

# Emerging Blockchain Models for Digital Currencies

Name: Shaik Aarif

Regd No: 2200033144

Exp 2: Implementation of the RSA Encryption Algorithm with Key Generation.

## Description of the RSA Algorithm Implementation:

### 1. Purpose:

- Implements the RSA encryption and decryption process to securely encode and decode messages.
- 

## Steps in the Algorithm:

### 2. Key Components:

- **Public Key:** (e, n) used for encryption.
- **Private Key:** (d, n) used for decryption.

### 3. Prime Number Generation:

- `generate_prime_candidate(length)`: Generates a random odd number within the range specified by the bit length.
- `is_prime(n)`: Validates whether a number is prime using trial division up to the square root of the number.
- `generate_prime_number(length)`: Continually generates and validates random primes of the specified bit length.

### 4. Key Generation (`generate_keys`):

- Two distinct large prime numbers, p and q, are generated.
- The modulus n is calculated as  $n = p \times q$ .
- Euler's totient function is computed as  $\phi = (p-1) \times (q-1)$ .
- Public exponent e is chosen such that  $1 < e < \phi$  and  $\gcd(e, \phi) = 1$ .
- Private exponent d is calculated as the modular inverse of e modulo phi.

### 5. Encryption (`encrypt`):

- Converts the plaintext characters to their Unicode code points.
- Each code point is raised to the power of e (public exponent) modulo n.

- Produces the ciphertext as a list of integers.

#### 6. **Decryption (decrypt):**

- Each ciphertext integer is raised to the power of  $d$  (private exponent) modulo  $n$ .
  - Converts the resulting values back to characters, forming the original plaintext.
- 

### **Key Features:**

#### 7. **Security:**

- The security of RSA relies on the difficulty of factoring the product of two large prime numbers.
- Only the private key holder can decrypt the ciphertext.

#### 8. **Randomness:**

- Prime numbers  $p$  and  $q$  are randomly generated for each key pair.
- This ensures unique and secure keys for every execution.

#### 9. **Modular Arithmetic:**

- The algorithm heavily uses modular arithmetic for key generation, encryption, and decryption, which is efficient and secure.
- 

### **Example Walkthrough:**

#### • **Key Generation:**

- Random primes  $p$  and  $q$  are selected (e.g.,  $p = 211$ ,  $q = 223$  for an 8-bit size).
- $n = 211 \times 223 = 47053$ ,  $\phi = (211 - 1) \times (223 - 1) = 46620$ .
- $e$  is chosen (e.g.,  $e = 13$  such that  $\gcd(13, 46620) = 1$ ).
- $d$  is computed as  $e^{-1} \bmod \phi$  (e.g.,  $d = 35957$ ).

#### • **Encryption:**

- Message: "Hello Aarif".
- Convert each character to ciphertext:  $\text{cipher} = \text{ord}(\text{char})^e \bmod n$ .

#### • **Decryption:**

- Convert each ciphertext value back to plaintext:  $\text{plain} = \text{cipher}^d \bmod n$ .

### **Output:**

- Displays public and private keys.
- Shows the encrypted and decrypted message, verifying the correctness of the implementation.

## Code:

```
1. import random
2.
3. def gcd(a, b):
4.     while b != 0:
5.         a, b = b, a % b
6.     return a
7.
8. def mod_inverse(e, phi):
9.     d, x1, x2, y1 = 0, 0, 1, 1
10.    phi0 = phi
11.    while e > 0:
12.        temp1, temp2 = phi // e, phi % e
13.        x = x2 - temp1 * x1
14.        y = d - temp1 * y1
15.        phi, e = e, temp2
16.        x2, x1 = x1, x
17.        d, y1 = y1, y
18.    if phi == 1:
19.        return d + phi0
20.
21. def is_prime(n):
22.     if n <= 1:
23.         return False
24.     for i in range(2, int(n**0.5) + 1):
25.         if n % i == 0:
26.             return False
27.     return True
28.
29. def generate_prime_candidate(length):
30.     p = random.randrange(2**(length - 1), 2**length)
31.     if p % 2 == 0:
32.         p += 1
33.     return p
34.
35. def generate_prime_number(length=8):
36.     p = 4
37.     while not is_prime(p):
38.         p = generate_prime_candidate(length)
39.     return p
40.
41. def generate_keys(keysize=8):
42.     p = generate_prime_number(keysize)
43.     q = generate_prime_number(keysize)
44.     while p == q:
45.         q = generate_prime_number(keysize)
46.     n = p * q
47.     phi = (p - 1) * (q - 1)
48.
49.     e = random.randrange(1, phi)
50.     while gcd(e, phi) != 1:
51.         e = random.randrange(1, phi)
52.
53.     d = mod_inverse(e, phi)
54.
55.     return (e, n), (d, n)
56.
57. def encrypt(public_key, plaintext):
58.     e, n = public_key
59.     ciphertext = [pow(ord(char), e, n) for char in plaintext]
60.     return ciphertext
61.
62. def decrypt(private_key, ciphertext):
63.     d, n = private_key
64.     plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])
65.     return plaintext
66.
67. public_key, private_key = generate_keys(keysize=8)
```

```

68. print("Public key:", public_key)
69. print("Private key:", private_key)
70.
71. message = "Hello Aarif"
72. print("\nOriginal message:", message)
73.
74. ciphertext = encrypt(public_key, message)
75. print("\nEncrypted message:", ciphertext)
76.
77. decrypted_message = decrypt(private_key, ciphertext)
78. print("\nDecrypted message:", decrypted_message)
79.
80.

```

## Execution: (Screenshot)

The screenshot shows a Visual Studio Code editor window titled "EBMDC [Administrator]". The Explorer sidebar on the left shows a project structure with "EBMDC" containing "Exps" (with sub-files "RSA.py" and "SymEnc.py") and "README.md". The main editor displays the code in "RSA.py", which includes functions for generating prime candidates and numbers, and for encrypting and decrypting messages. The code is as follows:

```

28
29 def generate_prime_candidate(length):
30     p = random.randrange(2**(length - 1), 2**length)
31     if p % 2 == 0:
32         p += 1
33     return p
34
35 def generate_prime_number(length=8):
36     p = 4
37     while not is_prime(p):

```

The TERMINAL pane at the bottom shows the execution of the script. The command executed is `python RSA.py`. The output is as follows:

```

PS C:\Users\Administrator\Desktop\Emerging Blockchain\EBMDC\Exps> python RSA.py
Public key: (1321, 23711)
Private key: (29281, 23711)

Original message: Hello Aarif

Encrypted message: [6171, 15165, 22220, 22220, 21980, 20876, 8942, 18593, 9345, 14946, 19919]

Decrypted message: Hello Aarif
PS C:\Users\Administrator\Desktop\Emerging Blockchain\EBMDC\Exps>

```

The status bar at the bottom indicates the current file is "main\*", the cursor is at line 55, column 18, and the editor is using Python 3.13.0.