Project Report RSA Algorithm Implementation
ECE 543

SAMMAN CHOUHAN A20561414

RSA Algorithm Implementation

The RSA algorithm, named after Ron Rivest, Adi Shamir, and Leonard Adleman who first publicly described it in 1978, remains one of the most widely used public-key cryptosystems for secure data transmission. Unlike symmetric key algorithms, which use the same key for both encryption and decryption, RSA utilizes a pair of keys—a public key for encryption and a private key for decryption. This duality allows users to distribute their public keys openly while keeping their private keys secret. The strength and security of the RSA algorithm lie in the practical difficulty of factoring the product of two large prime numbers, the "factoring problem". Our project focuses on implementing the RSA algorithm from scratch, which includes generating large prime numbers, testing their primality, and constructing cryptographic keys from these primes. The project is divided into two main parts: The generation and validation of prime numbers, which are fundamental to all subsequent cryptographic operations. The core RSA functionalities of key generation, encryption, and decryption.

RSA Algorithm

The RSA algorithm is a public-key cryptosystem algorithm that was publicly described by Ron Rivest, Adi Shamir, and Leonard Adleman in 1978. This algorithm is one of the most commonly utilized methods of ensuring the security of data transmission worldwide. RSA cryptosystems are unique in terms of operation. Also, they are designed to work with several keys simultaneously, a public and a private key. The public key is designed to encrypt data, and a private key is designed to decrypt it. A public key can be shared and transferred to a recipient through distributed networks, while a private one is accessible only to the recipient. The strength of RSA lies in the practical impossibility of the factorization of the product of two large prime numbers, i.e., the “factoring problem”. For that reason, the generation and validation of cryptographic prime numbers is fundamental to the RSA algorithm. The implementation of the RSA algorithm consists of two core parts:

- Generation and validation of prime numbers that is necessary for any cryptographic function.
- RSA functionality, which is the generation of keys, encryption, and decryption of messages.

The project plan consisted of two major areas, which include prime number operations and the RSA key management and associated data encryption/decryption processes. Our plan involved the following major activity areas: for prime numbers, primality testing, and generating new prime numbers. Concerning the RSA algorithm, our activities were key generation and encryption and decryption. The approach developed in the code was as follows:

For prime number activities:

Primality Testing: We used the Miller-Rabin test, a deterministic primality check that provided a relatively good compromise between the effort to implement and the actual performance for non-prime numbers.
Prime Number Generation: We utilized a random number generator combined with primality testing to generate random prime numbers within a given number of bits.
For the RSA algorithm approach:

Key Generation: This was based on two large prime numbers, including modulus calculation based on primality; totient calculation based on the number of primes and its prime factors; and calculating the public and private exponents.
Encryption and Decryption: Data encryption and decryption used modular exponentiation with separate functions for each operation.

Implementation Details
This section outlines the specific implementation strategies for each of the five core programs of the RSA algorithm project: **primecheck**, **primegen**, **keygen**, **encrypt**, and **decrypt**.
1. Prime Number Testing (primecheck)
The **primecheck** program is designed to determine whether a given number is prime. We opted for the Miller-Rabin primality test due to its probabilistic nature and efficiency with large numbers, crucial for cryptographic applications.
Algorithm:

- **Input:** An integer nn.
- **Output: True** if nn is likely prime, **False** otherwise.
- Convert nn into $2r \cdot d+12r \cdot d+1$ form, where dd is odd.
- For a predetermined number of trials, pick a random base aa and test if $ad \equiv 1ad \equiv 1$ (mod nn) or $ad \cdot 2j \equiv -1ad \cdot 2j \equiv -1$ (mod nn) for any jj from 0 to $r-1r-1$.
- If any of these conditions hold for each trial, nn is likely prime.

2. Prime Number Generation (primegen)
The **primegen** program generates a large prime number suitable for RSA encryption. It uses the primality testing function from **primecheck** to validate potential primes.
Algorithm:

- **Input:** A desired bit length ll.
- **Output:** A prime number with ll bits.
- Generate a random odd number of ll bits.
- Use the Miller-Rabin test to verify primality.
- If the number is not prime, increment and test again until a prime is found.

