ISEC2000 - Fundamental Concepts Of Cryptography

Assignment 02

Assignment Report

By: Sauban Kidwai

ID: 20748199

Computer Labratory: (Building: 314.237 | Time: 3pm - 4pm | Day: Every Wednesday)

Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Kidwai	Student ID:	20748199	
Other name(s):	Sauban Mehmood			
Unit name:	Fundamental Concepts of Cryptography	Unit ID:	ISEC2000	
Lecturer / unit coordinator:	Dr. Nur Al Hasan Haldar	Tutor:	Chung Chan	
Date of submission:	May 19, 2024	Which assignment?	02	(Leave blank if the unit has only one assignment.)

Signature:	Sauban Mehmood Kidwai	Date of signature:	May 19, 2024

(By submitting this form, you indicate that you agree with all the above text.)

Table of contents

Question 1

Explain the principle of public key cryptography and how it differs from symmetric key cryptography. Also, describe the process of encryption in the RSA algorithm (10 Marks)

Public Key Cryptography (also known as Asymmetric Cryptography) uses two mathematically linked Keys: A Public key and a Private Key. The Public key can be freely shown anywhere since it is public however the corresponding Private key must be kept a secret. For Encryption, anyone is able to encrypt a message using the public key, but only the person with the corresponding private key can decrypt it. The private key can be used to create a digital signature for a message. This signature can be verified using the public key, ensuring the message originated from the holder of the private key and hasn't been tampered with.

In Symmetric Key Cryptography however, encryption and decryption is done by one single secure key that is shared between the sender and reciever. It is imperitive to securely share the key through secure methods otherwise anyone can decrypt a ciphertext using the symmetric key cryptography.

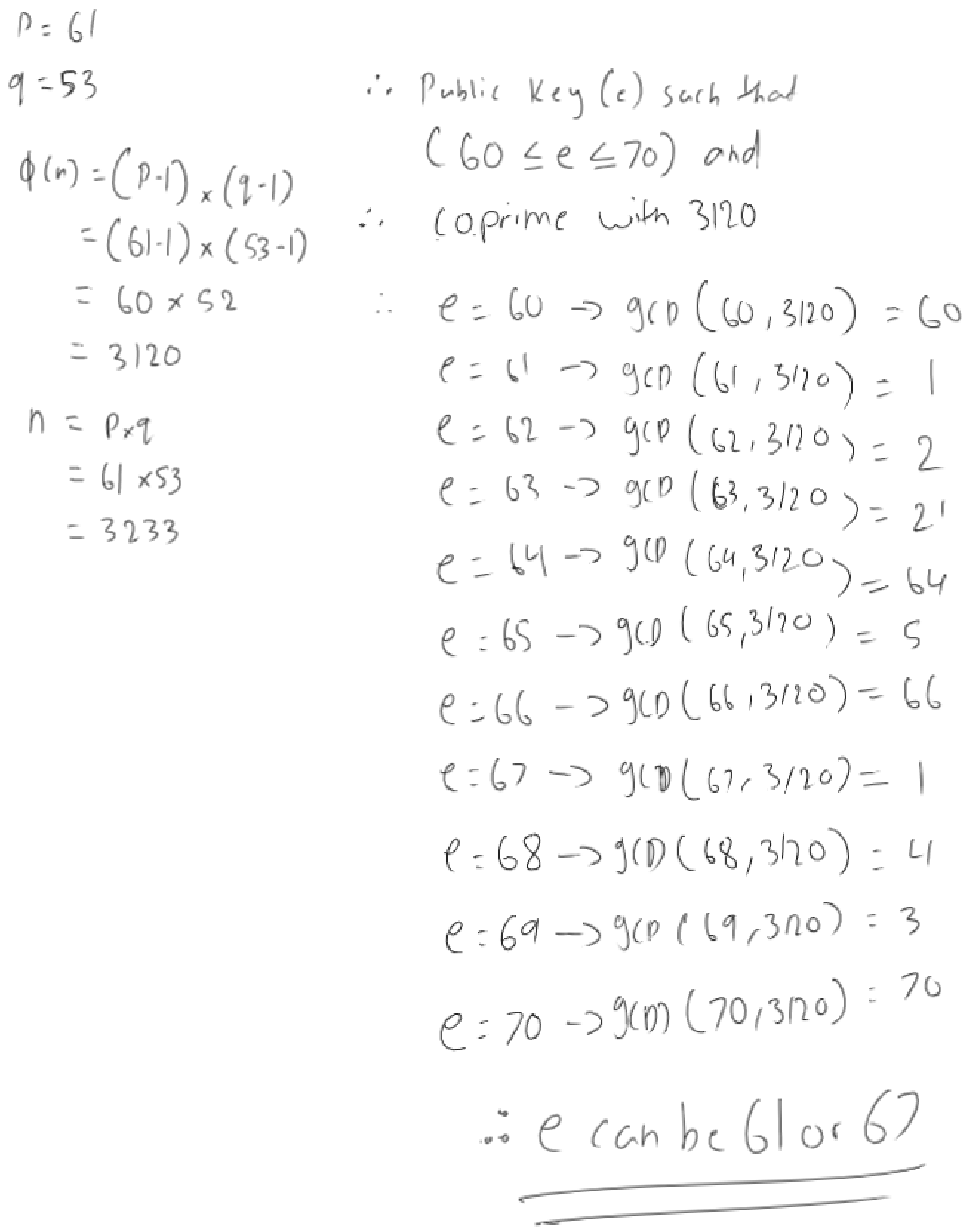
Public Key Cryptography (Asymmetric Cryptography) differs from Symmetric key Cryptography in 2 ways. The first is the key distribution where the public key eliminates the need for secure sharing because the key is broadcast and only the person with the private key can decrypt the message. The second way is that public key cryptography offers a digital signature whereas symmetric key cryptography does not.

