RSA Algorithms Report:

Course: Computer science fundamentals

Name: Mohamed Moheb

Id:2201214

Introduction:

Confidentiality, integrity, availability are three main pillars of information security world and commonly known by the acronyms CIA, these pillars are used to protect any communication between two or multiple parties. To achieve this protection, confidentiality, the first pillar, ensures the accessibility of information only for authorized individuals or corporations, it can be achieved through security measures such as the encryption. The art of hiding a message, referring to the encryption security measure, operates through two primary methods: symmetric encryption and asymmetric encryption. While both methods aim to conceal messages, this report focuses on asymmetric encryption due to the task requirements. Also known as public key encryption, asymmetric encryption uses public-private key pairing enabling a secure data transfer/exchange without relying on a shared secret key between parties unlike symmetric encryption (cloudflare, 2024).One of the algorithms used in asymmetric encryption is RSA algorithms, named after its inventors Ron Rivest, Adi Shamir and Leonard Adleman, its security relies on generating a public key that consists of the multiplication of two prime numbers and chosen exponent, from which the private key is derived using mathematical equations. The strength of encryption relies on the size of the key, when it increases the strength of encryption increases exponentially (Simplilearn, 2023). The concept of RSA algorithm will be explained in detail later in the report.

Problem statement:

This report focuses on the implementation of asymmetric encryption by delving into two key algorithms of the RSA encryption model. The first algorithm involves implementing a python code to factor a modulus *N* and calculate the private exponent (private key), check if its correct. As an additional code serves as a generator for **N** and **e** values The second algorithm involves implementing a python code that attempts to brute force the value of the private exponent, check if its correct. Each code is accompanied by a set of steps (decomposition of the problem, pseudocode, testing, Calculate the Big O complexity and justify, generate 8/16 bits test cases, runtime, determine the better code) allowing to achieve an understanding idea of the RSA algorithms and security implications.

First Algorithm: Factoring Modulus N.

Decomposition of The Problem:

The problem of factoring the modulus ***N*** and calculating the private exponent ***d*** in the RSA algorithm is approached by decomposing it into several steps:

I. Identifying potential prime factors:
   - First step is to identify potential prime factors ***p*** of ***N***.
   - By iterating through possible values of ***P***.

- Check if ***N*** is divisible by ***p*** (N % p == 0)

II. Checking primality of Factors:
- For each potential factor ***p***, code checks whether its prime using **is_prime** function.
- Ensuring that only prime factors are considered for further calculations.

III. Calculations:
- Once a prime factor ***p*** is identified, calculate prime factor ***q***.
- Corresponding prime factor ***q*** is calculated by dividing ***N*** by ***p***.
- Using prime factor ***p*** and ***q***, in addition to public exponent ***e*** to calculate private exponent ***d***.
- By using **calculate_private_exponent** function.

IV. Code structure:
- Define a **factor_modulus** function taking as arguments modulus ***N*** and public exponent ***e***.
- Encapsulate inside this function both function **is_prime(n)** and **calculate_private_exponent(p,q,e).**

V. Checking correctness of d:
- Use public exponent to encrypt a plain message.
- Use private exponent to decrypt cypher message.
- If decrypted message is identical to the plain message of the start.
- Return True.

V. Results:
- The code finally returns prime factors ***p*** and ***q***, along with ***d*** the private exponent.
- The run time of the code
- Whether the private exponent is correct or not.

Code:

Below is the python code for factoring modulus **N** and calculate private exponent **d**:

```
def factor_modulus(N, e):
    """Factor the modulus to obtain prime factors and calculate the private
exponent."""

    def is_prime(n):
        """Check if a number is prime using trial division."""
        if n <= 1:
            return False
        if n <= 3:
            return True
        if n % 2 == 0:
            return False
        sqrt_n = int(n**0.5) + 1 #square root of n.
        for i in range(3, sqrt_n, 2):
            if n % i == 0:
```

```
                return False #in case not prime.
        return True #in case prime.

    def calculate_private_exponent(p, q, e):
        """Calculate the private exponent d."""
        Euler_totient = (p - 1) * (q - 1)#essential for deriving the private
exponent.
        d = pow(e, -1, Euler_totient)# pow() function calculates modular
exponentiation efficiently.
        return d


    for p in range(2, N):
        if N % p == 0 and is_prime(p):#checks if p factor of N and p is prime
factor.
            q = N // p
            d = calculate_private_exponent(p, q, e)
            return p, q, d
```