3. Key Generation (keygen)
keygen is tasked with generating RSA key pairs based on two large prime numbers.
Algorithm:

- **Inputs:** Two prime numbers pp and qq.
- **Outputs:** Public key (n, e) and private key (n, d).
- Compute $n=p\cdot qn=p\cdot q$ and $\phi(n)=(p-1)\cdot(q-1)\phi(n)=(p-1)\cdot(q-1)$.
- Choose an encryption exponent ee that is coprime with $\phi(n)\phi(n)$.
- Calculate the decryption exponent dd as the modular inverse of ee modulo $\phi(n)\phi(n)$.

4. Encryption (encrypt)

The **encrypt** function encrypts a plaintext character using the RSA public key.

Algorithm:

- **Inputs:** Public key components nn and ee, and a plaintext message mm.
- **Output:** Ciphertext cc.
- Compute the ciphertext as $c=me\text{mod }nc=me\text{mod}n$.

5. Decryption (decrypt)

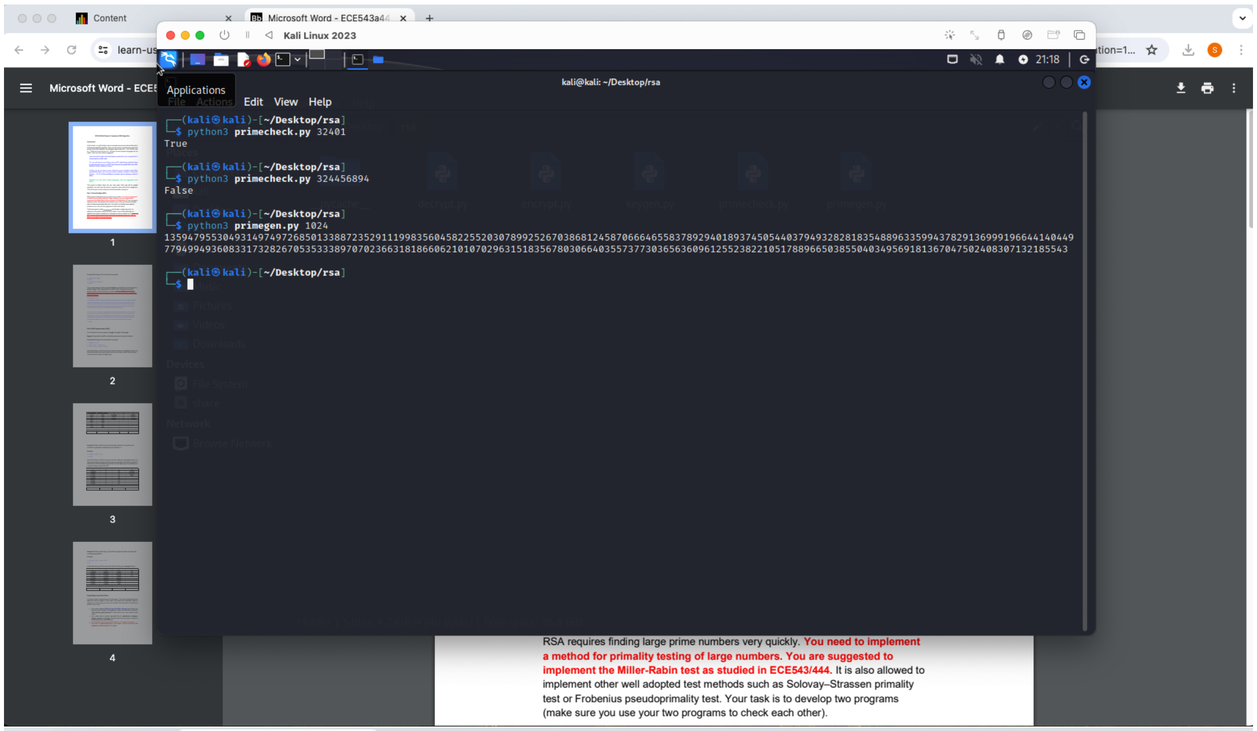
decrypt reverses the encryption process, recovering the original plaintext from the ciphertext using the private key.

Algorithm:

- **Inputs:** Private key components nn and dd, and ciphertext cc.
- **Output:** Plaintext mm.
- Compute the plaintext as $m=cd\text{mod }nm=cd\text{mod}n$.

