Implementation of RSA Algorithm

Network Security Assignment - CS1702

Submitted By
Sandeep U (CS22B1050)

To

Dr. Narendran Rajagopalan
Associate Professor
Department of Computer Science and Engineering
National Institute of Technology Puducherry
Karaikal – 609609

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY
KARAIKAL – 609 609
April 2025

Introduction to RSA:

RSA (Rivest–Shamir–Adleman) is among the earliest public-key cryptographic systems and remains widely used for secure communication. In this type of system, the encryption key is public, while the decryption key is private and confidential. RSA relies on the computational difficulty of factoring large composite numbers — a challenge known as the "factoring problem" — to maintain its security. The name "RSA" comes from its inventors: Ron Rivest, Adi Shamir, and Leonard Adleman, who introduced the algorithm in 1977. Interestingly, a similar method was developed earlier in 1973 by Clifford Cocks, a British mathematician at GCHQ, but it remained classified until 1997.

An RSA user generates and shares a public key derived from two large secret prime numbers and an auxiliary value. While anyone can encrypt messages using this public key, only someone with knowledge of the original prime numbers can decrypt them. The difficulty of breaking RSA encryption is known as the "RSA problem," though it's still uncertain whether this problem is exactly equivalent to the factoring problem. So far, no effective attacks have been made public against RSA when strong, properly-sized keys are used.

Due to its slower performance, RSA is not typically used for direct encryption of large amounts of data. Instead, it is often employed to securely exchange keys for symmetric encryption systems, which are much faster for bulk data processing.

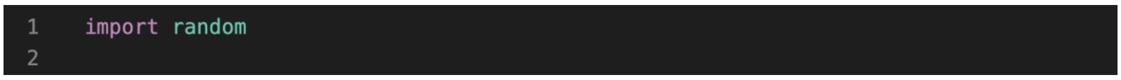
RSA Algorithm:

- Pick two large primes p ,q.
- Compute $n= pq$ and $\phi(n)=\mathrm{lcm}(p-1,q-1)$
- Choose a public key e such that $1<e<\phi(n)$ and $\gcd(e, \phi(n)) =1$
- Calculate d such that $de\equiv 1 \pmod{\phi(n)}$
- Let the message key be : m
- Encrypt: $c \equiv m^{**}e \pmod n$

- Decrypt: m ≡ c**d (mod n)

RSA Code and Explanation:

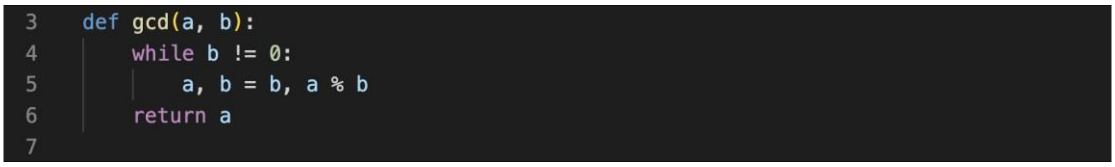
INPUT:



```python
import random

```

--Used to generate random prime numbers for key generation.

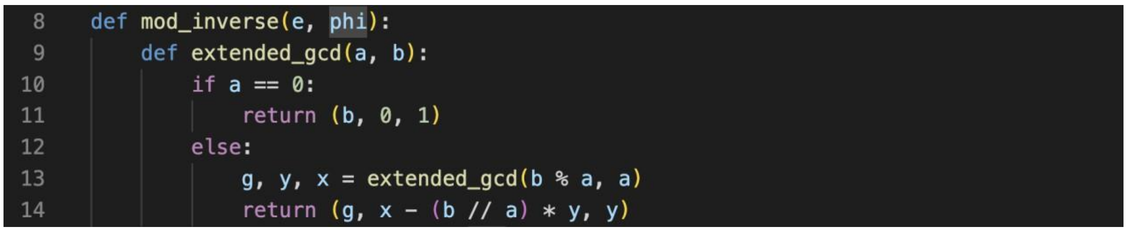


```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

```

--Implements Euclidean Algorithm to find the greatest common divisor of a and b.

