LONDON METROPOLITAN UNIVERSITY

CS7080 Cloud Computing and Internet of Things – Spring 2023-24

Coursework – I

Case Study

RSA algorithm Report

Submitted By:

Rasoul Abareyan (23048080)
Alireza Samaei-Yekta (22079130)
Usama Hussain (23030627)

Submitted To:
Graham Taylor-Russel

Date: April 2024

Abstract:

Encryption is a primary means of guaranteeing the security of confidential information. It dealt with digital signatures, authentication, secret storage, system security, and other topics in addition to information secrecy approaches. Therefore, the purpose of employing encryption techniques is to ensure the information's confidentiality, integrity, and certainty and prevent it from being changed, fabricated, or counterfeited..[1]
RSA is now the most popular and commonly used public key system. The 1978 publication "A method for obtaining digital signatures and public-key cryptosystems" by RL Rivest and coworkers contained the first reference to it. It functions as a block cipher and is an asymmetric (public key) cryptosystem based on number theory. The complexity of the huge number prime factorization—a well-known mathematical problem without an effective solution—is the foundation for its security [2]. One of the most popular methods for applying public key cryptography to encryption and digital signature standards is the RSA public key cryptosystem.

Introduction:

We worked on the RSA algorithm in Maple software, but it didn't work out as we expected. Therefore, we decided to implement our algorithm with Python and follow our coursework structure. We will describe our algorithm in this report, followed by a discussion of the advantages and disadvantages of RSA. For running our rsa.py file, you can use Jupiter or PyCharm to run this algorithm.

Why we choose RSA:

Due to the difficulty of factoring large composite numbers into their prime factors, RSA provides robust security guarantees. Since there are currently no effective quantum or classical algorithms for factoring large numbers, RSA is a reliable option for secure communication.
Because RSA is an asymmetric encryption algorithm, it encrypts and decrypts data using two different keys: a public key and a private key. This makes it possible for parties to communicate securely without first exchanging secret keys.[6]

Explain Algorithm:

In the first step, because we need to calculate gcd (greatest common divisor) in different parts of our algorithm, define a function whose name is gcd and it will take two numbers (a and b) and calculate gcd. Then everywhere we want to calculate them, we will use this function. However, you can use the gcd library in Python math but we implement this function.
from math import gcd

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

In the next step, we implemented a function (is_prime) to see whether a certain number, n, is prime. This function makes use of a probabilistic approach that works well for big numbers: the Fermat primality test. The function is broken into as follows:

If n is less than or equal to 1, it is not prime

```
if n <= 1:
    return False
```

Perform the Fermat primality test k times

```
for i in range(k):
```

Generate a random integer 'a' in the range (2, n-1)

```
    a = random.randint(2, n - 1)
```

If a^(n-1) mod n is not equal to 1, then n is definitely composite

```
    if pow(a, n - 1, n) != 1:
        return False
```

If n passes the test for all k random 'a's, it is likely prime

The function returns True if the integer n passes the Fermat primality test for all k random a values, meaning that n is probably prime (with a probability dependent on the selected value of k).

```
def is_prime(n, k=5):
    if n <= 1: return False
    for i in range(k):
        a = random.randint(2, n - 1)
        if pow(a, n - 1, n) != 1:
            return False
    return True
```

The following function, generate_prime, produces prime numbers with a given bit length. To verify if the produced numbers are primate, this program uses the is_prime function. The function is broken into as follows:

The single argument bits, which indicates how many bits the created prime number should have, is what the function generate_prime accepts as input.

The code uses random to create a random integer p with bits number of bits inside the while loop. acquirerandbits (bits).

The produced number p is then checked to see if it is prime by using the is_prime function.

The function returns p if it is prime. If not, the cycle keeps on, producing new random numbers until a prime number is discovered.

```
def generate_prime(bits):
   while True:
        p = random.getrandbits(bits)
        if is_prime(p):
            return p
```

To create a public-private key pair for an RSA encryption system, use generate_keypair.
First, we select two (often big) primes, p and q, and a smaller integer, e, which does not share any factors with (p-1)(q-1).
We calculate *n* = *pq* and our public (freely available) key is (*n*,*e*) , and calculate phi and e which we use the random library for calculating e.
public exponent Choose a value of e such that 1<e<phi(n) and gcd(phi(n), e) = 1.