The process of encryption in the RSA algorithm is as follows:

1. Key Generation:
    - A large number (modulus, n) is created by multiplying two large prime numbers (p and q).
    - Public key (e) and private key (d) are mathematically generated based on n and another value (ϕ(n)). The public key is made available, and the private key is kept secret.
2. Encryption:
    - The sender obtains the receiver's public key (e, n).
    - The sender converts the message (m) into a numerical format suitable for encryption (normally using padding techniques).
    - The sender then encrypts the message (m) using the public key: c = m^e (mod n)
        - where c is the ciphertext (encrypted message). Due to the mathematical properties of RSA, only the private key can decrypt this ciphertext.
3. Decryption:
    - The receiver then uses their private key (d, n) to decrypt the ciphertext (c): m = c^d (mod n).
    - This recovers the original message (m).

Question 2

Given two prime numbers p = 61 and q = 53, calculate the modulus n, Euler's totient function ϕ(n), and select an appropriate public key e, such that (60 ≤ e ≤ 70). Now, proceed to compute the private key d, assuming e = 17. (10 Marks)



$p = 61$

$q = 53$

$\phi(n) = (p-1) \times (q-1)$
$= (61-1) \times (53-1)$
$= 60 \times 52$
$= 3120$

$n = p \times q$
$= 61 \times 53$
$= 3233$

∴ Public Key (e) such that $(60 \leq e \leq 70)$ and

∴ coprime with 3120

∴ $e = 60 \rightarrow gcD(60, 3120) = 60$
$e = 61 \rightarrow gcD(61, 3120) = 1$
$e = 62 \rightarrow gcD(62, 3120) = 2$
$e = 63 \rightarrow gcD(63, 3120) = 21$
$e = 64 \rightarrow gcD(64, 3120) = 64$
$e = 65 \rightarrow gcD(65, 3120) = 5$
$e = 66 \rightarrow gcD(66, 3120) = 66$
$e = 67 \rightarrow gcD(67, 3120) = 1$
$e = 68 \rightarrow gcD(68, 3120) = 4$
$e = 69 \rightarrow gcD(69, 3120) = 3$
$e = 70 \rightarrow gcD(70, 3120) = 70$

∴ e can be 61 or 67

Therefore:

- n = 3233
- ϕ(n) = 3120
- e = can be either 61 or 67

NOTE: The Calculations for the GCD was done on a Scientific Calculator - Specifically the CASIO fx-82AU PLUS II Calculator.

Then to find out the private key we would first find the GCD until the remainder = 1 and then after that use the backwards substitution rule to then find the value of d which is the private key.

$17x = 1 \bmod 3120$

$3120 = \underline{17}(183) + \underline{9}$

$17 = \underline{9}(1) + \underline{8}$

$9 = 8(1) + 1$

$\therefore$ Private Key

$$\underline{\underline{d = 2753}}$$

$\therefore 1 = 9 - 8$

Sub $8 = 17 - 9$

$\therefore 1 = 9 - (17 - 9)$

$\therefore 1 = 9 - 17 + 9$

$1 = 9(2) - 17$

Sub $9 = 3120 - 17(183)$

$\therefore 1 = 2(3120 - 17(183)) - 17$

$1 = 2 \times 3120 - 2 \times 17 \times 183 - 17$

$1 = 2 \times 3120 - 366 \times 17 - 17$

$1 = 2 \times 3120 - 366 - 1 \times 17$

$1 = 2 \times 3120 - 367 \times 17$

$\therefore 17x = x = -367$

$= -367 + 3120$

$= 2753$

Therefore:

The private Key d is = 2753

Question 3

The Euclidean algorithm is based on the following assertion. Given two integers a, b, (a > b),

gcd(a, b) = gcd(b, a mod b). (1)

Prove the assertion (1) mathematically. (Note that proof by example is NOT appropriate here) (10 Marks)

Prove that $\gcd(a,b) = \gcd(b, a \bmod b)$

① By definition of mod function

$a = qb + (a \bmod b)$ for some integer q
where $q = a \text{ div } b$

② Let $d = \gcd(a,b)$

∴ means $d \mid a$ and $d \mid b$

∴ means $a = Kd$ and $b = Jd$ for some integer K and J

∴ Since $a = qb + (a \bmod b)$
Then $(a \bmod b) = a - qb$
$= Kd - qJd$
$= d(K - qJ)$

∴ Therefore we must have

$\boxed{d \le \gcd(b, a \bmod b)}$

③ Let $e = \gcd(b, a \bmod b)$

∴ means $e \mid b$ and $e \mid (a \bmod b)$

∴ means $b = me$ and $(a \bmod b) = ne$ For some integer m and n

∴ Since $a = qb + (a \bmod b)$
Then $a = qme + ne$
$= e(qm + n)$

∴ Therefore we must have

$\boxed{e \le \gcd(a,b)}$

Therefore $\gcd(a,b) = \gcd(b, a \bmod b)$

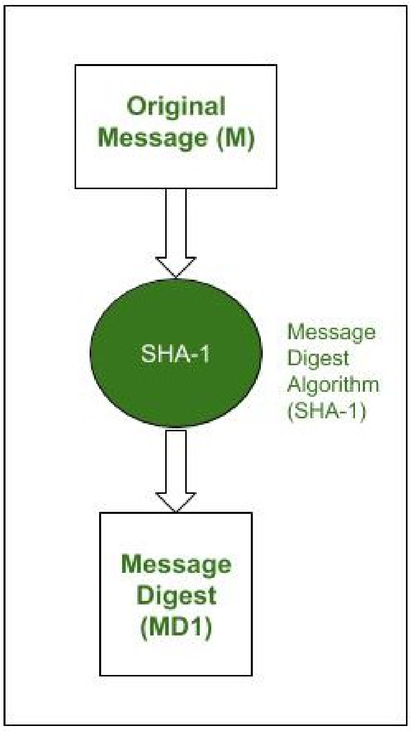
Question 4

Provide a diagram of the RSA signature structure and explain how it is used to authenticate a message. Discuss the steps involved when Alice signs a document m using her private key and how this signature is verified by Bob using Alice's public key. (10 marks)

