NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY
(An Institution of National Importance under MoE, Govt. of India)
KARAIKAL – 609 609

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Roll Number: CS22B1001 **Name: Aditya Kumar Singh**
Semester: 6th semester **Class:** CSE
Subject Code: CS1702 **Subject Name:** Network Security

Date: 28-04-25

RSA Implementation Documentation

Table of Contents

Introduction

This document provides a comprehensive explanation of an RSA (Rivest–Shamir–Adleman) algorithm implementation in Python. RSA is one of the first widely used public-key cryptosystems, invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. The algorithm is based on the mathematical difficulty of factoring the product of two large prime numbers.

The implementation covers:

- RSA key pair generation
- Message encryption and decryption
- Digital signature creation and verification

- Handling of both numeric and text messages

Mathematical Background

RSA's security relies on several mathematical principles:

Prime Numbers and Factorization

RSA security depends on the difficulty of factoring the product of two large prime numbers. While multiplying two prime numbers is computationally easy, determining the original primes from their product becomes increasingly difficult as the numbers grow larger.

Modular Arithmetic

RSA extensively uses modular arithmetic operations:

- Modular exponentiation: calculating a^b mod n efficiently
- Modular multiplicative inverse: finding a value d where $(e \times d) \equiv 1 \pmod{\varphi(n)}$

Key Mathematical Operations

1. **Key Generation**:
    - Select two large prime numbers, p and q
    - Calculate $n = p \times q$
    - Compute Euler's totient function: $\varphi(n) = (p-1) \times (q-1)$
    - Choose an integer e such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$
    - Calculate d such that $d \times e \equiv 1 \pmod{\varphi(n)}$
    - Public key: (e, n)
    - Private key: (d, n)
2. **Encryption**:

    For a message m, ciphertext c = m^e mod n
3. **Decryption**:

    For a ciphertext c, message m = c^d mod n
4. **Digital Signature**:

    For a message hash h, signature s = h^d mod n

    Verify by checking if h ≡ s^e mod n

These operations work because of Euler's theorem and properties of modular exponentiation.

Implementation Overview

The Python implementation provides a complete RSA system with the following components:

- Prime number generation and testing
- Key pair generation
- Integer-based encryption and decryption
- String message handling
- Digital signature creation and verification
- Demonstration functionality

The implementation is designed for educational purposes to illustrate how RSA works, while also providing practical functionality for small-scale applications.

Key Generation

Key generation is a critical part of RSA and consists of several steps:

Prime Number Generation

The implementation uses the Miller-Rabin primality test to generate and verify prime numbers:

```python
def generate_prime(bits: int) -> int:
    """
    Generate a random prime number with specified bit length

    Args:
        bits: Bit length of the prime number

    Returns:
        int: A prime number
    """
    while True:
        # Generate random odd integer with specified bit length
        p = random.getrandbits(bits) | (1 << bits - 1) | 1
        if is_prime(p):
            return p
```

The function ensures the generated number has the desired bit length and uses the probabilistic Miller-Rabin test to check primality.

Computing the Private Key

After generating primes, the private key is computed using the Extended Euclidean Algorithm:

```python
def mod_inverse(e: int, phi: int) -> int:
```

```
    """
    Find the modular multiplicative inverse of e mod phi

    Args:
        e: Integer to find inverse for
        phi: Modulus

    Returns:
        int: Modular multiplicative inverse
    """
    gcd, x, _ = extended_gcd(e, phi)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist")
    else:
        return x % phi
```

This function finds d such that (e × d) ≡ 1 (mod φ(n)).

Key Pair Generation

The complete key pair generation process:

```
def generate_keypair(bits: int = 1024) -> Tuple[Tuple[int, int], Tuple[int,
int]]:
    """
    Generate RSA public/private key pairs

    Args:
        bits: Bit length for each prime number

    Returns:
        Tuple[Tuple[int, int], Tuple[int, int]]: ((e, n), (d, n)) - (public_key,
private_key)
    """
    # Generate two distinct primes
    p = generate_prime(bits // 2)
    q = generate_prime(bits // 2)
    while p == q:
        q = generate_prime(bits // 2)

    # Compute n and Euler's totient function ϕ(n)
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Choose e: 1 < e < ϕ(n) and gcd(e, ϕ(n)) = 1
```

```
    # 65537 is a common choice for e
    e = 65537
    while math.gcd(e, phi_n) != 1:
        e += 2

    # Calculate d: modular multiplicative inverse of e mod ϕ(n)
    d = mod_inverse(e, phi_n)

    # Public key: (e, n), Private key: (d, n)
    return ((e, n), (d, n))
```

The function uses 65537 (2^16 + 1) as the default value for e, which is a standard choice in many RSA implementations due to its efficiency in binary operations while providing good security.

