Implementation of RSA Algorithm

Submitted by

Varnith Yemula

A20561527

INDEX

Introduction

The RSA algorithm is an essential component in modern cryptography, providing a strong foundation for safe communication over open channels. rooted in the beauty of number theory, RSA encryption protects confidential data from prying eyes by relying primarily on the characteristics of prime numbers and modular arithmetic. RSA's strength in cryptography is only one aspect of its essence; its practicality makes it an essential tool in a variety of domains, from cybersecurity to finance.

At its core, RSA relies on the mathematical elegance of prime numbers and modular arithmetic to achieve its cryptographic prowess. Through our project, we aim to elucidate the inner workings of RSA, shedding light on the fundamental principles that render it both robust and versatile. By embarking on this journey, we endeavor not only to deepen our understanding of cryptography but also to hone our skills in algorithmic design and implementation.

Throughout this report, we shall navigate through the essential components of RSA, beginning with the foundational concepts of prime number generation and primality testing. Subsequently, we shall delve into the heart of RSA encryption, elucidating the intricate processes of key generation, encryption, and decryption. Our journey will be characterized by meticulous exploration, rigorous analysis, and pragmatic implementation, culminating in a comprehensive understanding of RSA's mechanisms and applications.

Prime Numbers

Prime numbers serve as the cornerstone of RSA encryption, forming the basis for generating secure keys. In this section, we delve into the methodologies employed for generating large prime numbers efficiently.

Our primary focus lies on implementing the Miller-Rabin primality test, renowned for its accuracy and efficiency in identifying prime numbers.

Implementation of Miller-Rabin Test:

The Miller-Rabin test is based on the observation that if n is a prime number, then for any a less than n, either $a^{n-1} = 1 (mod n)$ or $a^d \equiv -1 (mod n)$, where d is the largest odd divisor of $n-1$. This property forms the basis of the Miller-Rabin test, where multiple random witnesses a are chosen to test the primality of n.

The Miller-Rabin test offers a balance between accuracy and efficiency. By choosing a sufficient number of random witnesses and performing multiple iterations of the test, the probability of erroneously classifying a composite number as prime can be made arbitrarily small. This probabilistic nature allows for rapid primality testing of large numbers, making it suitable for RSA key pair generation.

Optimizations include selecting the best witness values and an appropriate number of iterations are used to increase efficiency. It is possible to increase the Miller-Rabin test's efficiency without sacrificing its accuracy by utilizing mathematical properties and heuristics.

Key Pair Generation

RSA key pair generation involves selecting two distinct prime numbers p and q, computing the modulus $n=pq$, and determining the public and private exponents e and d respectively. Key validity and security considerations are paramount to ensure the integrity and confidentiality of encrypted messages.

The prime numbers p and q are chosen randomly, typically of equal bit length, to ensure computational security. The randomness of prime number generation is crucial to thwart potential attacks based on factorization.

Calculation of Public and Private Exponents:

Once p and q are selected, the modulus $n=pq$ is computed. The totient function $\phi(n)$ is then calculated as $\phi(n)=(p-1)(q-1)$. The public exponent e is chosen such that it is relatively prime to $\phi(n)$ and $1<e<\phi(n)$. The private exponent d is computed as the modular inverse of e modulo $\phi(n)$, i.e., $d \equiv e^{-1} \pmod{\phi(n)}$.

Ensuring the validity and security of RSA keys involves rigorous validation of prime numbers, adherence to key length recommendations, and protection against common attacks such as factorization and chosen ciphertext attacks. Additionally, key management practices such as key rotation and storage encryption are essential to mitigate potential vulnerabilities.

Encryption and Decryption

The fundamental functions of the RSA algorithm are RSA encryption and decryption, which enable safe communication between parties. Using the recipient's public key, the encryption process converts plaintext messages into ciphertext; the decryption process uses the recipient's private key to undo this conversion.

Implementation of Encryption Function:

The encryption function takes the public key (n,e) and plaintext message m as input and computes the ciphertext c using modular exponentiation, i.e., $c \equiv m^e \pmod{n}$. This process ensures that only the corresponding private key can decrypt the ciphertext and recover the original plaintext.

Implementation of Decryption Function:

The decryption function takes the private key (n,d) and ciphertext c as input and computes the plaintext message m using modular exponentiation, i.e., $m \equiv c^d \pmod{n}$. By leveraging the mathematical properties of modular arithmetic, decryption yields the original plaintext message, thus ensuring confidentiality and integrity.