To check whether the value of **d** is correct or not:

```
def rsa_encrypt(message, e, N):
    """Encrypt a message using RSA."""
    encrypted_message = pow(message, e, N)
    return encrypted_message

def rsa_decrypt(encrypted_message, d, N):
    """Decrypt an encrypted message using RSA."""
    decrypted_message = pow(encrypted_message, d, N)
    return decrypted_message

def check_private_exponent(d, N, e, test_message):
    """Check if the private exponent is correct by encrypting and decrypting a
test message."""
    encrypted_message = rsa_encrypt(test_message, e, N)
    decrypted_message = rsa_decrypt(encrypted_message, d, N)
    return decrypted_message == test_message
```

Test Cases and Run Time:

bits	***N***	***e***	***p***	***q***	***d***	***Runtime***
8	**2773**	**13**	**47**	**59**	**821**	**0.000034 s**
8	**4087**	**17**	**61**	**67**	**233**	**0.000027 s**
16	***2737202731***	**65537**	**50131**	**54601**	**865353473**	**0.02892490000000003s**
16	**3255716201**	**65537**	**54959**	**59239**	**1357043129**	**0.028169800000000023s**

To calculate runtime of the code a python module named **timeit** is being used, it provides an accurate timing for short code snippets. Here are the additional lines included in the code to use **timeit**:

```
import timeit

runtime="""p, q, d = factor_modulus(N, e)"""
execution_time = timeit.timeit(runtime, globals=globals(), number=1)#number=1
ensure that the code is only timed once
print("Runtime:", execution_time, "seconds")
```

Big O Complexity:

To calculate the time complexity, a breakdown of the complexity of each component in the code is necessary:

I. **Is_prime** function:
- Inside the loop it performs a constant time operation.
- It iterates up to the square root of n, which is $\sqrt{N}$.
- Therefore, the complexity of **is_prime** function is $O(\sqrt{N})$.

II. **Calculate_private_exponent** function:
- Performing a constant number of arithmetic operations regardless of the size of the input.
- Therefore, the complexity of **is_prime** function is $O(1)$.

III. **factor_modulus** function:
- the outer loop iterates from 2 up to **N**.
- inside this loop the **is_prime** function is called.
- Therefore, the overall time complexity of the **factor_modulus** function is $O(N*\sqrt{N})$.
- It can be simplified to $O(N\sqrt{N})$.

To conclude, the overall time complexity of the code is $O(N\sqrt{N})$.

Second Algorithm: Brute Force To Obtain The Value Of Private exponent d.

Decomposition of The Problem:

The problem of brute forcing private exponent ***d*** in the RSA algorithm is approached by decomposing it into several steps:

I. Candidate private exponent:
- Start with a candidate value of 2 for **d**.

II. Incrementing/testing:
- Increment **d** by 1 until finding a value that satisfies the condition **pow(e, d, Euler_totient) == 1.**

III. Calculations:
- Calculate Euler totient using this formula: **(p-1)*(q-1).**
- While loop that stops only if the condition is met.
- Condition is if **(e*d) % Euler totient == 1.**

IV. Code structure:
- Define a **brute_force_private_exponent** function that takes **p** and **q**, along with e public exponent as parameters.

V. Result:
- Return value of private exponent **d**.
- The run time of the code
- Whether the private exponent is correct or not.

Checking correctness of d:

- Use public exponent to encrypt a plain message.
- Use private exponent to decrypt cypher message.
- If decrypted message is identical to the plain message of the start.
- Return True.

