Implementation of RSA Algorithm

Network Security Assignment - CS1072

Submitted By

Hamsashree S (CS22B1016)

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

Public-key cryptography, or asymmetric cryptography, is a cryptographic system that uses pairs of keys: public keys, which may be disseminated widely, and private keys,which are known only to the owner. The generation of such keys depends on cryptographic algorithms based on mathematical problems to produce one-way functions. Effective security only requires keeping the private key private; the public key can be openly distributed without compromising security.

In such a system, any person can encrypt a message using the receiver's public key, but that encrypted message can only be decrypted with the receiver's private key.

RSA Algorithm

RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and distinct from the decryption key which is kept secret (private). In RSA, this asymmetry is based on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem". The acronym RSA is the initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman, who publicly described the algorithm in 1977. Clifford Cocks, an English mathematician working for the British intelligence agency Government Communications Headquarters (GCHQ), had developed an equivalent system in 1973, which was not declassified until 1997.

A user of RSA creates and then publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but only someone with knowledge of the prime numbers can decode the message. Breaking RSA encryption is known as the RSA problem. Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used.

RSA is a relatively slow algorithm, and because of this, it is less commonly used to directly encrypt user data. More often, RSA passes encrypted shared keys for symmetric key cryptography which in turn can perform bulk encryption-decryption operations at much higher speed.

RSA Algorithm

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

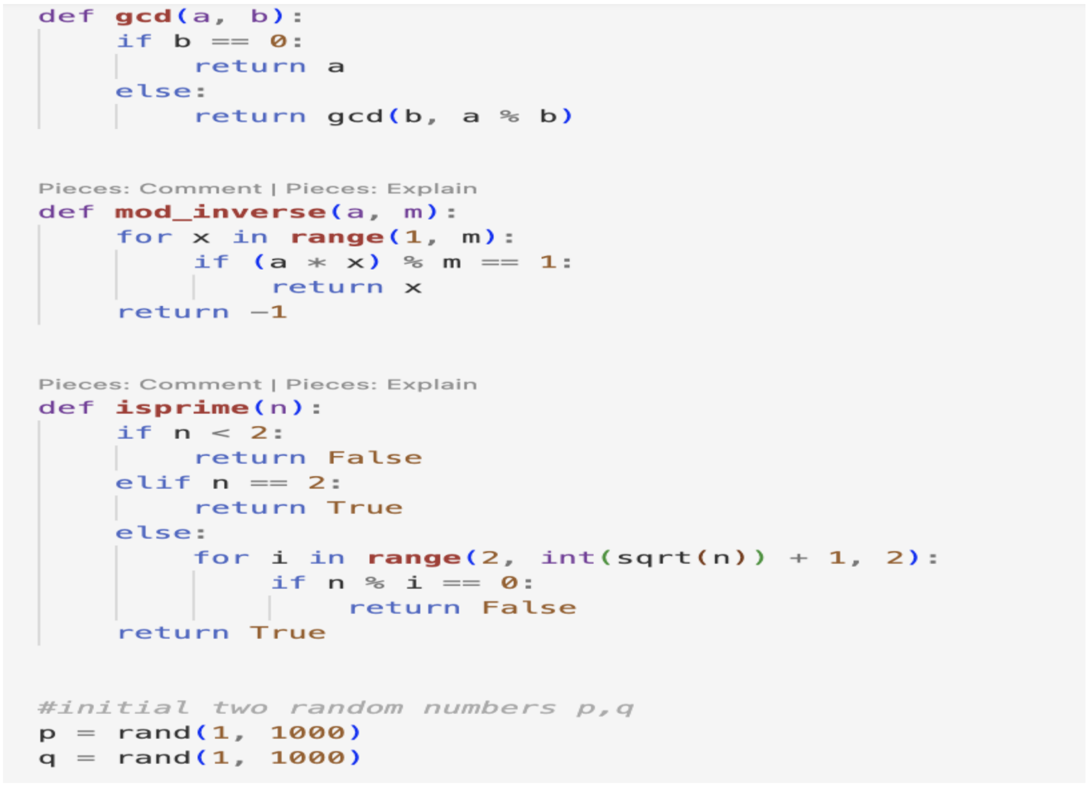
- Decrypt: m ≡ c**d (mod n)

```
from math import sqrt
#required for the sqrt() function, if you want to avoid doing **0.5
import random
#required for randrange
from random import randint as rand
```

Imports and Helper Functions

The program first imports necessary libraries:

- math.sqrt is used to compute square roots.
- random is used to generate random numbers.
- randint is imported with an alias rand, mimicking C++'s rand() function.