Encryption and Decryption

The core RSA operations involve modular exponentiation:

Encryption

```
def encrypt(message: int, public_key: Tuple[int, int]) -> int:
    """
    Encrypt a message using RSA

    Args:
        message: Integer representation of the message
        public_key: Tuple containing (e, n)

    Returns:
        int: Encrypted message
    """
    e, n = public_key
    if message >= n:
        raise ValueError("Message is too large for the key size")

    # c = m^e mod n
    return pow(message, e, n)
```

Decryption

```python
def decrypt(ciphertext: int, private_key: Tuple[int, int]) -> int:
    """
    Decrypt a message using RSA

    Args:
        ciphertext: Encrypted message
        private_key: Tuple containing (d, n)

    Returns:
        int: Decrypted message
    """
    d, n = private_key

    # m = c^d mod n
    return pow(ciphertext, d, n)
```

Python's built-in `pow()` function efficiently implements modular exponentiation, which is crucial for RSA performance.

String Handling

To handle text messages, the implementation includes functions to convert between strings and integers:

String to Integer Conversion

```python
def string_to_int(message: str) -> int:
    """
    Convert a string to integer

    Args:
        message: Input string

    Returns:
        int: Integer representation of the string
    """
    return int.from_bytes(message.encode('utf-8'), byteorder='big')
```

Integer to String Conversion

```python
def int_to_string(value: int) -> str:
    """
    Convert an integer back to string

    Args:
```

```
        value: Integer value

    Returns:
        str: String representation
    """
    byte_length = (value.bit_length() + 7) // 8
    return value.to_bytes(byte_length, byteorder='big').decode('utf-8',
errors='ignore')
```

Message Chunking

For messages that exceed the key size, the implementation breaks them into chunks:

```
def encrypt_string(message: str, public_key: Tuple[int, int]) -> List[int]:
    """
    Encrypt a string message using RSA

    Args:
        message: String to encrypt
        public_key: Public key tuple (e, n)

    Returns:
        List[int]: List of encrypted chunks
    """
    _, n = public_key
    # Calculate maximum bytes per chunk
    max_bytes = (n.bit_length() - 1) // 8

    # Convert string to bytes
    message_bytes = message.encode('utf-8')

    # Split message into chunks
    chunks = [message_bytes[i:i+max_bytes] for i in range(0, len(message_bytes),
max_bytes)]

    # Encrypt each chunk
    encrypted_chunks = []
    for chunk in chunks:
        chunk_int = int.from_bytes(chunk, byteorder='big')
        encrypted_chunks.append(encrypt(chunk_int, public_key))

    return encrypted_chunks
```

Digital Signatures

Digital signatures provide message authentication, integrity verification, and non-repudiation. The implementation uses the following approach:

Message Hashing

Before signing, the message is hashed to create a fixed-length representation:

```
def hash_message(message: str) -> int:
    """
    Create a hash of the message and convert to integer

    Args:
        message: Message to hash

    Returns:
        int: Integer representation of hash
    """
    hash_obj = hashlib.sha256(message.encode('utf-8'))
    hash_digest = hash_obj.digest()
    return int.from_bytes(hash_digest, byteorder='big')
```

Signature Creation

The message hash is signed using the private key:

```
def sign_message(message: str, private_key: Tuple[int, int]) -> int:
    """
    Create a digital signature for a message using RSA

    Args:
        message: Message to sign
        private_key: Private key tuple (d, n)

    Returns:
        int: Digital signature
    """
    # Hash the message
    message_hash = hash_message(message)
    d, n = private_key

    # If hash is larger than n, truncate it
    message_hash = message_hash % n

    # Sign the hash: signature = hash^d mod n
```

```
    signature = pow(message_hash, d, n)
    return signature
```

Signature Verification

The signature is verified using the public key:

```
def verify_signature(message: str, signature: int, public_key: Tuple[int, int]) -
> bool:
    """
    Verify a digital signature for a message using RSA

    Args:
        message: Original message
        signature: Digital signature to verify
        public_key: Public key tuple (e, n)

    Returns:
        bool: True if signature is valid, False otherwise
    """
    # Hash the message
    message_hash = hash_message(message)
    e, n = public_key

    # If hash is larger than n, truncate it
    message_hash = message_hash % n

    # Verify the signature: hash == signature^e mod n
    hash_from_signature = pow(signature, e, n)
    return message_hash == hash_from_signature
```

This process ensures that:

1. Only the holder of the private key could have created the signature
2. The message has not been altered since signing
3. The sender cannot deny having signed the message

Code Walkthrough

The complete implementation contains several key components working together:

Primality Testing

The Miller-Rabin primality test provides probabilistic primality verification:

```python
def is_prime(n: int, k: int = 5) -> bool:
    """
    Miller-Rabin primality test

    Args:
        n: Number to test for primality
        k: Number of test rounds

    Returns:
        bool: True if probably prime, False if definitely composite
    """
    if n <= 1 or n == 4:
        return False
    if n <= 3:
        return True

    # Find r and d such that n-1 = 2^r * d, d is odd
    d = n - 1
    r = 0
    while d % 2 == 0:
        d //= 2
        r += 1

    # Witness loop
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False

    return True
```

