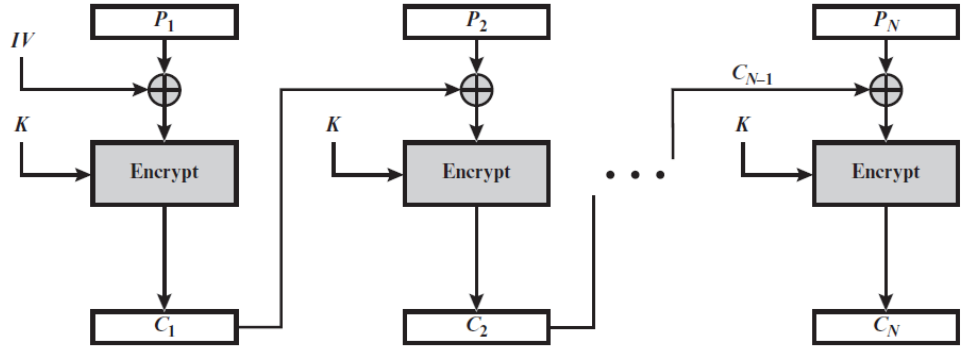
CSc8222 Assignment 1

Short Answer Questions:

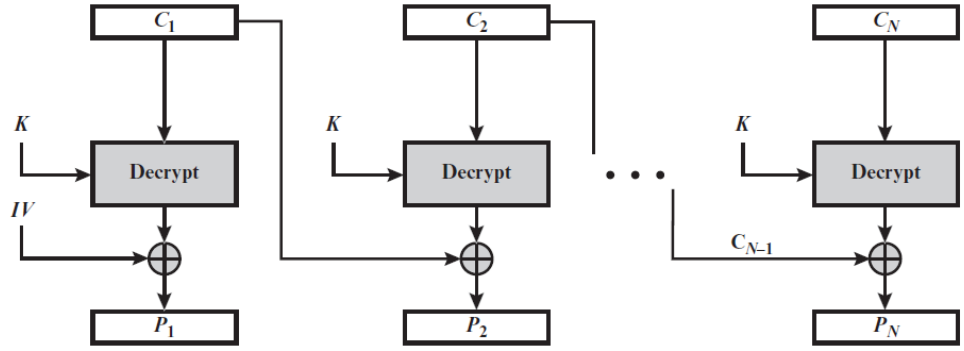
1. What is the difference between secret-key encryption and public-key encryption?
    a. Secret key encryption relies on the hiddenness of the keys used to encrypt and decrypt a message. Public-key encryption separates the responsibility of the encrypting and decrypting keys such that only the decrypting key needs to be kept private, while the key to encrypt messages intended for you can be publicized.

2. What is the difference between a block cipher and a stream cipher?
    a. Block cyphers operate on inputs which are broken down into blocks of fixed length, and each block is encrypted using a secret key. Block ciphers encrypt the data wholly in advance of transmission. Stream ciphers introduce key streams, which are streams of random numbers produced by an RNG seeded on the secret key. Each incoming block of data is encrypted with another token from the keystream. The intended receiver can generate the same key stream and decode the message block by block. It is suitable for applications where the total length of the transmitted document is not known in advance, such as for media streams. Block cyphers, on the other hand, require the fixed length data to be encrypted wholly before transmission.
3. What are the two general approaches to attacking a cipher?
    a. Repition attack and Insertion attack.
        i. In repition attacks, when two different messages are encrypted using the same keystream, if the attacker posseses these two encrypted streams, they can xor two corresponding elements of both streams and recover two plaintexts back. Since the corresponding messages are XOR'd with the same element of the key stream, XORing them together undoes their original XORs with the same key.
        ii. In insertion attack, the attacker has possession of encrypted messages and modifies some bytes and then retransmits them. If the modification he made happens to modify some important message content, in such a way that the decryption produces plausible plaintext, then upon decryption it could be interpreted as real and cause major damage.
4. With the ECB mode, if there is an error in a block of the transmitted ciphertext, only the corresponding plaintext block is affected. However, in the CBC mode, this error propagates. For example, an error in the transmitted C_1 (as the following figure of CBC) obviously corrupts P_1 and P_2.
    a. Are any blocks beyond P_2 affected?
        - Since the error is in the transmission of C1, decrypting C1 and C2 will produce erroneous plaintexts. But since C3 and so on were encrypted based on uncorrupted version of the previous ciphertexts, their decryption will not produce correct plaintexts, as we are using the corrupted C1 to decrypt C2, which itself becomes corrupted, and that corrupts C3, and so on. So all plain text after an error in transmission of ciphertext will be corrupted.
    b. Suppose that there is a bit of an error in the source version of P_1. Through how many ciphertext blocks is this error propagated? What is the effect at the receiver?
        - In this case there is an error at the input plaintext, and since all cipher texts depend on previous cipher texts, all subsequent cipher texts will be different than if

we had encrypted the uncorrupted data. At the receiving end, however, when they decode the message they will see an error in P1 as was given, but P2 and so on will not be affected.