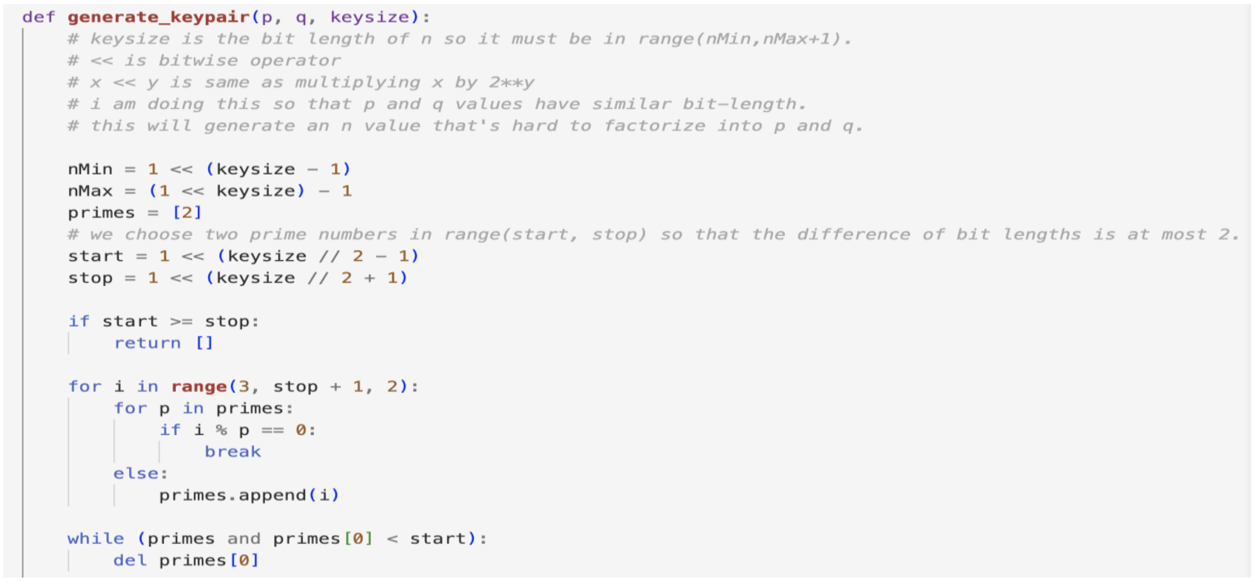


```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)


Pieces: Comment | Pieces: Explain
def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return -1


Pieces: Comment | Pieces: Explain
def isprime(n):
    if n < 2:
        return False
    elif n == 2:
        return True
    else:
        for i in range(2, int(sqrt(n)) + 1, 2):
            if n % i == 0:
                return False
    return True


#initial two random numbers p,q
p = rand(1, 1000)
q = rand(1, 1000)
```