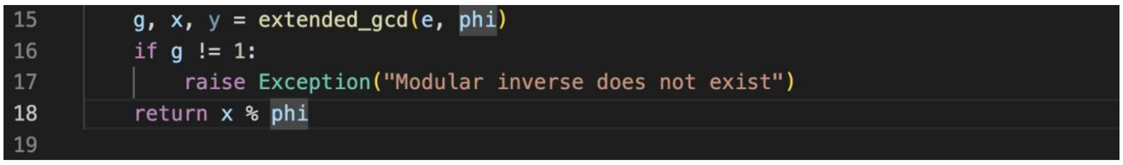
-- Used to ensure e is coprime to ϕ(n).



```python
def mod_inverse(e, phi):
    def extended_gcd(a, b):
        if a == 0:
            return (b, 0, 1)
        else:
            g, y, x = extended_gcd(b % a, a)
            return (g, x - (b // a) * y, y)
```

--Inner function extended_gcd: Recursively applies Extended Euclidean Algorithm.

--Finds integers x and y such that: a*x + b*y = gcd(a,b).

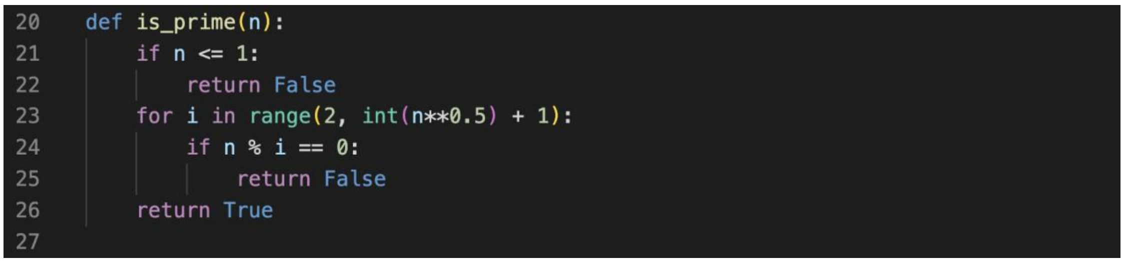


```python
    g, x, y = extended_gcd(e, phi)
    if g != 1:
        raise Exception("Modular inverse does not exist")
    return x % phi

```

--If gcd(e,phi)=1, returns x mod ϕ(n) as the modular inverse d.

--Else, raises an error.

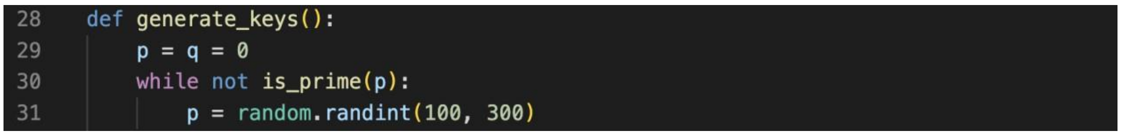


```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

```

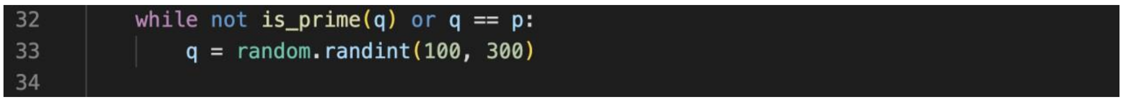
--Checks if a number is prime using trial division.