Code and Code Structure

```
import random
import math

def compute_gcd(x, y):
    """ Utility function to compute the greatest common divisor. """
    while y:
        x, y = y, x % y
    return x

def rsa_encryption(public_key, plain_text):
    """ Function to encrypt a message with RSA. """
    n, e = public_key
    return pow(plain_text, e, n)

def prime_gen(bits):
    """ Function to generate a prime number with the specified bit length.
"""
    if bits < 2:
        raise ValueError("Bit length must be at least 2.")

    while True:
        p = random.getrandbits(bits)
        p |= (1 << (bits - 1)) | 1  # Ensure the number is of 'bits'
length and is odd

        if prime_check(p):
            return p

def prime_check(num, iterations=10):
    """ Implementing the Miller-Rabin test for checking primality. """
    if num in (2, 3):
        return True
    if num < 2 or num % 2 == 0:
        return False

    r = 0
```

```
    m = num - 1
    while m % 2 == 0:
        m >>= 1
        r += 1

    for _ in range(iterations):
        base = random.randrange(2, num - 1)
        val = pow(base, m, num)
        if val in (1, num - 1):
            continue
        for _ in range(r - 1):
            val = pow(val, 2, num)
            if val == num - 1:
                break
        else:
            return False
    return True

def rsa_decryption(private_key, cipher_text):
    """ Function to decrypt a message with RSA. """
    n, d = private_key
    return pow(cipher_text, d, n)

def generate_rsa_keys(bit_size):
    """ RSA key pair generation. """
    p = prime_gen(bit_size // 2)
    q = prime_gen(bit_size // 2)
    print("The value of p:", p)
    print("The value of q:", q)

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

    e = random.randint(3, 40)
    while compute_gcd(e, totient) != 1:
        e = random.randint(3, 40)

    d = pow(e, -1, totient)

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

```
# Request bit size from the user
key_length = int(input("Enter the bit length for the prime numbers: "))

# Key generation
public, private = generate_rsa_keys(key_length)
print("RSA Public Key:", public)
print("RSA Private Key:", private)

# Request a message to encrypt
message = int(input("Provide an integer message to encrypt: "))

# Encryption process
encoded_message = rsa_encryption(public, message)
print("Encoded Message:", encoded_message)

# Decryption process
decoded_message = rsa_decryption(private, encoded_message)
print("Decoded Message:", decoded_message)
```

Utility Functions

- **compute_gcd(x, y)**: Computes the greatest common divisor of two numbers using the Euclidean algorithm.

RSA Encryption and Decryption Functions

- **rsa_encryption(public_key, plain_text)**: Encrypts a message using the RSA algorithm.
- **rsa_decryption(private_key, cipher_text)**: Decrypts a cipher text using the RSA algorithm.

Prime Number Generation and Primality Testing Functions

- **prime_gen(bits)**: Generates a prime number with a specified bit length using probabilistic primality testing.

- **prime_check(num, iterations)**: Performs the Miller-Rabin test to check for primality of a number.

Key Pair Generation Function

- **generate_rsa_keys(bit_size)**: Generates a pair of RSA public and private keys based on a given bit size, ensuring their validity and compatibility.

Main Program

- Requests the bit length for prime numbers from the user.
- Generates RSA public and private key pairs.
- Requests an integer message from the user to encrypt.
- Encrypts the message using the public key.
- Decrypts the encrypted message using the private key.
- Outputs the generated keys, encrypted message, and decrypted message.

TEST CASES

Test case 1

Given input for bit length : 256

Integer message to encrypt : 2345676543876

```
input
Enter the bit length for the prime numbers: 256
The value of p: 31020565172193872972455869631048382563
The value of q: 19360175681463148792169334276853131973
RSA Public Key: (60056359147195053423045501930473262471897644295013103612143127923480144115989, 25)
RSA Private Key: (60056359147195053423045501930473262471897644295013103612143127923480144115989, 240225436588780213692182007721
8930498855753475459061335779875035376045158825)
Provide an integer message to encrypt: 2345676543876
Encoded Message: 50076107050135328435683800499236785137961491825983659567661007544113236959856
Decoded Message: 2345676543876


...Program finished with exit code 0
Press ENTER to exit console.
```

Test case 2

Given input for bit length : 1024

Integer message to encrypt : 65432345678987654345 67