(a) Encryption



(b) Decryption

5. Name three broad categories of applications of public-key cryptosystems.
    a. Authentication
    b. Confidentiality
    c. Integrity
6. Consider an automated cash deposit machine in which users provide a card or an account number to deposit cash. Give examples of confidentiality, integrity, and availability requirements associated with the system, and, in each case, indicate the degree of importance of the requirement.
    a. Confidentiality: Only the user and the bank should have plaintext access to user's account details. Actions taken on the machine, including type of request (view balance, make deposit, etc), the size of a deposit, should all be encrypted as well and knowable only to the user and the bank.
    b. Integrity: The account details should be correct, and requests generated by the use of the application should be received unmodified. When a user deposists an amount of cash, and a request is generated to update the account balance, it should not be modifyable in transport to change the amount deposited. Requests should also be timestamped to protect against replay attacks, ensuring only the user can take actions affecting their account.
    c. Availability: The service which updates accounts balances in response to deposits should always be available, and be resilient to DDOS attacks.

Requests sent from a deposit machine to the bank server should include details of their identification number, address, and deposit time, and any other metadata to identify the request during encryption. This will allow us to develop systems which discard requests which come too quickly from the same machine, and prevent DDOS attacks.

7. Find the multiplicative inverse of each nonzero element in Z_5.
   a. 1: 1, 2:3, 3:2, 4:4
8. In a public-key system using RSA, you intercept the ciphertext $C = 20$ sent to a user whose public key is $e = 13, n = 77$. What is the plaintext M?
   a. Using n = p*q; p = 7, q = 11. Phi(n) = 6 * 10 = 60. Given that e=13, find d such that e*d mod phi(n) = 1. Using extended Euclid gcd we find that d = 37.
   b. Decryption: M = C^d mod Phi(n); Given C = 20, d = 37, Phi(n) = 60, therefore **M = 20.**

9. In an RSA algorithm, the public key of a given user is $e = 65, n = 2881$. What is the private key of this user? Hint: First use trial-and-error to determine p and q; then use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo $\phi(n)$.
   a. Given n = 2881, using brute force, found p = 43, q = 67 => Phi(n) = 2772
   b. Find inverse mod Phi(n) of 65: **725**

10. a. Consider the following hash function. Messages are in the form of a sequence of numbers in Z_n, M = (a_1, a_2, … a_t). The hash value h is calculated as ($\sum_{i=1}^{t} a_i$) for some predefined value n. Does this hash function satisfy any of the requirements for a hash function listed in the following Table 11.1? Explain your answer.

    a. variable input size: true, sum can take any number of inputs

    b. fixed output size: false; only one number, the sum, but it is unbounded and thus has variable size.

    c. Efficiency: true, simply add up the values

    d. preimage resistant: true, to crack input requires testing all possible inputs

    e. weak collision resisant: false, given an x and H(x), it's simple to find y, such that H(y) = H(x). Simply subtract 1 from some x_i and add it to another x_j. The sum will be the same.

    e. collision resistant: false, many collisions that sum to same number, e.g. (1, 3) and (2, 2).

    b. Repeat part (a) for the hash function h=($\sum_{i=1}^{t}(a_i)^2$) mod n.

    a. variable input size: true, for same reason as above.

    b. fixed output size: true, it will be some residue modulo n.

    c. efficiency: true. It's linear w.r.t length of input.

    d. preimage resistance: true, attacked would need to test every possible variable length input string.

    e. weak collision resistance: false. Since the sum is being mod n, we need only to collide with the sum. Therefore we can make some modifications to the input such that the sum of squares remains the same, then we will collide.

    e. collision resistant: false, this function maps all inputs to one of N output numbers. Therefore if we try N+1 different inputs we will collide by pigeonhole principle.

c. Calculate the hash function of part (b) for M = (189, 632, 900, 722, 349) and n = 989.

(189^2 + 632^2 + 900^2 + 722 ^ 2 + 349 ^ 2) mod 989 = **229**

Table 11.1 Requirements for a Cryptographic Hash Function H

Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	$H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h, it is computationally infeasible to find y such that $H(y) = h$.
Second preimage resistant (weak collision resistant)	For any given block x, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) with $x \neq y$, such that $H(x) = H(y)$.