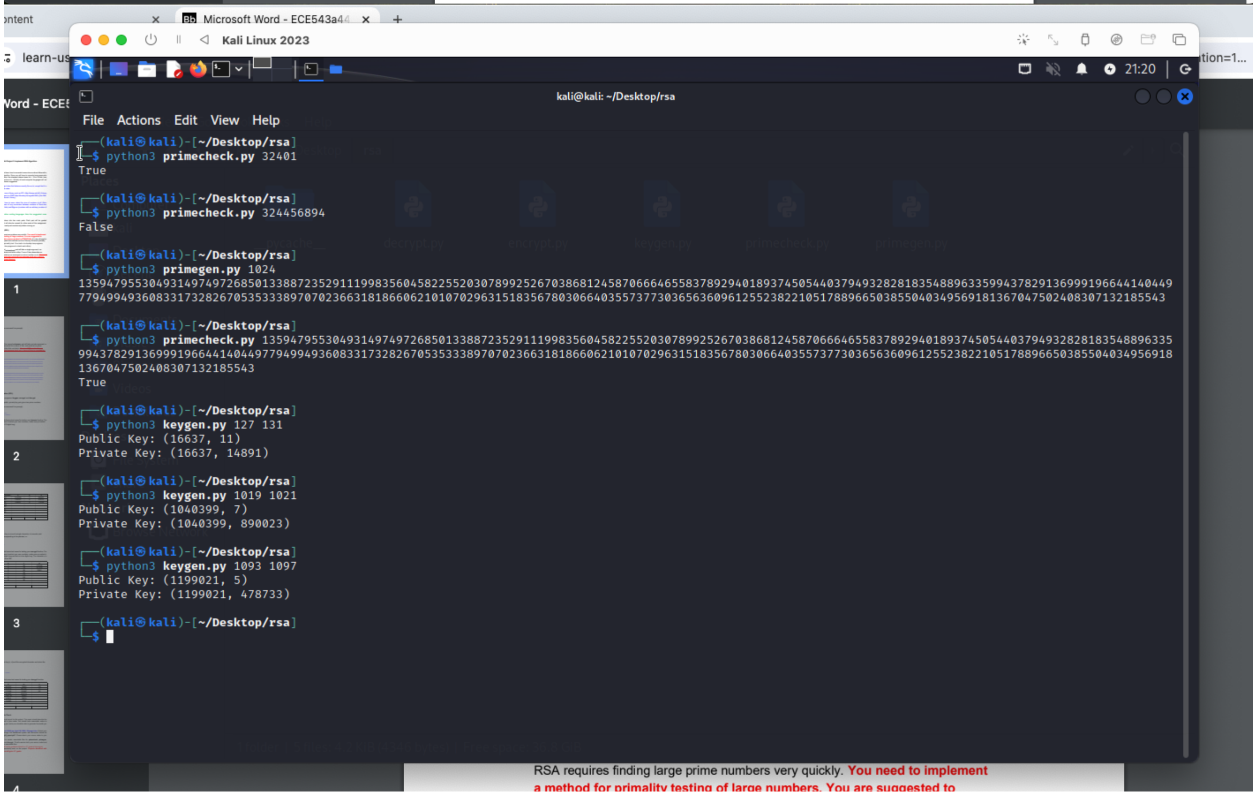
Results

This section should demonstrate the programs in action using various test cases



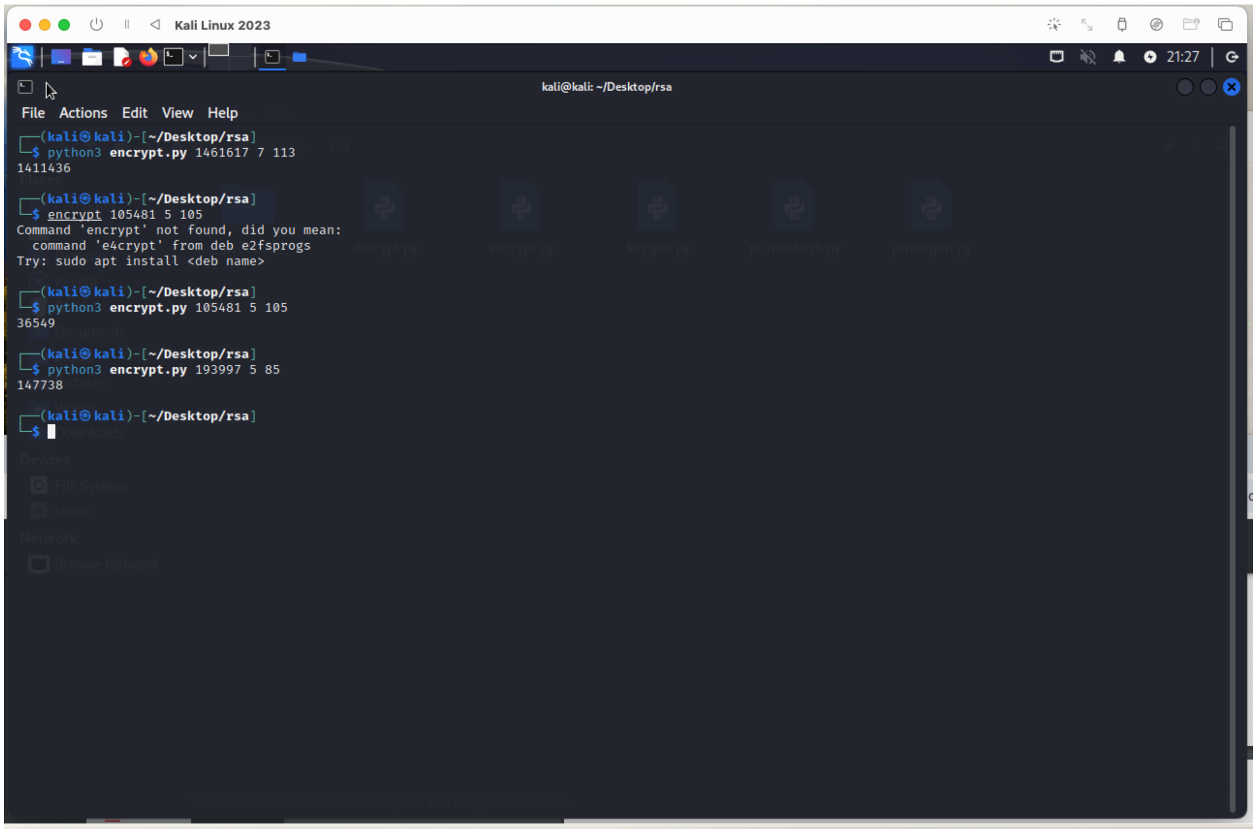
```
┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primecheck.py 32401
True

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primecheck.py 324456894
False

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primegen.py 1024
13594795530493149749726850133887235291119983560458225520307899252670386812458706664655837892940189374505440379493282818354889633599437829136999196644140449
7794994936083317328267053533389707023663181866062101070296315183567803066403557377303656360961255238221051788966503855040349569181367047502408307132185543

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ 
```

RSA requires finding large prime numbers very quickly. **You need to implement a method for primality testing of large numbers. You are suggested to implement the Miller-Rabin test as studied in ECE543/444.** It is also allowed to implement other well adopted test methods such as Solovay–Strassen primality test or Frobenius pseudoprimality test. Your task is to develop two programs (make sure you use your two programs to check each other).

