:
    print("Sender digital signature verification failed!")
    print("Authentication failed!")
print()

print("End Decryption Process ---------------------------------")
print("------------------------ END OF PROGRAM ----------------")
print("-------------------------- THANK  YOU ------------------")
```