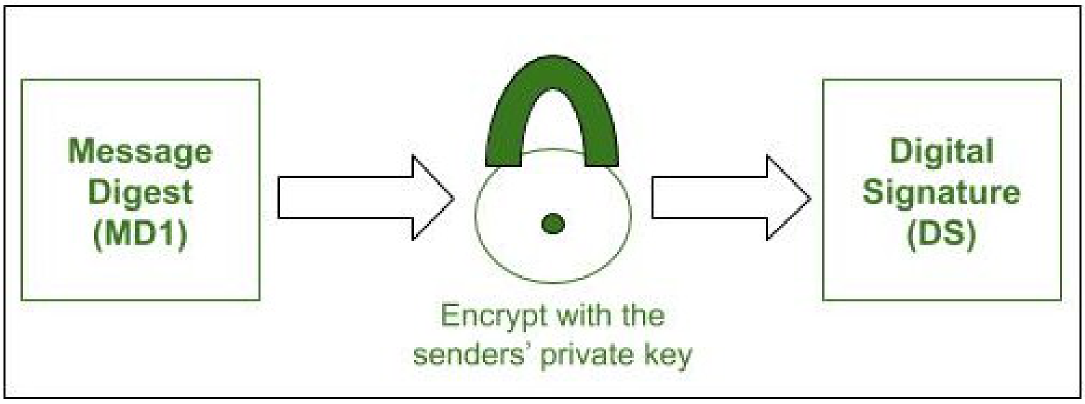
NOTE: The following image to aid the explanation is taken from reference 8 in the reference section of this report.

Digital signatures are used to verify the authenticity of the message sent electronically. The following steps explain how they work.

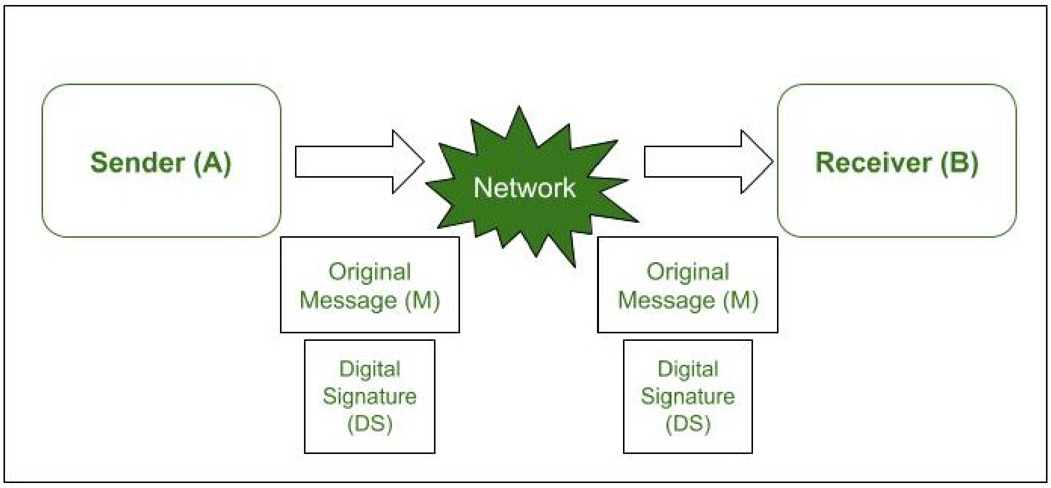
1. **Sender (A) creates a message digest (MD1):** A uses a hashing algorithm (like SHA-1) to generate a unique fingerprint (MD1) of the original message (M). This fingerprint is a fixed-size summary representing the message's content.



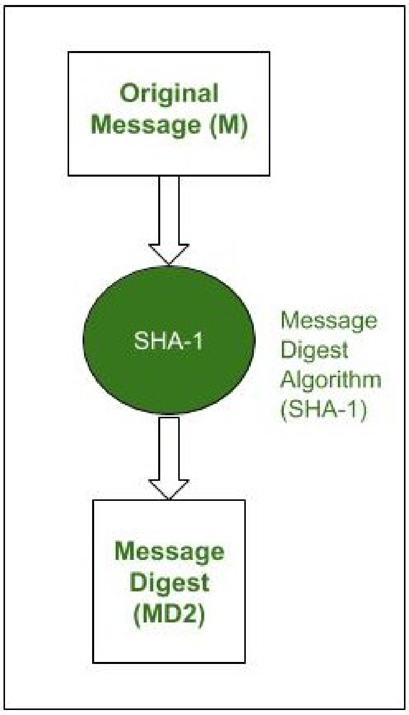
2. **Sender (A) creates a digital signature (DS):** A encrypts the message digest (MD1) using their private key. This encryption acts like a digital signature (DS), proving the message originated from them.



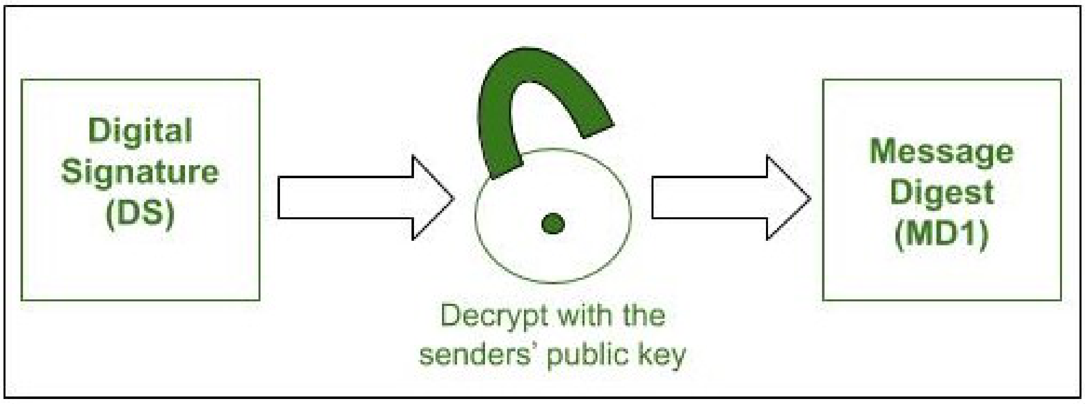
3. **Sender (A) transmits both message (M) and digital signature (DS):** A sends both the original message (M) and the digital signature (DS) to the receiver (B).



4. **Receiver (B) verifies the message:** Upon receiving them, B uses the same hashing algorithm to create their own message digest (MD2) of the received message (M).

