



NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY

(An Institution of National Importance under MoE, Govt. of India)

KARAIKAL – 609 609

NETWORK SECURITY

Assignment-1

Subject Code: CS1702

Semester: 6th Sem

Department: CSE

Submitted To: Dr. Narendran Rajagopalan

Submitted By: Ayon Biswas

Roll No: CS22B1065

RSA (Rivest–Shamir–Adleman) Algorithm

Introduction:

RSA is one of the first and most widely used **public-key cryptosystems** for secure data transmission. Unlike symmetric key encryption like DES, RSA is based on **asymmetric encryption**, meaning it uses two different keys: a **public key** for encryption and a **private key** for decryption. RSA's security relies on the computational difficulty of factoring large composite numbers (the product of two large primes).

RSA can encrypt data securely without sharing a private secret key, which makes it ideal for secure communications over open networks.

Encryption Process:

The RSA encryption process involves several steps:

- **Key Generation:** Two large prime numbers are selected randomly. Their product n becomes part of both the public and private keys.
- A number e is chosen such that it is relatively prime to $\phi(n)$ (Euler's Totient Function of n).
- The **public key** is the pair (e, n) .
- To encrypt a message, each character of the plaintext is converted into its ASCII value and raised to the power of e , modulo n :

$$C = M^e \bmod n$$

Where:

- M = Plaintext (numerical form)
- C = Ciphertext

Decryption Process:

The decryption process in RSA mirrors the encryption process but uses the private key (d, n):

- The private key exponent d is computed such that it is the modular multiplicative inverse of e modulo $\phi(n)$.
- Decryption is performed by raising the ciphertext to the power of d modulo n:

$$M = C^d \bmod n$$

The decrypted numerical values are then converted back to characters to recover the original message.

Code Implementation:

The **SymPy** Python library is used for mathematical operations like checking prime numbers (isprime) and finding modular inverses (mod_inverse).

Key Generation:

```
def rsa_key_generation():  
    p = generate_prime()  
    q = generate_prime()  
    n = p * q  
    phi_n = (p - 1) * (q - 1)  
  
    e = random.randint(2, phi_n - 1)
```

```

while gcd(e, phi_n) != 1:
    e = random.randint(2, phi_n - 1)

d = mod_inverse(e, phi_n)

return (p, q, n, phi_n, e, d)

```

This `rsa_key_generation()` function performs:

- Prime number generation for p and q
- Calculation of n and $\phi(n)$
- Selection of a suitable e
- Calculation of private exponent d

The output includes all values necessary for encryption and decryption.

Encryption Function:

```

def encrypt(message, e, n):
    numerical_message = [ord(char) for char in message]
    cipher_text = [pow(num, e, n) for num in numerical_message]
    return cipher_text

```

The `encrypt()` function:

- Converts each character into its ASCII integer value.
- Encrypts each integer using the public key (e, n).
- Returns a list of encrypted numbers.

Decryption Function:

```
def decrypt(cipher_text, d, n):  
    decrypted_numerical_message = [pow(num, d, n) for num in  
cipher_text]  
    decrypted_message = ''.join([chr(num) for num in  
decrypted_numerical_message])  
    return decrypted_message
```

The decrypt() function:

- Applies modular exponentiation using the private key (d, n).
- Converts the resulting numerical values back into their respective characters.
- Returns the original plaintext.

User Input Message:

```
message = input("Enter the message you want to encrypt: ")
```

The user is prompted to enter a message, which is stored in the message variable for encryption.

Key Generation, Encryption, and Decryption:

```
p, q, n, phi_n, e, d = rsa_key_generation()  
cipher_text = encrypt(message, e, n)
```

```
decrypted_message = decrypt(cipher_text, d, n)
```

```
print("\n--- RSA Encryption and Decryption ---")
```

```
print(f"Prime numbers: p = {p}, q = {q}")
```

```
print(f"Public Key: (e = {e}, n = {n})")
```

```
print(f"Private Key: (d = {d}, n = {n})")
```


```
print(f"Original Message: {message}")
```

```
print(f"Encrypted Message: {cipher_text}")
```

```
print(f"Decrypted Message: {decrypted_message}")
```

- Key generation happens first.
- The message is encrypted and then decrypted.
- All key parameters and message details are printed.

OUTPUT:



```
Enter the message you want to encrypt: This is secured message

--- RSA Encryption and Decryption ---
Prime numbers: p = 197, q = 211
Public Key: (e = 1957, n = 41567)
Private Key: (d = 10453, n = 41567)
Original Message: This is secured message
Encrypted Message: [8648, 18091, 36813, 23989, 19506, 36813, 23989, 19506, 23989, 16723, 31331, 15235, 18751, 16723, 41166, 19506, 11036, 16723, 23989, 23989, 20495, 12097, 16723]
Decrypted Message: This is secured message
```

Conclusion:

RSA provides a robust and secure way to encrypt data without the need for secure key exchange beforehand.

In this Python implementation:

- Random small prime numbers are used (for demonstration).

- Encryption and decryption are performed at the character level.
- SymPy simplifies the number theory operations like prime checking and modular inverses.

This code is ideal for understanding the conceptual working of RSA and can be expanded for more secure real-world applications by using larger primes and optimized encryption standards.