The public key (n, e) and private key (n, d) are then returned after computing the private exponent d as the modular inverse of e modulo (p-1)(q-1).
Pow(e, -1, phi), the modular inverse computation, might not always produce the right answer, particularly if e and phi are not coprime. Using a reliable technique to compute the modular inverse, such as the extended Euclidean algorithm, is a safer option.

So after that, we can create our public and private keys in our encryption and decryption functions.

```
def generate_keypair(bits):

#     get random our p and q
    p = generate_prime(bits // 2)
    q = generate_prime(bits // 2)

    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randint(1, phi)

#   public exponent: Choose a value of e such that 1<e<phi(n) and gcd(phi(n), e) = 1.

    public_exponent = gcd(e, phi)
    while public_exponent != 1:
        e = random.randint(1, phi)
        public_exponent = gcd(e, phi)
#     d = mod_inverse(e, phi)

#     the inverse d of e mod (p-1)(q-1).
    phi = (p - 1) * (q - 1)
    d = pow(e, -1, phi)
    return ((n, e), (n, d))
```

Avoiding frequency analysis:

The purpose of this code is to modify plaintext by adding or deleting padding zeros so that it fits inside a specific block size.

The two inputs required by the add_zero_padding function are block_size and plaintext.
It determines how much padding is required to completely fill the plaintext to the block size.
Subsequently, it generates a zero padding (\x00) using the determined length.
Lastly, it returns the padding that was applied to the plaintext's start.
The input for the remove_padding function is the padded plaintext.
It looks up the plaintext's index for the first non-zero byte.
It eliminates the padding by returning the plaintext substring beginning at that index if it discovers a non-zero byte.
It yields an empty string if every byte is 0.

```
def add_zero_padding(plaintext, block_size):
    """Add padding zeros from the plaintext."""
    padding_length = block_size - len(plaintext)
    padding = b"\x00" * padding_length
    return padding + plaintext


def remove_padding(plaintext):
    """Remove padding zeros from the plaintext."""
    # Find the index of the first non-zero byte
    index = next((i for i, byte in enumerate(plaintext) if byte != 0), None)
    if index is not None:
        return plaintext[index:]
    else:
        return b""
```

Now we can implement our encryption and decryption functions :

Encryption function:

```
def encrypt(public_key, plaintext):
    n, e = public_key
    block_size = (n.bit_length() + 7) // 8
    encrypted_chunks = []

    if len(plaintext) > block_size:
        for i in range(0, len(plaintext), block_size):
            chunk = plaintext[i:i + block_size]
            padded_plaintext = add_zero_padding(chunk.encode(), block_size)
            padded_plaintext_int = int.from_bytes(padded_plaintext, 'big')
            ciphertext = pow(padded_plaintext_int, e, n)
            encrypted_chunks.append(ciphertext)
    else:
        padded_plaintext = add_zero_padding(plaintext.encode(), block_size)
        padded_plaintext_int = int.from_bytes(padded_plaintext, 'big')
        ciphertext = pow(padded_plaintext_int, e, n)
        encrypted_chunks.append(ciphertext)

    return encrypted_chunks
```

It begins by extracting n and e, the two splits of the public key.
It uses the length of n to compute the block size. In the event that the plaintext is longer than the block size, this is crucial for dividing it into smaller sections.
To store the encrypted portions of the plaintext, it creates an empty list called encrypted_chunks.
If the plaintext's length exceeds the block size, it must be divided into many portions in order to be encrypted. It then goes into an iteration loop over every section.
Using the add_zero_padding function, it adds zero padding to each chunk of plaintext such that its length corresponds to the block size.
It uses int.from_bytes() to transform the padded plaintext into an integer representation.
Using the RSA encryption algorithm, it conducts modular exponentiation on the padded plaintext integer: ciphertext = (padded_plaintext_int ** e) % n.
The generated ciphertext is appended to the encrypted_chunks list.
It just inserts zero padding, transforms the plaintext into an integer, does modular exponentiation, and appends the ciphertext to the list if the plaintext is shorter than or equal to the block size.
The list of encrypted ciphertext chunks is finally returned.