```
input
Enter the bit length for the prime numbers: 1024
The value of p: 6836622401870064719488320280480527249778157508198678351225225147279901875314329196948498100044188938807168786
46291361642936046919319884554878742055141029
The value of q: 7973321110400274559116889217382692337991191603144914550839896193690827610911737318605711868455058098564479258
5397443588132144040508910807103949489058370 19
RSA Public Key: (5451058572066601652271964086378672918224145078469680467431851384268956576579763403523222024792616127082967598
0378224675056378000231737659910908643182046887081253072930929566146409323932904332989080686560509477736826667202620546702478181
767979049814513158889849286454842668219179207359304146038820375333952551, 19)
RSA Private Key: (5451058572066601652271964086378672918224145078469680467431851384268956576579763403523222024792616127082967598
0378224675056378000231737659910908643182046887081253072930929566146409323932904332989080686560509477736826667202620546702478181
767979049814513158889849286454842668219179207359304146038820375333952551, 2295182556659621748325037510054178070831218980408286
5126028847933764027690862161699045145367547857377191442518053989336865843368518626383120382586602967104114235604488669594754
18940045039357693462382114396487396703382157132255177954175419527119213274363011292683834476082011074895005856554979938807763 0
726107)
Provide an integer message to encrypt: 65432345678987654345 67
Encoded Message: 4230621384903359005127374957886224577161380916587623960769321286654539832660134290587647781871242780279950298
7351870187288839961298242146148889865554169188526013629733041311676598900901875182453445537715094426641680523459152584041100163
723708582886139219502457020231406541057480704954006530987861346898767647
Decoded Message: 65432345678987654345 67


...Program finished with exit code 0
Press ENTER to exit console.
```

Test case 3

Given input for bit length : 1024

Integer message to encrypt : 457890934

```
input
Enter the bit length for the prime numbers: 1024
The value of p: 118251570139391730189363081145746826843203353668594993968230407152663102105625750285857768143886866200093672902776683124555482685301934430383103067305954 41
The value of q: 125140461247296781718873350379306374767812101680656529662743692330343793359410107341232559903963406642627793275364627518903141597377681077398732097713417 87
RSA Public Key: (147980560304605481041597118585272509317285353631650337785980679476256056786988917408504507269016389312956964652363305745944112919814859140942364998227119925053073700582258967941984061108569001867064253371163150402974728905437819740501966004120348848603996698438957093551729702718506876793115867535784034993067, 11)
RSA Private Key: (147980560304605481041597118585272509317285353631650337785980679476256056786988917408504507269016389312956964652363305745944112919814859140942364998227119925053073700582258967941984061108569001867064253371163150402974728905437819740501966004120348848603996698438957093551729702718506876793115867535784034993067, 121075003885586302670397642478859325805051652971350276370347828662391319189354568788776415038286136710601152897388159246681546934393975660771025907640370827856803187020060908551369834311749051535424406001281553250007547767521314466567802695980717870199128348784641228616908019245882498134916891288218890682051)
Provide an integer message to encrypt: 457890934
Encoded Message: 1855164789455409498938794379686436205620258958215428690742838648760721333922073487195163600547 84
Decoded Message: 457890934


...Program finished with exit code 0
Press ENTER to exit console.
```

Results

Key Pair Generation

The RSA key pair generation process was successfully executed, resulting in the generation of valid public and private key pairs. The prime number generation algorithm based on the Miller-Rabin test demonstrated robustness in identifying prime numbers of the specified bit length. Test cases involving different bit lengths for prime numbers were conducted, and the key pair generation process consistently produced valid keys within a reasonable timeframe.

Encryption and Decryption

The encryption and decryption functionalities of the RSA algorithm were rigorously tested to verify their correctness and efficiency. Various plaintext messages were encrypted using the recipient's public key, and the resulting ciphertexts were decrypted using the corresponding private key. The decrypted plaintexts matched the original messages, confirming the integrity and accuracy of the encryption and decryption processes.

Conclusion

In conclusion, the exploration of the RSA algorithm has provided valuable insights into its inner workings and practical applications in modern cryptography. Rooted in the elegance of number theory, RSA encryption stands as a robust and versatile tool for securing communication channels and protecting sensitive data.

Through this project, we have learned how to navigate the complex world of RSA, from the implementation of encryption and decryption operations to the fundamental ideas of prime number creation.

The RSA algorithm's effectiveness and dependability are demonstrated by the successful completion of key pair generation, encryption, and decryption processes. Through the use of probabilistic primality testing methods like the Miller-Rabin test, we have proven our capacity to produce secure prime numbers quickly, establishing the foundation for secure key pairs.

Furthermore, our testing and performance analysis have showcased the robustness and efficiency of the implemented RSA algorithm. Key pair generation consistently produced valid keys within a reasonable timeframe, while encryption and decryption operations demonstrated accuracy and efficiency across various input sizes and workload conditions.