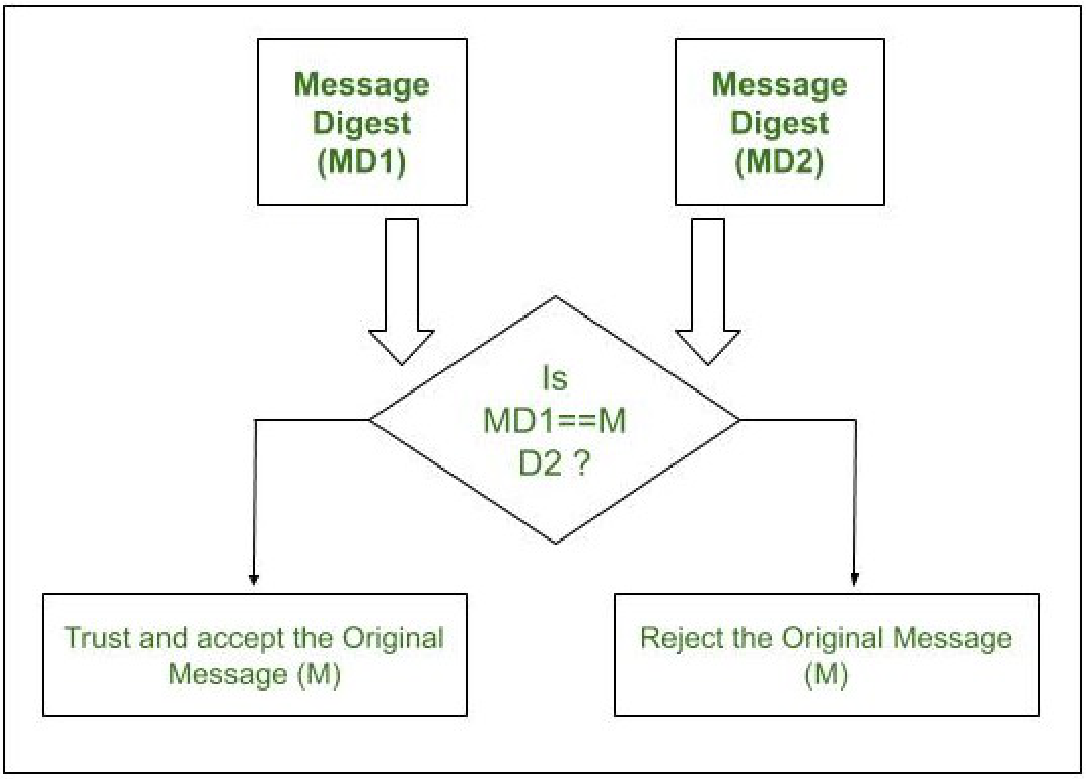


5. **Receiver (B) verifies the signature:** B then uses the sender's public key (which is publicly available) to decrypt the digital signature (DS). This decryption reveals the original message digest (MD1) created by the sender (A).



6. **Verification outcome:** If B's message digest (MD2) matches the sender's message digest (MD1) from the decrypted signature, it signifies success:
    - The message (M) is authentic and hasn't been modified.
    - The message genuinely originated from the sender (A) and not an impersonator.

This process ensures message integrity and authenticity through digital signatures.


Message Digest (MD1)
Message Digest (MD2)
Is MD1==MD2 ?
Trust and accept the Original Message (M)
Reject the Original Message (M)

These are the steps that would be involved for alice to sign a document using her private key:

1. Hashing:
    - Alice creates a hash (fingerprint) of the message (m) using a cryptographic hash function (e.g., SHA-256). This ensures a fixed-size output regardless of the original message length.
2. Padding:
    - Depending on the implementation, Alice could pad the hash value according to a specific scheme (e.g., PKCS#1) to ensure proper functioning with the RSA algorithm. Padding helps

prevent attacks that exploit the structure of the message.

3. Signing:
   - Alice uses her private key exponent (d) and the padded hash (h) to compute the signature (s): s = h^d (mod n) where n is the modulus (public key component). This calculation encrypts the hash with Alice's private key.
4. Transmission: Alice sends the message (m) and the signature (s) to Bob.
5. Verification: Bob receives the message (m) and the signature (s). He uses Alice's public key (e, n) to perform the following:
   - a. He recreates the hash (h') by applying the public key exponent (e) to the signature: h' = s^e (mod n)
   - b. He computes the hash of the received message (m') using the same hash function Alice used.
   - c. Bob compares the recreated hash (h') with the newly computed hash (m'). If they match, it indicates that the message originated from Alice and has not been tampered with during transmission.

Question 5

Assume that Alice has signed a document m using the RSA signature scheme, and the signature, along with the message, is sent to Bob. Describe a scenario where Bob discovers another message m' (where m ≠ m') such that H(m) = H(m'), with H() being the hash function used in the signature scheme.

1. *Describe, in detail, the steps Bob would take to forge Alice's signature for m' using the signature of m.*
2. *Discuss the cryptographic vulnerabilities exploited in this process and the potential impact on the authenticity of signed messages.* (15 Marks)

Answer to Part 1

There are 2 different steps that Bob can take to forge Alice's signature

1. Collision verification
   - Since Bob has two messages: the original message (m) signed by Alice and for which he has the signature (s), and the new message (m') for which he wants to forge a signature, he would need to verify if these messages collide under the hash function used in the signature scheme (H()). Bob would achieve this by:
     - Computing the hash of the new message (m'): h' = H(m').
     - Comparing h' with the hash of the original message (m) that Alice signed: H(m).
     - If h' = H(m), Bob has the necessary condition for forgery. This collision would mean that the same signature (s) can be used for both messages as they produce the same hash output.
2. Signature Reuse
   - If the hash collision is confirmed (h' = H(m)), Bob can then directly reuse the original signature (s) received from Alice. This signature (s) was created using Alice's private key and the hash of the original message (H(m)). But because of the collision, it also can act as a valid signature for the new message (m') because H(m') = H(m).

Answer to Part 2

This scenario exploits a weakness in the hash function used for RSA signatures. A good hash function should be resistant to collisions, meaning it would be extremely difficult to find two different messages with the same hash value. However, if a collision exists, then there are 2 potential vulnerabilities:

1. Signature Forgery:
   - An attacker like bob can forge a signature for a new message (m') by simply using the existing valid signature for a different message (m) that shares the same hash value. This compromises the entire purpose of digital signatures, which is to guarantee the authenticity of the signed message.