Decryption function:

```python
def decrypt(private_key, ciphertext_chunks):
    n, d = private_key
    decrypted_chunks = []
    block_size = (n.bit_length() + 7) // 8
    for chunk in ciphertext_chunks:
        padded_plaintext_int = pow(chunk, d, n)
        padded_plaintext = padded_plaintext_int.to_bytes(block_size, 'big')
        padding_index = padded_plaintext.find(b"\x00", 2)
        if padding_index != -1:
            plaintext = padded_plaintext[padding_index + 1:]
        else:
            plaintext = padded_plaintext
        # Remove trailing zeros
        plaintext = remove_padding(plaintext)
        decrypted_chunks.append(plaintext)
    return b"".join(decrypted_chunks).decode('utf-8')
```

It extracts n and d, the two halves of the private key.
In order to store the decrypted portions of the ciphertext, it initializes an empty list called decrypted_chunks.
It uses the length of n to compute the block size. Every given ciphertext chunk is iterated over.
The RSA decryption formula, padded_plaintext_int = (chunk ** d) % n, is used to conduct modular exponentiation for each chunk using the private key.
It uses to_bytes() to convert the resultant padded plaintext integer back to bytes.
It looks through the padded plaintext bytes for the padding index. This index shows the beginning of the zero padding.

It slices the padded plaintext to remove the padding and leave only the true plaintext if padding is detected (padding index is not -1). In all other cases, it preserves the padded plaintext.
The remove_padding function is used to eliminate any trailing zeros from the plaintext.
The decrypted plaintext is appended to the decrypted_chunks list.
Once every chunk of the ciphertext has been decrypted, it uses b"".join(decrypted_chunks) to combine the decrypted chunks into a single byte string, which it then decodes into a UTF-8 encoded string.

Save and retrieve the private key:

```
# Saving the Private Keys in order to retrieve the original message later on
def save_private_key(private_key, filename):
    with open(filename, 'wb') as f:
        f.write(str(private_key).encode())


# Retrieve the saved Private Key
def load_private_key(filename):
    with open(filename, 'rb') as f:
        private_key_str = f.read().decode()
        # Convert the string back to a tuple
        private_key = eval(private_key_str)

    return private_key
```

To be able to decode encrypted messages later on, use the same private key which has been used for encryption.RSA private keys can be saved and loaded using the methods load_private_key(filename) and save_private_key(private_key, filename). The function save_private_key accepts a filename and a private key tuple as inputs, opens the file in binary write mode, transforms it into a string representation, uses the.encode() function to encode it into bytes, and then closes the file. The load_private key function reads the contents of the file into a string, uses eval to convert it back to a tuple, and then returns the loaded private key tuple. The file is opened in binary read mode. These features enable the RSA private keys to be saved and retrieved.

Inputs and En/decryption messages:

```
if __name__ == "__main__":
    bits = 1024
    public_key, private_key = generate_keypair(bits)

    if len(sys.argv) != 2:
        userInput = input("for Encrypting, enter 1 \n for Decrypting, enter 2: ")
    else:
        userInput = input("Invalid input\n for Encrypting, enter 1 \n for Decrypting, enter 2:  ")

    if userInput == '1':
        userMessage = input("Please enter your message to encrypt: ")
        encrypted_message = encrypt(public_key, userMessage)
        save_private_key(private_key, 'private_key.txt')
        print("Encrypted message:", encrypted_message)
        # print("Encrypted message:",''.join(map(str, encrypted_message)))

    elif userInput == '2':
        userMessage = input("Please enter your message to decrypt: ")
        ciphertext = [int(char) for char in userMessage.split(',') if char.strip()]
        loaded_private_key = load_private_key('private_key.txt')
        decrypted_message = decrypt(loaded_private_key, ciphertext)
        print("Decrypted message:", decrypted_message)
```

Here we are asking the user to select the encryption or decryption function and after that, the user can enter whether the message is to be encrypted or the encrypted cipher text to be decrypted.

Our algorithm has the capability to flexible in bit length e.g. from 16 to 2048 bits. The command-line interface (CLI) for RSA encryption and decryption is provided by this script. With the help of the generate_keypair function, an RSA key pair with a given key size of 1024 bits is produced. The user is prompted to input '1' for encryption or '2' for decryption if the script is executed with command-line parameters. The script assumes an incorrect input if no parameters are given.

Choose '1' to request the user to input a message to be encrypted; the message is encrypted using the encrypt function and the generated public key. Using the save_private_key method, the private key is stored to a file called "private_key.txt," enabling the user to decrypt the message at a later time.

The load_private_key function is used to decode the encrypted message, which is a list of integers, if the user selects option '2'. The private key is obtained from the 'private_key.txt' file. After that, the message is encrypted and printed to the console.