--Needed to select good values for p and q.



```python
def generate_keys():
    p = q = 0
    while not is_prime(p):
        p = random.randint(100, 300)
```

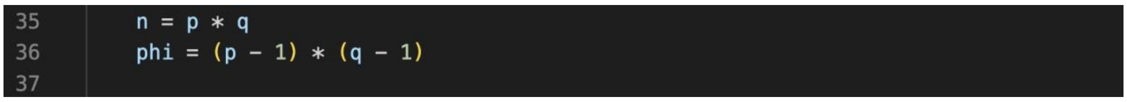
--Keeps generating random numbers until it finds a prime p.



```python
    while not is_prime(q) or q == p:
        q = random.randint(100, 300)

```

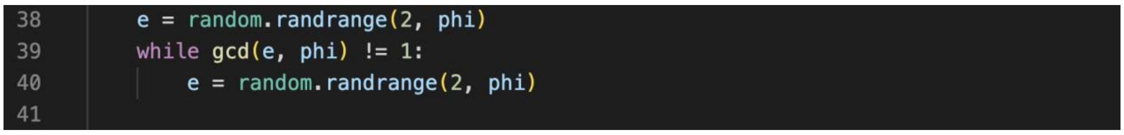
--Same for q, ensuring p ≠ q.



```python
    n = p * q
    phi = (p - 1) * (q - 1)

```

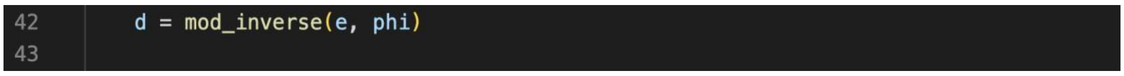
--Computes modulus n and Euler's totient ϕ(n).



```python
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)

```

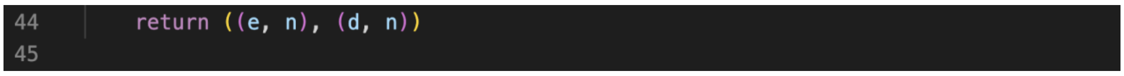
--Selects e such that gcd(e, ϕ(n)) = 1 (ensures e has a modular inverse).



```python
    d = mod_inverse(e, phi)

```

--Computes d = e^{-1} mod ϕ(n)

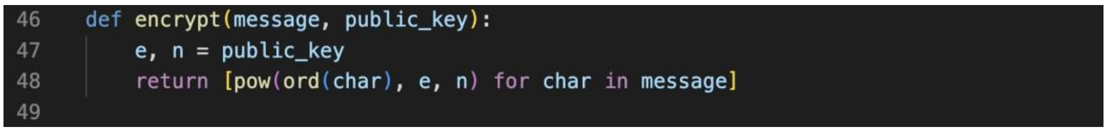


```python
    return ((e, n), (d, n))

```

--Returns:

- **Public key**: (e,n)
- **Private key**: (d,n)



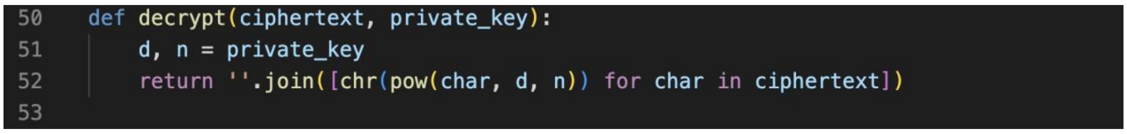
```python
def encrypt(message, public_key):
    e, n = public_key
    return [pow(ord(char), e, n) for char in message]

```

--For each character:

- ord(char) : Convert to ASCII.
- pow(base, exp, mod): Efficiently compute char^e mod n.

––Returns list of encrypted integers.



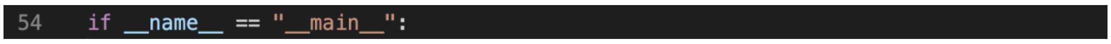
```python
def decrypt(ciphertext, private_key):
    d, n = private_key
    return ''.join([chr(pow(char, d, n)) for char in ciphertext])

```

--For each encrypted number:

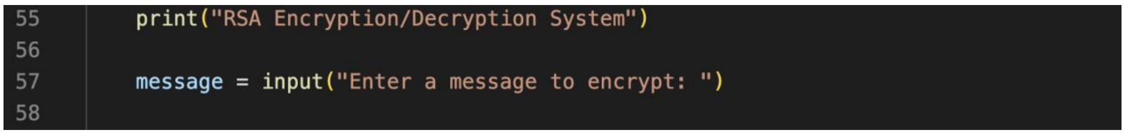
- Computes char^d mod n.
- Converts back to character with chr().