2. Loss of Trust:
    - If such a forgery goes undetected, it can lead to a loss of trust in the entire communication system. Recipients (like Bob) can no longer be certain that a signed message truly originated from the claimed signer (Alice) and hasn't been tampered with and hence the whole signature is useless.

RSA Implementation

For this Part of the Assignment, the task was to implement the RSA Algorithm. The implementation should include steps to encrypt and decrypt a short message. There are a total of 7 functions created to establish the RSA Algorithm:

1. `generate_large_prime(bits=64)`
   - Component: Prime Test
   - Description: This function uses the `sympy.randprime` (refer to reference 13 in the reference section of this report) function to generate a large prime number with a specified number of bits. This is for generating the prime numbers p and q used in the RSA algorithm.
2. `extended_gcd(a, b)`
   - Component: Extended Euclidean Algorithm
   - Description: This function computes the greatest common divisor (GCD) of two numbers and also finds the coefficients of the integers in the equation $ax+by=gcd(a,b)$. This is for finding the modular inverse in the RSA key generation process.
3. `mod_inverse(e, phi)`
   - Component: Modular Inversion
   - Description: Implements the calculation of the modular inverse using the extended Euclidean algorithm. This is used to compute the private key d in RSA, which satisfies $ed \equiv 1 (\mathrm{mod} \phi(n))$.
4. `mod_exp(base, exp, mod)`
   - Component: Binary Modular Exponentiation
   - Description: Efficiently calculates the result of (base^exp)mod mod using the method of repeated squaring. This is for both encryption and decryption processes in RSA.
5. `generate_keys()`
   - Component: Key Schedule
   - Description: Integrates the processes of generating large prime numbers, calculating the modulus and Euler's totient function, and determining both the public and private keys. This function contains the entire key generation logic of RSA.
6. `encrypt_file(public_key, input_filename, output_filename, block_size=64)`
   - Component: Encryption
   - Description: Implements RSA encryption by reading plaintext from a file, dividing it into manageable blocks, encrypting each block using the public key, and writing the ciphertext to another file.

7. `decrypt_file(private_key, input_filename, output_filename)`
   - Component: Decryption
   - Description: Decrypts blocks of ciphertext read from a file using the private key and writes the decrypted plaintext to another file. This function completes the RSA cycle by recovering the original plaintext from the ciphertext.

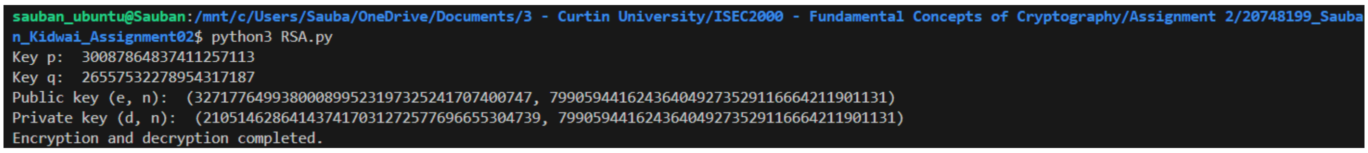
Here is the full Source code for the RSA Algorithm Implementation:

```
import sympy
import random

def generate_large_prime(bits=64):
    """
    Inputs:
        bits (int): The number of bits for the prime number.
    Output:
        int: A large prime number with the specified number of bits.
    Description:
        Generates a large prime number between 2^bits and 2^(bits+1).
    """
    return sympy.randprime(2**bits, 2**(bits+1))

def extended_gcd(a, b):
    """
    Inputs:
        a (int): The first integer.
        b (int): The second integer.
    Output:
        tuple: A tuple containing the gcd of a and b, and the coefficients x and y
such that ax + by = gcd(a, b).
    Description:
        Computes the extended Euclidean algorithm to find the gcd of a and b and
the coefficients x and y.
    """
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1  # Update x coefficient
    y = x1  # Update y coefficient
    return gcd, x, y

def mod_inverse(e, phi):
    """
    Inputs:
        e (int): The exponent in the RSA algorithm.
        phi (int): Euler's totient function value of n (p-1)*(q-1).
    Output:
        int: The modular inverse of e modulo phi.
    Description:
        Calculates the modular inverse of e modulo phi using the extended
Euclidean algorithm.
```

```
    """
    gcd, x, _ = extended_gcd(e, phi)
    if gcd != 1:
        raise Exception('Modular inverse does not exist')  # e must be coprime to
phi
    else:
        return x % phi

def mod_exp(base, exp, mod):
    """
    Inputs:
        base (int): The base of the exponentiation.
        exp (int): The exponent.
        mod (int): The modulus.
    Output:
        int: The result of (base^exp) % mod using the method of repeated squaring.
    Description:
        Efficiently computes the modular exponentiation using the method of
repeated squaring.
    """
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:  # If the exponent is odd, multiply the base with the
result
            result = (result * base) % mod
        exp = exp >> 1  # Right shift exponent, equivalent to dividing by 2
        base = (base * base) % mod  # Square the base
    return result

def generate_keys():
    """
    Output:
        tuple: A tuple containing the public and private keys.
    Description:
        Generates RSA public and private keys.
    """
    p = generate_large_prime()
    q = generate_large_prime()
    print("Key p: ", p)
    print("Key q: ", q)
    n = p * q
    phi = (p - 1) * (q - 1)
    e = sympy.randprime(2, phi)  # Choose e that is a prime number and coprime to
phi
    while sympy.gcd(e, phi) != 1:
        e = sympy.randprime(2, phi)
    d = mod_inverse(e, phi)
    print("Public key (e, n): ", (e, n))
    print("Private key (d, n): ", (d, n))

    return (e, n), (d, n)

def encrypt_file(public_key, input_filename, output_filename, block_size=64):
    """
    Inputs:
        public_key (tuple): The public key (e, n).
        input_filename (str): The path to the input file containing plaintext.
        output_filename (str): The path to the output file to write ciphertext.
        block_size (int): The size of each block to be encrypted.
    Description:
        Encrypts a file's contents block-wise and saves the ciphertext to another
file.
    """
    with open(input_filename, 'r', encoding='utf-8') as file:
        plaintext = file.read()

    e, n = public_key
    block_int_size = n.bit_length() // 8 - 1  # Calculate block size in bytes
    blocks = [plaintext[i:i+block_int_size] for i in range(0, len(plaintext),
block_int_size)]

    ciphertext_blocks = []
    for block in blocks:
        m = int.from_bytes(block.encode('utf-8'), 'big')
        if m >= n:
            raise ValueError("Block size too large for the key size")
        c = mod_exp(m, e, n)
        ciphertext_blocks.append(hex(c)[2:])  # Convert ciphertext to hex without
'0x'

    with open(output_filename, 'w') as file:
        file.write(' '.join(ciphertext_blocks))  # Write hex ciphertext blocks
separated by spaces

def decrypt_file(private_key, input_filename, output_filename):
    """
    Inputs:
        private_key (tuple): The private key (d, n).
        input_filename (str): The path to the input file containing ciphertext.
        output_filename (str): The path to the output file to write decrypted
text.
    Description:
        Decrypts a file's contents that were encrypted in blocks and saves the
plaintext to another file.
    """
    with open(input_filename, 'r') as file:
        ciphertext_blocks = file.read().split()

    d, n = private_key
    plaintext_blocks = []
    for block in ciphertext_blocks:
        c = int(block, 16)
        m = mod_exp(c, d, n)
        message_length = (m.bit_length() + 7) // 8
        plaintext_block = m.to_bytes(message_length, 'big').decode('utf-8')
        plaintext_blocks.append(plaintext_block)
    with open(output_filename, 'w', encoding='utf-8') as file:
        file.write(''.join(plaintext_blocks))  # Reassemble the plaintext blocks
and write

# Key generation
public_key, private_key = generate_keys()

# Encryption
encrypt_file(public_key, 'RSA-test.txt', 'Encrypted-RSA.txt')

# Decryption
decrypt_file(private_key, 'Encrypted-RSA.txt', 'Decrypted-RSA.txt')

print("Encryption and decryption completed.")
```