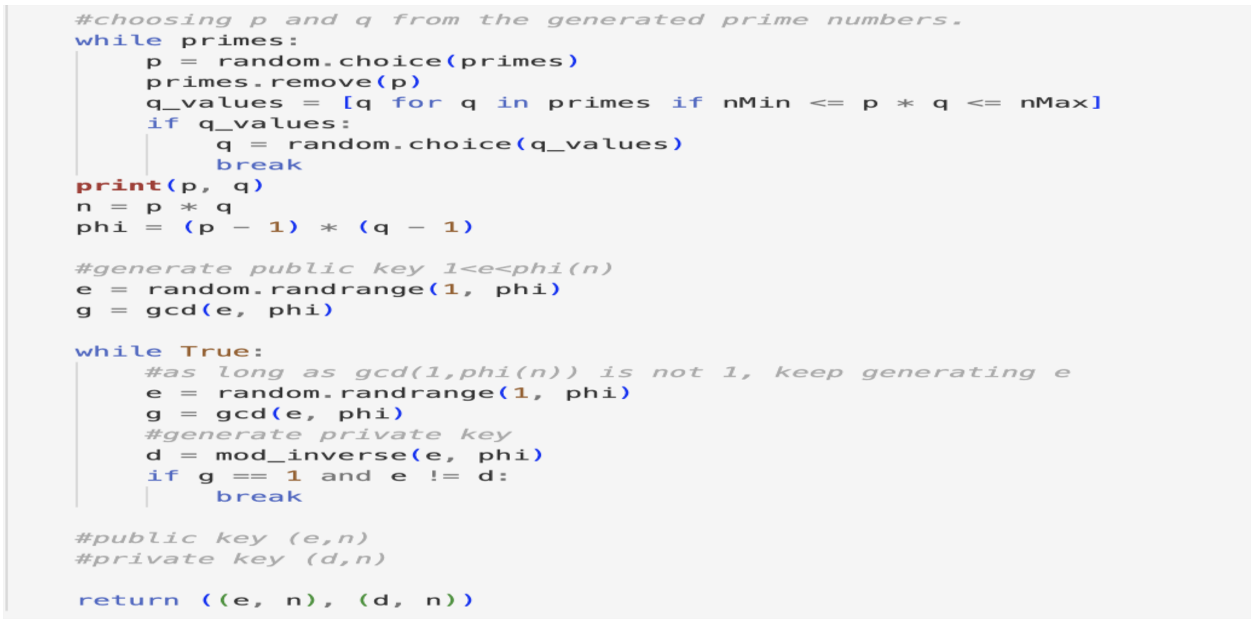
Several helper functions are defined:

1. **gcd(a, b)**: Computes the greatest common divisor (GCD) using recursion.
2. **mod_inverse(a, m)**: Finds the modular inverse of a modulo m. This is used to compute the private key.

3. **isprime(n)**: Checks whether a given number n is prime. It does this by checking divisibility up to sqrt(n), skipping even numbers



```python
def generate_keypair(p, q, keysize):
    # keysize is the bit length of n so it must be in range(nMin,nMax+1).
    # << is bitwise operator
    # x << y is same as multiplying x by 2**y
    # i am doing this so that p and q values have similar bit-length.
    # this will generate an n value that's hard to factorize into p and q.

    nMin = 1 << (keysize - 1)
    nMax = (1 << keysize) - 1
    primes = [2]
    # we choose two prime numbers in range(start, stop) so that the difference of bit lengths is at most 2.
    start = 1 << (keysize // 2 - 1)
    stop = 1 << (keysize // 2 + 1)

    if start >= stop:
        return []

    for i in range(3, stop + 1, 2):
        for p in primes:
            if i % p == 0:
                break
        else:
            primes.append(i)

    while (primes and primes[0] < start):
        del primes[0]
```



```python
    #choosing p and q from the generated prime numbers.
    while primes:
        p = random.choice(primes)
        primes.remove(p)
        q_values = [q for q in primes if nMin <= p * q <= nMax]
        if q_values:
            q = random.choice(q_values)
            break
    print(p, q)