Programming Problem:

11. Design your system called Secure Messaging System with RSA. Implement a secure file transfer system that allows users to securely transfer files between two parties using RSA encryption. Your program should include the following functionalities at least:

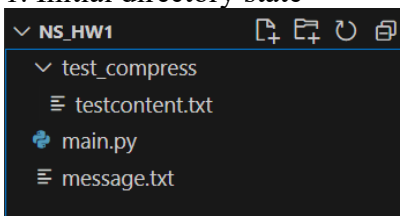
a. User Registration and Key Generation
b. File Compression
c. File Encryption
d. File Transfer
e. File Decryption

Top level menu of my program:

```
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation -- Generate public and private keys for a new
user
2. File Compression -- Compress a file into .zip format
3. File Decompression -- Decompress a .zip file
4. File Encryption -- Encrypt a file using your own public key
5. File Transfer -- Encrypt using the destination user's public key and send it to them
6. File Decryption -- Decrypt a file using your own private key
7. Switch User -- Log in as a different user
8. Show Detailed Menu
9. Exit
Enter your choice:
```

PROGRAM OUTPUT:

1. Initial directory state



2. **User registration and key generation**

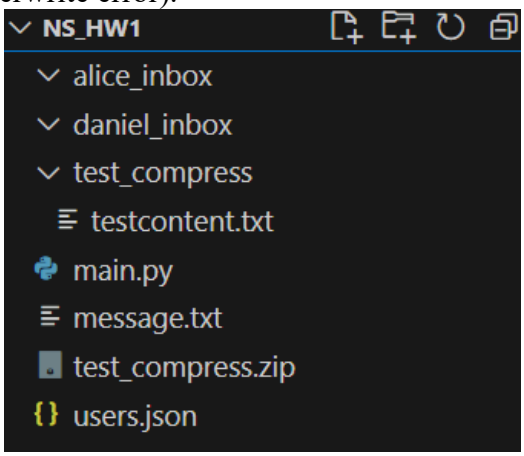
```
--Logging in to Secure Messaging System--
Enter username: daniel
User not registered
--User Registration and Key Generation--
Enter password: test
Generating key pair
User daniel registered successfully
--Logging in to Secure Messaging System--
Enter username: daniel
Enter password: test
Login successful
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 1.
--User Registration and Key Generation--
Enter username: alice
Enter password: test
Generating key pair
User alice registered successfully
```

Two new folders are created, “daniel_inbox” and “alice_inbox”, and two entries are created in users.json giving their public and private keys. My program stores both keys internally, but the private keys could easily be distributed as files instead.

3. **File compression**

```
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 2
--File Compression--
Enter directory name: test_compress
File compressed successfully as test_compress.zip
```

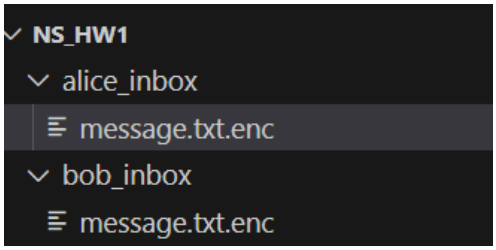
The test_compress.zip archive is now in the working directory. It can be decompressed using option 3, so long as the decompressed version isn't already in the target directory (overwrite error).



4. **File encryption and transfer**

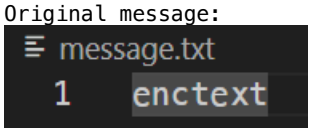
```
--Logging in to Secure Messaging System--
Enter username: daniel
Enter password: test
Login successful
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 1
--User Registration and Key Generation--
Enter username: bob
Enter password: test
Generating key pair
User bob registered successfully
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 5
--File Transfer--
Enter file name: message.txt
Logged in as:  daniel
Available Users:  ['daniel', 'alice', 'bob']
Enter destination user: alice
File transfer successful
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 5
--File Transfer--
Enter file name: message.txt
Logged in as:  daniel
Available Users:  ['daniel', 'alice', 'bob']
Enter destination user: bob
File transfer successful
```

Now both bob and alice have an encrypted message in their inbox.



Both of their encrypted copies are different from eachother.

Original message:



Alice’s encrypted copy:

```
1   3647375 24448638 1290042 43499310 3647375 19622699 43499310
```

Bob’s encrypted copy:

```
1   41659661 28053720 1378891 52643140 41659661 5435462 52643140
```

5. **File decryption**

```
--Secure Messaging System--
Logged in as daniel
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 7
--Logging in to Secure Messaging System--
Enter username: alice
Enter password: test
Login successful
--Secure Messaging System--
Logged in as alice
1. User Registration and Key Generation
2. File Compression
3. File Decompression
4. File Encryption
5. File Transfer
6. File Decryption
7. Switch User
8. Show Detailed Menu
9. Exit
Enter your choice: 6
--File Decryption--
alice's inbox:
['message.txt.enc']
Enter file name: message.txt.enc
100%|████████████████████████████████████████| 8/8 [07:27<00:00, 55.90s/it]
File decrypted successfully as alice_inbox/message.txt.enc.dec
```

Decrypting using alice’s private key reveals the original plaintext message. The same works for bob’s encrypted copy using bob’s secret key.