Code:

Below is the python code for brute forcing the value of private exponent **d**:

```
def brute_force_private_exponent(p, q, e):
    """Brute force the private exponent d."""
    Euler_totient = (p - 1) * (q - 1)
    d = 2  # Start with a candidate value of 2
    while True: # loop continues until correct value of d is found.
        if (e * d) % Euler_totient == 1:
            return d
        d += 1  # Try the next value of d
```

To check whether the value of **d** is correct or not:

```
def rsa_encrypt(message, e, N):
    """Encrypt a message using RSA."""
    encrypted_message = pow(message, e, N)
    return encrypted_message

def rsa_decrypt(encrypted_message, d, N):
    """Decrypt an encrypted message using RSA."""
    decrypted_message = pow(encrypted_message, d, N)
    return decrypted_message

def check_private_exponent(d, N, e, test_message):
    """Check if the private exponent is correct by encrypting and decrypting a
test message."""
    encrypted_message = rsa_encrypt(test_message, e, N)
    decrypted_message = rsa_decrypt(encrypted_message, d, N)
```

```
    return decrypted_message == test_message
```

Test Cases and Run Time:

bits	***e***	***p***	***q***	***d***	***Runtime***
8	**13**	**47**	**59**	**821**	**0.00019 s**
8	**17**	**61**	**67**	**233**	**0.0000589s**
16	**65537**	**50131**	**54601**	**865353473**	**Non calculated due to bit size.**
16	**65537**	**54959**	**59239**	**1357043129**	**Non calculated due to bit size.**

To calculate runtime of the code a python module named **timeit** is being used, it provides an accurate timing for short code snippets. Here are the additional lines included in the code to use **timeit**:

```
import timeit
code_to_measure = """d = brute_force_private_exponent(p, q, e)"""
execution_time = timeit.timeit(code_to_measure, globals=globals(), number=1)
print("Runtime:", execution_time, "seconds")
```

Big O Complexity:

In the worst-case scenario, the code iterates through every possible value of private exponent **d** until it finds the correct value. **N** represents the value of the Euler's totient function, which is related to the modulus **N** in the RSA algorithm. Therefore, the big O complexity of the **brute_force_private_exponent** function is **O(N).**

***Comparison Between factor_modulus and* brute_force_private_exponent:**

As seen below, the run time of the factor_modulus function is faster than the run time of the brute_force_private_exponent function to find the same private exponent **d**, because it attempts to factorize the modulus **N** into its prime factors **p** and **q**, reducing the search space and time for the private exponent **d**. While brute_force_private_exponent iterates through every possible value of the private exponent until it finds the correct value.

Private exponent d	***Runtime factor_modulus***	***Runtime brute_force_private_exponent***
821	**0.000034 s**	**0.00019 s**
233	**0.000027 s**	**0.0000589s**
865353473	**0.028924900000000003s**	**Code runner crashed while executing the test case.**
1357043129	**0.028169800000000023s**	**Code runner crashed while executing the test case.**

In terms of security, factor_modulus function offers a more robust resilience against attacks since it relies on factoring large numbers. When dealing with a large modulus value N, it becomes a computationally intensive task to factor its value. Therefore, factoring the modulus in RSA algorithm is indeed a laborious task for attackers to breach. In the other hand, brute force attacks become increasingly less effective as the modulus size grow, since the search space grows exponentially with the length of the key. For instance, attempting to find the value of d using brute force necessities testing every single value of d until the correct one is found, as the size of N increases the range of possible values of d increases.

To conclude, factoring the modulus in asymmetric encryption provides a more efficient and secure approach compared to brute force method. As it stands as a robust method for safeguarding the communication and data exchange.

Below is a link for GitHub repository:

https://github.com/mohamed-moheb/RSA-code.git

References

cloudflare. (2024). *What is asymmetric encryption?* Retrieved from cloudflare: https://www.cloudflare.com/it-it/learning/ssl/what-is-asymmetric-encryption/

Simplilearn. (2023, February 13). *What Is RSA Algorithm and How Does It Work in Cryptography?* Retrieved from Simplilearn: https://www.simplilearn.com/tutorials/cryptography-tutorial/rsa-algorithm#:~:text=The%20RSA%20algorithm%20is%20a,to%20be%20fast%20post%2Ddeployment.