    n = p * q
    phi = (p - 1) * (q - 1)

    #generate public key 1<e<phi(n)
    e = random.randrange(1, phi)
    g = gcd(e, phi)

    while True:
        #as long as gcd(1,phi(n)) is not 1, keep generating e
        e = random.randrange(1, phi)
        g = gcd(e, phi)
        #generate private key
        d = mod_inverse(e, phi)
        if g == 1 and e != d:
            break

    #public key (e,n)
    #private key (d,n)

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

Generating Prime Numbers

Initially, two random numbers p and q are selected. These numbers should ideally be prime, but the program does not check this explicitly before use.

The function **generate_keypair(p, q, keysize)** is responsible for generating RSA key pairs:

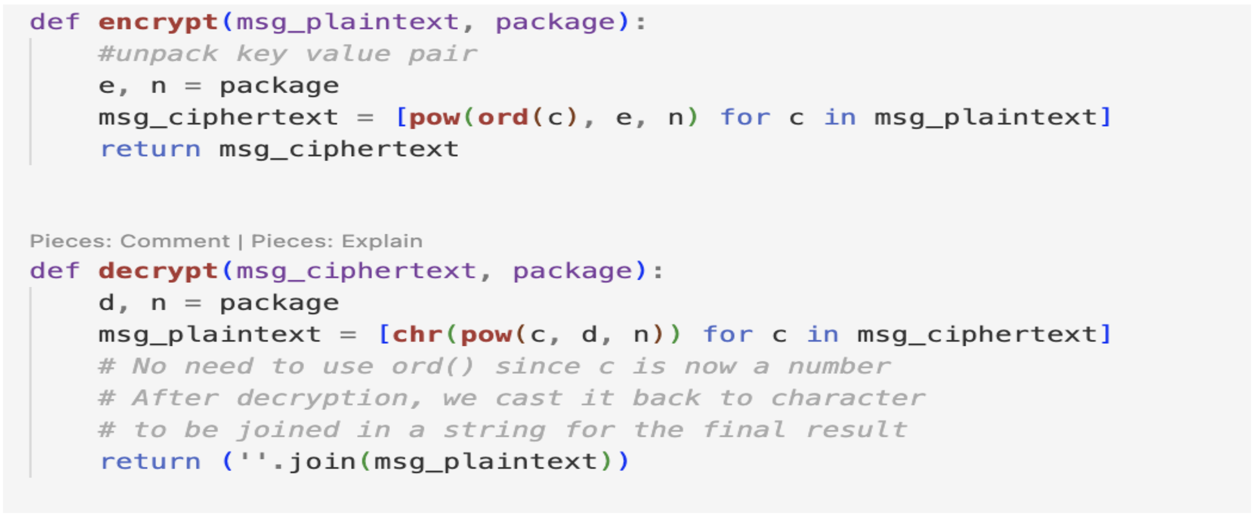
- It first defines the valid range for n = p * q based on the given key size.
- A list of prime numbers is generated in the specified range.
- It selects two prime numbers, p and q, such that p * q falls within the valid range.

Computing RSA Key Components

After selecting p and q, the function computes:

1. **n = p * q**: The modulus used in encryption and decryption.
2. **phi = (p - 1) * (q - 1)**: Euler's totient function, which is crucial for key generation.
3. **Selecting e (Public Exponent)**:
   - e is chosen randomly such that 1 < e < phi and gcd(e, phi) = 1.
4. **Computing d (Private Exponent)**:
   - d is the modular inverse of e modulo phi, ensuring e * d ≡ 1 (mod phi).

The function returns a public key (e, n) and a private key (d, n).