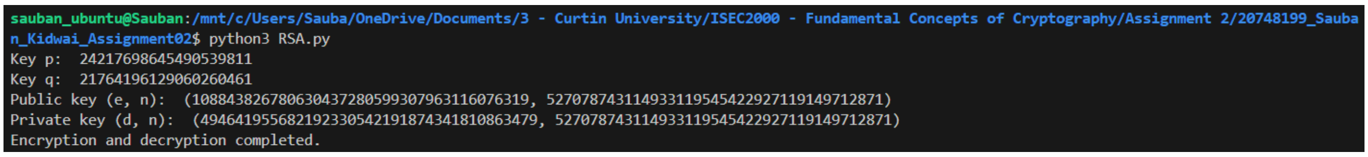
A few Print statements were added in the functions to see and make sure that the algorithm works as intended. To run this program (assuming you have a linux environment) go to the directory in which this assignment is located, open a terminal window and type the following command `python3 RSA.py`. This command will then run the RSA Algorithm taking in `RSA-test.txt` as the input file for the encryption, and then having its encrypted output in the file `Encrypted-RSA.txt` and the decrypted output in the file `Decrypted-RSA.txt`.

The following image shows an example of the output when running the program once:



```
sauban_ubuntu@Sauban:/mnt/c/Users/Sauba/OneDrive/Documents/3 - Curtin University/ISEC2000 - Fundamental Concepts of Cryptography/Assignment 2/20748199_Sauban_Kidwai_Assignment02$ python3 RSA.py
Key p:  30087864837411257113
Key q:  26557532278954317187
Public key (e, n):  (327177649938000899523197325241707400747, 799059441624364049273529116664211901131)
Private key (d, n):  (210514628641437417031272577696655304739, 799059441624364049273529116664211901131)
Encryption and decryption completed.
```

As shown, the terminal displays the value of the public Key (e, n) and the private key (d, n) as well as the prime numbers p and q. Running the program again, gives another value of the keys ensuring that the encryption is done successfully. One thing that was checked to make sure that the logic worked is that the value n is the same for the public and private key, which in both images show that has been done correctly.



```
sauban_ubuntu@Sauban:/mnt/c/Users/Sauba/OneDrive/Documents/3 - Curtin University/ISEC2000 - Fundamental Concepts of Cryptography/Assignment 2/20748199_Sauban_Kidwai_Assignment02$ python3 RSA.py
Key p:  24217698645490539811
Key q:  21764196129060260461
Public key (e, n):  (108843826780630437280599307963116076319, 527078743114933119545422927119149712871)
Private key (d, n):  (494641955682192330542191874341810863479, 527078743114933119545422927119149712871)
Encryption and decryption completed.
```