Advantages of RSA:

The main advantage of RSA encryption is that it is very user-friendly and provides a secure means of data transport without requiring the disclosure of a secret key. Larger key sizes can make RSA encryption less safe, and it operates more slowly than other encryption methods. [3]

It is not too difficult to implement the RSA algorithm.
Public keys may be easily distributed to users.
It is very difficult to break the RSA algorithm because of the intricate mathematics involved.
Sending sensitive data is safe and dependable when using the RSA method.
RSA is a trustworthy and secure technique. Sending sensitive information is therefore safe.[4]

Advantages of RSA for our algorithm:

The code is designed with a modular structure, ensuring readability, maintainability, and **reusability**. It uses strong cryptographic primitives like RSA encryption, **random padding (avoiding frequency analyses),** and modular arithmetic operations to ensure the confidentiality and integrity of encrypted messages. Random padding is added to the plaintext before encryption to prevent pattern recognition and attacks based on known plaintext. The code also includes functions for saving and loading RSA private keys from a file, allowing users to securely store them for future decryption. A **user-friendly** interface is provided for interactive encrypting and decrypting of messages. Users can specify the desired key size for RSA key pair generation, providing flexibility in balancing security and performance. The code also performs basic error handling, such as checking for valid user input and file existence, enhancing robustness, and preventing unexpected behavior. Overall, this code offers a practical and accessible implementation of RSA encryption and decryption.
Last but not least, we implemented a flexible bit size for generating keypair.

Disadvantages Of RSA

RSA may sometimes fail because it only uses asymmetric encryption, but full encryption needs both symmetric and asymmetric encryption.
There are situations where having a third party verify the trustworthiness of public keys is required.
The data transfer rate is sluggish since there are a lot of individuals involved.
For encryption of public data, like electoral voting, RSA cannot be utilized.
The receiver must do a lot of processing in order to decrypt.[4]

Disadvantages of RSA for our algorithm:

Absence of Input Validation: This makes the code susceptible to fraudulent inputs or a variety of input mistakes since it does not thoroughly validate user inputs.

Limited Error Handling: Although the code has some basic error handling, not all potential error scenarios may be effectively handled by it.
Fixed Chunk Size: For encryption, the algorithm separates plaintext into fixed-size chunks, which might not be the best option for all kinds of communications. Depending on the content of the message, variable-length chunking may increase flexibility and efficiency.
Limited Comments and Documentation: Other developers may find it difficult to comprehend and maintain this code due to the deficiency of thorough comments and documentation.

Conclusion

The code illustrates how RSA encryption and decoding work in their most basic form.
It has features for creating RSA key pairs, prime number generation, message encryption, and message decryption.
But there are a few things to remember:
Since the key generation procedure now depends on random selection and primality testing, it might be made more efficient.
Given improvements in processing power, using tiny prime bit lengths (512 bits in this example) would not be adequate for robust security in practical applications.
For increased robustness, error handling and input validation might be strengthened.
All things considered, the code offers a useful foundation for comprehending RSA encryption and has room for improvement for real-world application in safe communication networks.

References

1. *W. Diffie and M. E. Hellman, New directions in cryptography, IEEE Trans. Inform. Theory, Nov. 1976,22: 644-654.*

2. *A technique for getting digital signatures and public-key cryptosystems by R.L. Rivest, A. Shamir, and L. Adleman, Commun. ACM, Feb. 1978, 21(2): 120-126. ACM Digital Library, 10.1145/359340.359342. https://dl.acm.org/doi/10.1145/359340.359342*

3. *The Advantages and Disadvantages of Different Encryption Algorithms Link website*

4. *RSA full form / unacademy https://unacademy.com/content/gate-cse-it/rsa-full-form/*

5. *Xin Zhou and Xiaofei Tang, "Research and implementation of RSA algorithm for encryption and decryption," Proceedings of 2011 6th International Forum on Strategic Technology, Harbin, Heilongjiang, 2011, pp. 1118-1121, doi: 10.1109/IFOST.2011.6021216. keywords: {Cryptography;Algorithm design and analysis;RSA algorithm;encryption;decryption},*
6. *N. M. S. Iswari, "Key generation algorithm design combination of RSA and ElGamal algorithm," 2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE), Yogyakarta, Indonesia, 2016, pp. 1-5, doi: 10.1109/ICITEED.2016.7863255. keywords: {Algorithm design and analysis;Encryption;Public key cryptography;Mathematical model;RSA;ElGamal;public key cryptography;combination;algorithm design},*