--Returns the original message.



```python
if __name__ == "__main__":
```

--Ensures the code only runs when the script is executed directly.

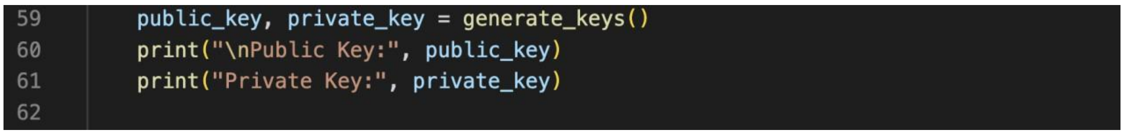


```python
    print("RSA Encryption/Decryption System")

    message = input("Enter a message to encrypt: ")

```

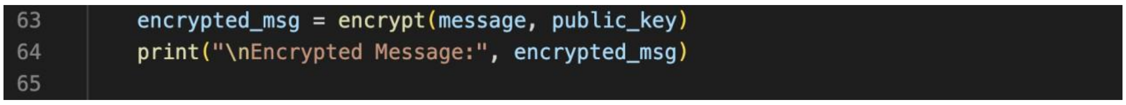
--Gets input message from user.



```python
    public_key, private_key = generate_keys()
    print("\nPublic Key:", public_key)
    print("Private Key:", private_key)

```

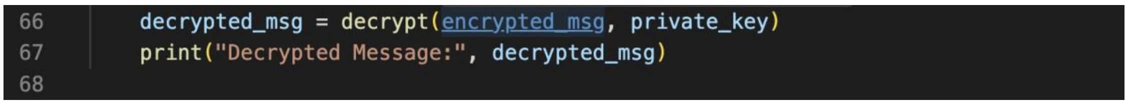
--Generates and displays keys.



```python
    encrypted_msg = encrypt(message, public_key)
    print("\nEncrypted Message:", encrypted_msg)

```

--Encrypts and prints the ciphertext.

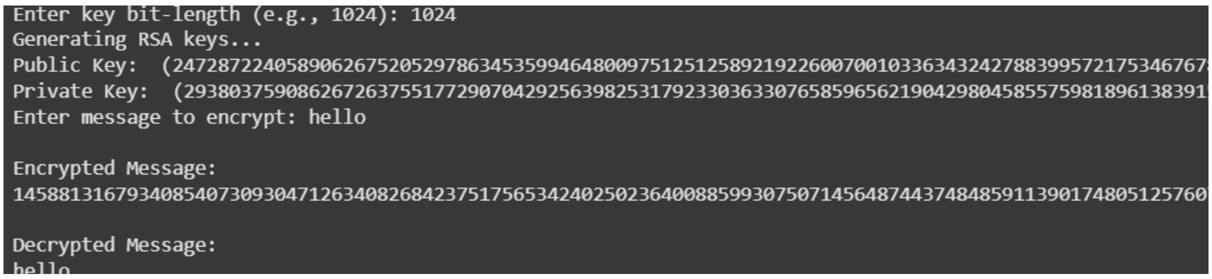


```python
    decrypted_msg = decrypt(encrypted_msg, private_key)
    print("Decrypted Message:", decrypted_msg)

```

--Decrypts and displays the original message.

OUTPUT:



```
Enter key bit-length (e.g., 1024): 1024
Generating RSA keys...
Public Key:  (247287224058906267520529786345359946480097512512589219226007001033634324278839957217534676
Private Key:  (29380375908626726375517729070429256398253179233036330765859656219042980458557598189613839
Enter message to encrypt: hello

Encrypted Message:
145881316793408540730930471263408268423751756534240250236400885993075071456487443748485911390174805125760

Decrypted Message:
hello
```