Then, to verify that the encryption is done correctly, the format of the cipher text in `Encrypted-RSA.txt` should be in hexadecimal format for easy readability, as shown below.

```
10f7b9f8c9d54fcdc121c077ec9a23013 1969683755eb09512b9af87470aeb8807
1d656bd785a2eea99d5d767dfa9c1e8f5 1d8ac956c941a402b36afebc1e6f2a7e1
1969683755eb09512b9af87470aeb8807 15ca5ff058707e8ccec021c02d8917db5
14c7b102f4664b6fdfc0b8b81dd393b40 1c0fa685bb49096dddb9378236cc970b2
175c2cc4d5c75ee36fb324f1f80e0c192 1a159aea344a5ec5f68a08fec81fbcade
2b71bc32423f3269a9d1776075785928 d2a4e573bf200296a118319805bbf72a
1c6fa4c108c3de68ec7d70632584e1ea2 5324fa4cfc72e61c0680a44a0e5c6951
1cb935c0ce0089f3c48e17fc6668bdd76 1157e59030493f908024cc06973dc55f1
111c0cf05e3863a829147a4ba735d4bf3 50743ad5f568222da36e15eb05550586
```

```
1b98daaea148413a127372643e7e06eec 8317d6c58a841bb0ad55bd7293b5ffd0
1d41a339949f32f6ce182854e8cca24a7 10475c96b62e8913933868bbfc566a65e
9c86dff5ae423cefcee884d78d0dc0be 1e366234ac94990c552e7eadcd32afd09
1ab7056becd614a5a18dce4389f2af07d eff7c54f1fefa987bfad1b10eb0b2dbd
7e75cf59ddca7f74476c0328e0586191 ccffa12de89ce5083148e4c455edb519
607aba5effca63e65632196d674d6a35 3603dace778824c487f1ccea64c7e527
61743611ea5ca179183fe058691105f8 f1089de80f3a73407cbd9ab9cdc15adc
29380e158157b17a3b614439e36b532c 129c31301a67f01f9d567aa03a475e8ea
1b13afde87bfe9164a2cd4e8ca0281249 8cdd3d56c2949554443d5928bd9f980
1889b8f87f72164a9e39b7af2058b27c 2744490904b2d73dae8e6c9a197f20b4
19466bdd89393fcf47e3bf6ac7c4597e0 2c367163d2c5177af81475c6fc1358a7
98325c7a624ca253f9279ec90d914421 13df2b0896081776163a131a3a6ef3fb8
aca3a62ed351b77ab7348b91ec1c9658 57cd772ced4a8db0842b6fda9714d69
1f740e2c12e38b00ecf231e34eff00e6d 141288466d096b822dab367b84c17f63c
196c87744f879f9c99fa59ffbe733b17d 16e8760c097059692d0dfaaab12725abf
784ca029d72de35d9c710863d4e4a10 adbfa31c33588e866594afc6ae82eed
ff317b2b08e78e98c122324b70809b15 a22c4b55f8fad5d97dbbcf3b8027b29
16b7a394396ba94f788b92d36a44b5f4 1041289e3c499a1a6468a93c43c15bae8
fd86754b77e9f39fbb87ee79fe7d8239 1752dcf327a0e84e0fee49202dfdbbbb8
6b23b53fb70035227149654366b2d129 105241d2bbd75bcec1aae82474ba14202
6f205002fe03a30a4600f2b27432c430 108d537b04ad5996f2d68b504bc5ba293
20c4278a4f27d95a47ed5856d506879d6 14b8bd5eba0751aa22943eeef9e0bb4b8
d5e88af754b3756c40767811c8bbcd1b 4b12cc4e94b5834ad62ff8b4caa9b22a
1cf8fd19e00a787bc82dbede91cc088c 1a4db64c57520ebb842eefdfdc7ecae3c
58e52d8ff1f7fd8d6d890c4e5487d572 1a6ba00550845cdfc4707ed8bcba5178d
f92545a3bdec9371f42dfdf28facff16 cafaa656432fb2ea164cda3e2505503
1994ab7ea7548fdc01db47d50cc578887 1ae8ffcab85925468b7b5883ee7d3c24b
1619c2668b14b237e6d05149ce0ac777d 1532b76ff257009fad8039379fc3a76f0
ec75c9f299f62e9b3bdfb1bdcc59facb c8ffb3ec40972bad7e8c7805e60647b8
da06f59232418bccfbcfaeb5404e0f79 a98a523b109b948c97965d73324e841
121a68f4cd74dd2039c2b72eeb048c30 aaeaad22321448ae1224814caed45336
d515a8fcce46ba67923f656fd16a2dfa 84a533e3986ec8fe2dbada89228e5694
1d5d5cf9ac2c30d954f8e8f1f08e89e2d f07c902a022836757248fb40d0d65d64
44788f945974f077b40baf255e557073 2a4d166b7ea5d023bd93b48b14915377
33e7f91fd0bbbb477700876c6415ad15 1e4d038ae888c1f9656df4f53f396eded
cefe05c861698fddb4a4eb08fd3516a4 1f690fa1e0e99392248ff1931b1cc2344
5c45e4974aa36dcf40082c8a53a6f95f 7ebda7e1523253504a15a719dfc9091f
e111125c259c462fd73d3d947dccba00 1919646df9709e36b4f96d8004feffe3b
1f8a57c32fc0493cd8287e6272c061e2 1affe5cd47bd88ec0b4f04adba28d23e6
6a0e0cb28ef500b6a1fc4f052acb0bad b5311792a619930189f5364e08bc34ae
86c71776829cfccedbacf5234e45657b 1b911e39e933496cbe444f2e6af4e15ad
1f0da80ae5e1441acee9b9b816927090b 19885ad0f424d747c287f2ec9a6e7e60b
f6e8c5ae7abbeb5c2b0fe592f3bd9b44 12ad3550fb3a7d292f5c61d8b456199ff
10ae6dddb7cb96eba20c5ba514868d955 10612a4a2db1dfa3a0cf2f2a8d5b84f0e
12bb44381d952e3a061c9d48ca0670055 d4975a912e8551ed01bc49eca01afce8
2587a308a45a155b982e83b0cdae2fe8 7822deae6f0ac836dadbef0747c37160
55114dc4d00cf3e1559bab76f17efc42 13a62ed3a154824089623e3bbdff36dc7
17def5b9d0af9aaab42ffd118e9ab0f79 19666c44226fd06f31e00b1672505bb47
ab487e9408629327adb3932da32a1044 92041c75515a4ec6b4c14121f13f23dc
d1c79a1e6232b2108e4689df7a349732 62f3d1911a5fcef75eaad30b0b398bc6
1713c517b7e00d32918a559f790a1a925 1c97b1cf77c2d966f3a5a7ff5f1f4c7e3
b0315705a4b768ec954e33dbd8128b19 1c50985182869eae1b971a00285b1e9bc
45a58b807ec6d7982f5a48d5e3632fa e3524276169222ba38d425cefdc2d5df
25ba70778728eb01d879cb0a7592223a c89a7e80b01502bdb9ae911f203ee178
```

```
79a417f16b87fbfd51da348ddf1326d e32d3a99362c81f8326a66e78d676256
1cd9285dc8ab1ad3cfba2027c36df0ac6 20696aa0214ad1ba954285353720cf6aa
15c7a0653cc1da5f00a3b10ff0f6a62eb 19ab3ec1174fcd069bb456e5c99470207
6d69b54a51b0e243c5221dca9b50b0b1 dc35b7263ef64d367255242074911 24e
1e4ba325cf933b8853c5c1d2cdf2d616c 1d2a89dbde482a8bc2ea96fef6961171
b6245b978bb1a10974363061a4ed9f51 1cca7eba6dae607525e47e17940a19695
1101891b98e388386ed34e9c568226924 1c59575187672dc499e82e7091e2cda18
1c619ea7064aaa31c2d09e5acde3efaa2 578195fd03231dec0ffd85e7b4b71387
18fc56a4bb5990c27864c4a85d612ebcf 75c40415cc4e99f8c7f994faeb33cbe2
1c166bc72bd8a683461b3305dc8217bde 8ca2079b21c7f4c6da7ec699bdfb8c57
1fa0aa23ed2c3041d0a134cbb68be0a30 7a4a8f582ea79c171cb5349974854f9
374eaf4876a5a2f057ce8bf2a6361b54 1ca4d037de1d9bcd95c4f4d494bbff32b
54528cac130c69f707c10f9e2da2cd50 841a0ea0720e68098c2018bd85968303
17731b8f95d2b6b8fb4ef28623eb29036 63d17931b2ac528d34bf87a300a13832
62c317343bb0d035f31ebd2506daceda 1be1004c383c6376313b282c89abb4d8f
172ac78130c8a74450d0fe02e3547f059
```