```python
def encrypt(msg_plaintext, package):
    #unpack key value pair
    e, n = package
    msg_ciphertext = [pow(ord(c), e, n) for c in msg_plaintext]
    return msg_ciphertext
```

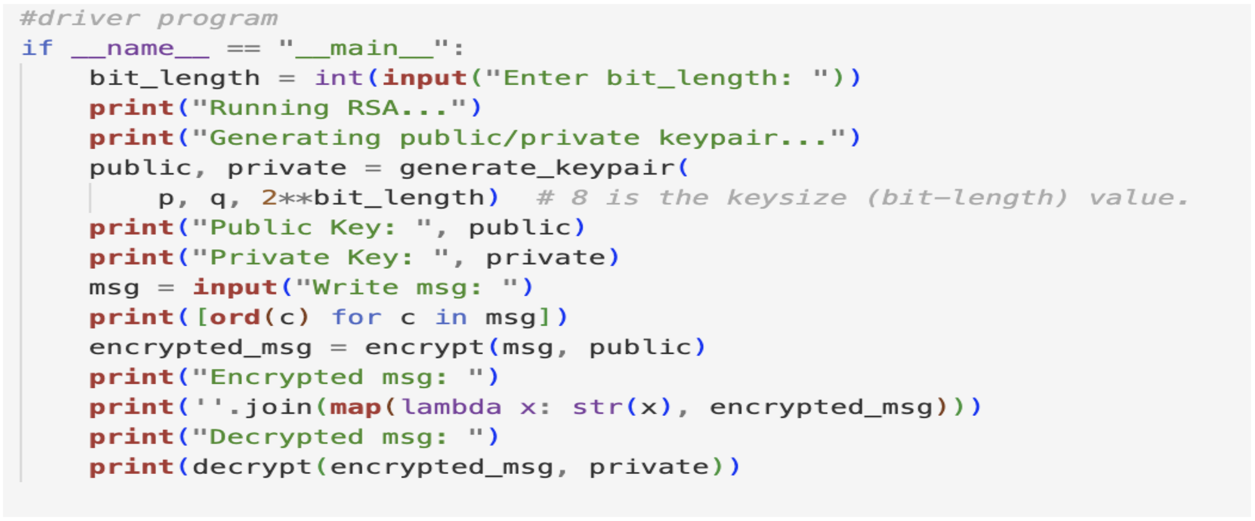
```python
def decrypt(msg_ciphertext, package):
    d, n = package
    msg_plaintext = [chr(pow(c, d, n)) for c in msg_ciphertext]
    # No need to use ord() since c is now a number
    # After decryption, we cast it back to character
    # to be joined in a string for the final result
    return (''.join(msg_plaintext))
```

Encryption and Decryption

Two functions handle message encryption and decryption:

1. **encrypt(msg_plaintext, package)**:
   - Uses the public key (e, n) to encrypt each character in the message using pow(ord(c), e, n).

    - ord(c) converts a character into its ASCII equivalent before exponentiation.
    - The result is a list of integers representing the encrypted message.
2. **decrypt(msg_ciphertext, package)**:
    - Uses the private key (d, n) to decrypt the message using pow(c, d, n).
    - The result is converted back to characters and joined into a string.



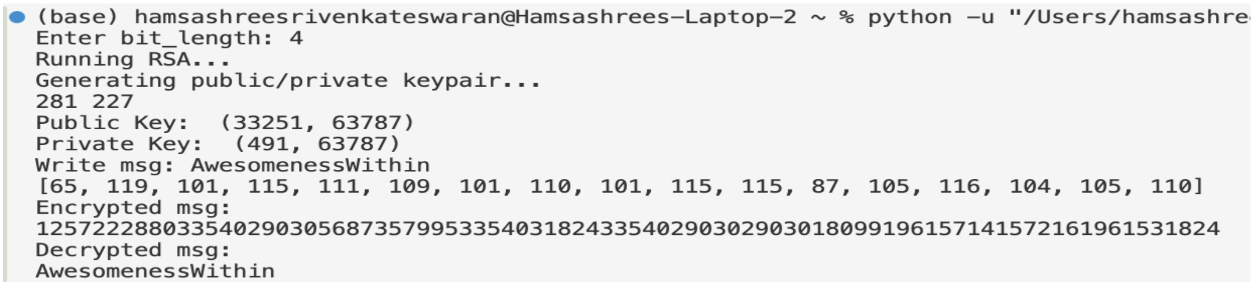
```python
#driver program
if __name__ == "__main__":
    bit_length = int(input("Enter bit_length: "))
    print("Running RSA...")
    print("Generating public/private keypair...")
    public, private = generate_keypair(
        p, q, 2**bit_length)  # 8 is the keysize (bit-length) value.
    print("Public Key: ", public)
    print("Private Key: ", private)
    msg = input("Write msg: ")
    print([ord(c) for c in msg])
    encrypted_msg = encrypt(msg, public)
    print("Encrypted msg: ")
    print(''.join(map(lambda x: str(x), encrypted_msg)))
    print("Decrypted msg: ")
    print(decrypt(encrypted_msg, private))
```

Driver Code

The program starts execution in the if __name__ == "__main__" block:

1. It asks for a bit length from the user.
2. Calls generate_keypair to create the public and private keys.
3. Prints the keys and asks the user for a message.
4. Encrypts the message and displays the encrypted form.
5. Decrypts the encrypted message and prints the original text.

OUTPUT:



```
● (base) hamsashreesrivenkateswaran@Hamsashrees-Laptop-2 ~ % python -u "/Users/hamsashre
  Enter bit_length: 4
  Running RSA...
  Generating public/private keypair...
  281 227
  Public Key:  (33251, 63787)
  Private Key:  (491, 63787)
  Write msg: AwesomenessWithin
  [65, 119, 101, 115, 111, 109, 101, 110, 101, 115, 115, 87, 105, 116, 104, 105, 110]
  Encrypted msg:
  125722288033540290305687357995335403182433540290302903018099196157141572161961531824
  Decrypted msg:
  AwesomenessWithin
```