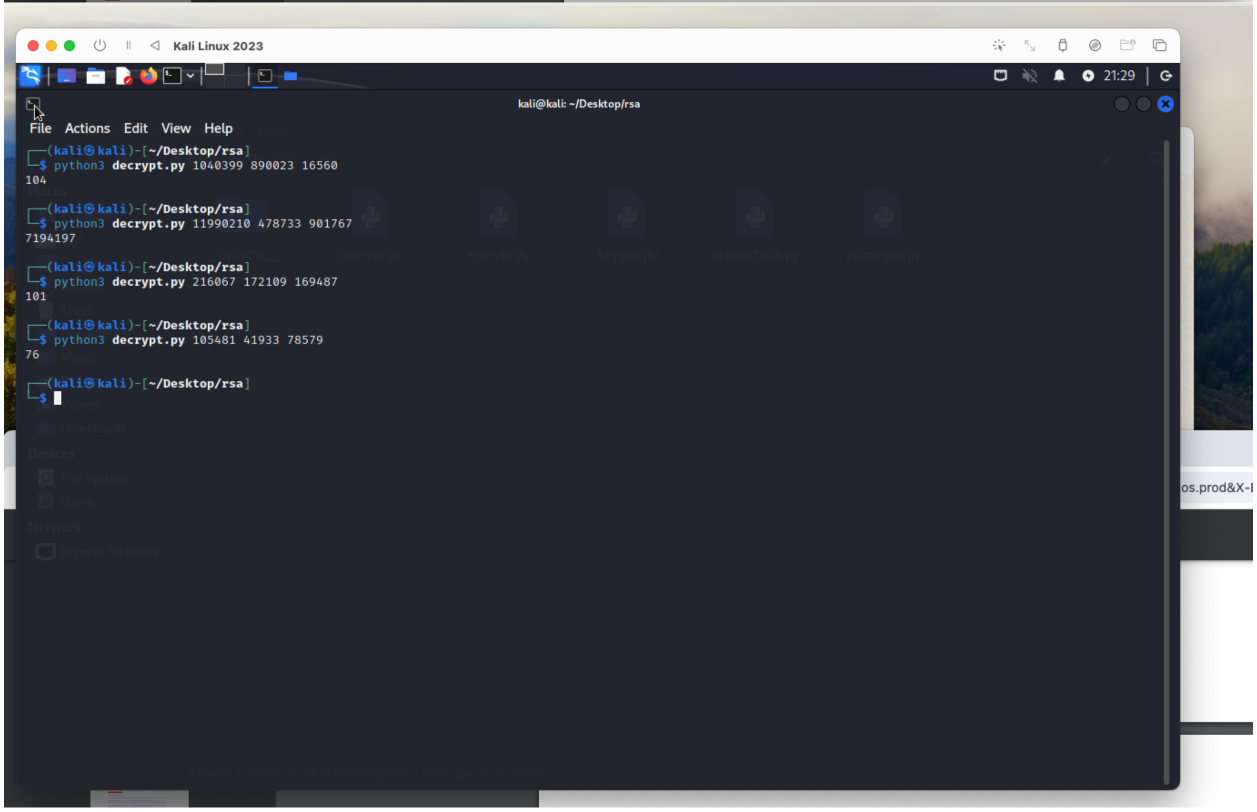


```
┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primecheck.py 32401
True

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primecheck.py 324456894
False

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primegen.py 1024
13594795530493149749726850133887235291119983560458225520307899252670386812458706664655837892940189374505440379493282818354889633599437829136999196644140449
7794994936083317328267053533389707023663181866062101070296315183567803066403557377303656360961255238221051788966503855040349569181367047502408307132185543

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 primecheck.py 13594795530493149749726850133887235291119983560458225520307899252670386812458706664655837892940189374505440379493282818354889633599437829136999196644140449779499493608331732826705353338970702366318186606210107029631518356780306640355737730365636096125523822105178896650385504034956918
13670475024083071321855543
True

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 keygen.py 127 131
Public Key: (16637, 11)
Private Key: (16637, 14891)

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 keygen.py 1019 1021
Public Key: (1040399, 7)
Private Key: (1040399, 890023)

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 keygen.py 1093 1097
Public Key: (1199021, 5)
Private Key: (1199021, 478733)

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ 
```

RSA requires finding large prime numbers very quickly. **You need to implement a method for primality testing of large numbers. You are suggested to**



```
┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 encrypt.py 1461617 7 113
1411436

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ encrypt 105481 5 105
Command 'encrypt' not found, did you mean:
  command 'e4crypt' from deb e2fsprogs
Try: sudo apt install <deb name>

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 encrypt.py 105481 5 105
36549

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 encrypt.py 193997 5 85
147738

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ 
```



```
┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 decrypt.py 1040399 890023 16560
104

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 decrypt.py 11990210 478733 901767
7194197

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 decrypt.py 216067 172109 169487
101

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ python3 decrypt.py 105481 41933 78579
76

┌──(kali㉿kali)-[~/Desktop/rsa]
└─$ 
```