Then to verify that the decryption is done correctly, the plaintext in the file `Decrypted-RSA.txt` should be exactly Identical to the test file `RSA-text.txt` which it is as shown below.

```
%%%========================== This is a sample message
================================
\uplevel{\Large \textbf{Question answering}}
\begin{questions}
\question[10] The Euclidean algorithm is based on the following assertion. Given
two integers $a$, $b$, ($a>b$),

\begin{equation}\label{gcd}
  \text{gcd}(a, b) = \text{gcd}(b, a \,\, \text{mod} \,\, b).
\end{equation}
Prove the assertion (\ref{gcd}) \textbf{mathematically}. (Note that proof by
example is NOT appropriate here)

\question[20] Assuming that Alice signed a document $m$ using the RSA signature
scheme. (You should describe the RSA signature structure first with a diagram and
explain the authentication principle). The signature is sent to Bob. Accidentally
Bob found one message $m'$ ($m\neq m'$) such that $H(m)=H(m')$, where $H()$ is the
hash function used in the signature scheme. Describe clearly how Bob can forge a
signature of Alice with such $m'$.

\vspace{0.5in}
\fullwidth{\Large \textbf{Programming}}
\question[50] Implement the RSA algorithm (C/C++, Java, Python). The requirements
are as follows:
\begin{itemize}
  \item Implement each component as a separate function, such as key schedule,
prime test, the extended Euclidean algorithm, binary modular exponentiation, and
so on.
  \item Implement both encryption and decryption of RSA. Encryption takes a txt
file as input and output another txt file containing ciphertext (use hexadecimal
for easy readability). Decryption should recover the plaintext.
```

```
  \item Your code should encrypt and decrypt standard keyboard characters,
including letters, numbers, and symbols.
  \item The prime numbers $p$ and $q$ should be larger than $2^{64}$. (you are
allowed to use libraries to handle large numbers, such as BigInteger in Java)
  \item The strategy of source coding (converting characters to integers in RSA)
is up to you. You can encrypt one or more characters at a time, but make sure the
constraint $m<n$ is satisfied.
  \item Use the provided file \text{RSA-test.txt} to test your code.
\end{itemize}
After implementing your code, please \textbf{answer the following questions} in
your report:
\begin{parts}
  \part[10] What are the lessons you learned, and difficulties you met, in the
process of implementing RSA?
  \part[10] Describe what you have done for source coding and decoding.
\end{parts}
\end{questions}
```

In conclusion, the RSA Implementation does work as intended, and fulfils all criteria required for this to be a accurate and proper algorithm.

Conclusion

This assignment was a fun one to complete. Proving the equation in question 3 was a bit hard, especially when there are so many terms to simplify or expand. The RSA Implementation was moderate difficulty. I think that splitting the text into blocks was the hardest part of the implementation as I needed to make sure that it is split correctly to convert to hexadecimal.

References

1. CSE311: Foundations of computing I. Accessed May 16, 2024. https://courses.cs.washington.edu/courses/cse311/.
2. Daniel, Brett. “Symmetric vs. Asymmetric Encryption: What’s the Difference?” Trusted Computing Innovator, February 7, 2024. https://www.trentonsystems.com/en-au/blog/symmetric-vs-asymmetric-encryption#:~:text=Unlike%20symmetric%20encryption%2C%20which%20uses,’%20and%20recipients’%20sensitive%20data.
3. “The Euclidean Algorithm (Article).” Khan Academy. Accessed May 16, 2024. https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm.
4. “Euclidean Algorithm.” Wikipedia, April 9, 2024. https://en.wikipedia.org/wiki/Euclidean_algorithm.
5. “Euclidean Algorithm: Brilliant Math & Science Wiki.” Brilliant. Accessed May 16, 2024. https://brilliant.org/wiki/euclidean-algorithm/.
6. Simplilearn. “What Is RSA Algorithm in Cryptography?: Simplilearn.” Simplilearn.com, February 13, 2023. https://www.simplilearn.com/tutorials/cryptography-tutorial/rsa-algorithm.
7. Uomustansiriyah. Accessed May 16, 2024. https://uomustansiriyah.edu.iq/media/lectures/5/5_2021_06_13!07_03_14_PM.pdf.
8. GeeksforGeeks. “RSA and Digital Signatures.” GeeksforGeeks, May 7, 2023. https://www.geeksforgeeks.org/rsa-and-digital-signatures/.
9. “Practical Cryptography for Developers.” RSA Signatures. Accessed May 17, 2024. https://cryptobook.nakov.com/digital-signatures/rsa-signatures.
10. “Cryptographic Hash Function.” Wikipedia, May 7, 2024. https://en.wikipedia.org/wiki/Cryptographic_hash_function.
11. “Hash Function.” Wikipedia, April 17, 2024. https://en.wikipedia.org/wiki/Hash_function.
12. GeeksforGeeks. “What Is a Symmetric Encryption?” GeeksforGeeks, January 29, 2024. https://www.geeksforgeeks.org/what-is-a-symmetric-encryption/.
13. GeeksforGeeks. “Python: Sympy.Randprime() Method.” GeeksforGeeks, August 27, 2019. https://www.geeksforgeeks.org/python-sympy-randprime-method/.