# 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).
    - $n=211\times223=47053$ , $\phi=(211-1)\times(223-1)=46620$.
    - 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: $cipher=ord(char)^{e} \bmod n$.
- **Decryption**:
    - Convert each ciphertext value back to plaintext: $plain=cipher^{d} \bmod$.

**Output:**

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

Code:

```
import random

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def mod_inverse(e, phi):
    d, x1, x2, y1 = 0, 0, 1, 1
    phi0 = phi
    while e > 0:
        temp1, temp2 = phi // e, phi % e
        x = x2 - temp1 * x1
        y = d - temp1 * y1
        phi, e = e, temp2
        x2, x1 = x1, x
        d, y1 = y1, y
    if phi == 1:
        return d + phi0

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def generate_prime_candidate(length):
    p = random.randrange(2**(length - 1), 2**length)
    if p % 2 == 0:
        p += 1
    return p

def generate_prime_number(length=8):
    p = 4
    while not is_prime(p):
        p = generate_prime_candidate(length)
    return p

def generate_keys(keysize=8):
    p = generate_prime_number(keysize)
    q = generate_prime_number(keysize)
    while p == q:
        q = generate_prime_number(keysize)
    n = p * q
    phi = (p - 1) * (q - 1)

    e = random.randrange(1, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(1, phi)

    d = mod_inverse(e, phi)

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

def encrypt(public_key, plaintext):
    e, n = public_key
    ciphertext = [pow(ord(char), e, n) for char in plaintext]
    return ciphertext

def decrypt(private_key, ciphertext):
    d, n = private_key
    plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])
    return plaintext

public_key, private_key = generate_keys(keysize=8)
```

```python
print("Public key:", public_key)
print("Private key:", private_key)

message = "Hello Aarif"
print("\nOriginal message:", message)

ciphertext = encrypt(public_key, message)
print("\nEncrypted message:", ciphertext)

decrypted_message = decrypt(private_key, ciphertext)
print("\nDecrypted message:", decrypted_message)

```

Execution: (Screenshot)