Primegen.py

```python
import random
import sys
from primecheck import is_prime

def generate_large_prime(bits):
    # Reuse the is_prime function from above, ensure it's defined in this script or
imported
    def is_probable_prime(n, k=5):
        return is_prime(n, k)

    p = random.getrandbits(bits)
    p |= (1 << bits - 1) | 1  # Ensure it's odd and has the exact number of bits

    while not is_probable_prime(p, 5):
        p += 2
    return p

def main():
    if len(sys.argv) != 2:
        print("Usage: python primegen.py <bits>")
        sys.exit(1)

    bits = int(sys.argv[1])
    prime = generate_large_prime(bits)
    print(prime)

if __name__ == "__main__":
    main()
```

primecheck.py

```python
import random
import sys

def is_prime(n, k=5):  # Number of iterations, higher k means more accuracy
    if n <= 1:
        return False
    elif n <= 3:
        return True
    elif n % 2 == 0 or n % 3 == 0:
        return False

    # Write (n - 1) as 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        d //= 2
        r += 1

    def miller_test(d, n):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)  # Compute a^d % n
        if x == 1 or x == n - 1:
            return True
        while d != n - 1:
            x = (x * x) % n
            d *= 2
            if x == 1:
                return False
            if x == n - 1:
                return True
        return False

    # Perform the test k times
    for _ in range(k):
        if not miller_test(d, n):
            return False
    return True

def main():
    if len(sys.argv) != 2:
        print("Usage: python primecheck.py <number>")
        sys.exit(1)

    number = int(sys.argv[1])
    result = is_prime(number)
    print("True" if result else "False")

if __name__ == "__main__":
    main()
```

keygen

```
import sys
from primecheck import is_prime

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
```

```python
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, _ = egcd(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    else:
        return x % m

def keygen(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError("Both numbers must be prime.")
    elif p == q:
        raise ValueError("p and q cannot be the same")

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

    # Choose e
    e = 3
    while egcd(e, phi)[0] != 1:
        e += 2

    # Compute d
    d = modinv(e, phi)

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

def main():
    if len(sys.argv) != 3:
        print("Usage: python keygen.py <prime1> <prime2>")
        sys.exit(1)

    p = int(sys.argv[1])
    q = int(sys.argv[2])
    public_key, private_key = keygen(p, q)
    print("Public Key:", public_key)
    print("Private Key:", private_key)

if __name__ == "__main__":
    main()
```

encrypt

```python
# import sys

# def encrypt(n, e, m):
#     # m is the plaintext presented as an integer
#     c = pow(m, e, n)
#     return c

# def main():
#     if len(sys.argv) != 4:
#         print("Usage: python encrypt.py <n> <e> <m>")
#         sys.exit(1)

#     n = int(sys.argv[1])
#     e = int(sys.argv[2])
#     m = int(sys.argv[3])
#     ciphertext = encrypt(n, e, m)
#     print(ciphertext)

# if __name__ == "__main__":
#     main()

import sys

def encrypt(n, e, m):
    # Perform RSA encryption
    # m is the plaintext presented as an integer
    # n is the modulus component of the public key
    # e is the exponent component of the public key
    c = pow(m, e, n)  # Compute m^e mod n
    return c

def main():
    if len(sys.argv) != 4:
        print("Usage: python encrypt.py <n> <e> <m>")
        sys.exit(1)

    n = int(sys.argv[1])
    e = int(sys.argv[2])
    m = int(sys.argv[3])

    ciphertext = encrypt(n, e, m)
    print(ciphertext)

if __name__ == "__main__":
    main()
```

decrypt

```python
import sys

def decrypt(n, d, c):
    # Perform RSA decryption
    # c is the ciphertext as an integer
    # n is the modulus component of the private key
    # d is the exponent component of the private key
    m = pow(c, d, n)  # Compute c^d mod n
    return m

def main():
    if len(sys.argv) != 4:
        print("Usage: python decrypt.py <n> <d> <c>")
        sys.exit(1)

    n = int(sys.argv[1])
    d = int(sys.argv[2])
    c = int(sys.argv[3])

    plaintext = decrypt(n, d, c)
    print(plaintext)

if __name__ == "__main__":
    main()
```