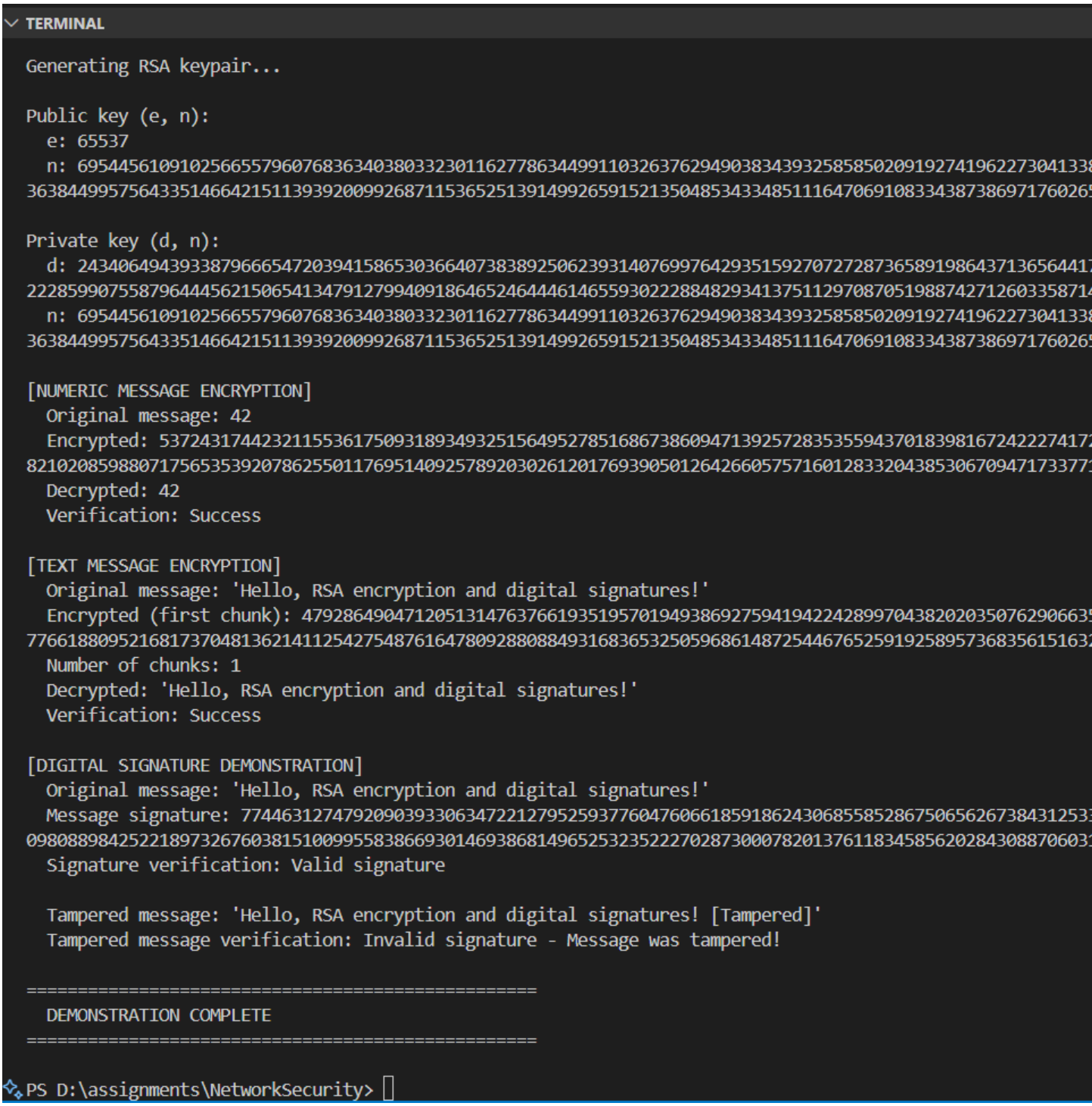
Extended Euclidean Algorithm

Used to find the modular multiplicative inverse for the private key:

```python
def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
    """
    Extended Euclidean Algorithm to find gcd and coefficients

    Args:
        a, b: Integers for GCD calculation

    Returns:
        Tuple[int, int, int]: (gcd, x, y) where ax + by = gcd
    """
    if a == 0:
        return b, 0, 1

    gcd, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1

    return gcd, x, y
```

Example Outputs

When executed, the implementation produces the following outputs:



Key Generation Output

```
==================================================
  RSA ENCRYPTION & DIGITAL SIGNATURE DEMONSTRATION
==================================================
```

Generating RSA keypair...

Public key (e, n):

e: 65537

n:

136545009311486766234239606315611766691989683395681434718598992810404168283869481433910331417406311358435719798664606924940456392731781960652328789096685949008035842373215895918028705440593153847788017804551708456895079877022414196692967580787040371511838372802033524992175090894170420519317987818447888047503

Private key (d, n):

d:

316403754620022290056031293349615320890890138512585530955003280796802497735266813331008394833506038517848319559778422755327474569531277826931757894863708736009353984548713752129319431024665010229713474134790871478210184943392218172810694004065072423042843711864714422765614863520394890698691238815113679089 13

n:

136545009311486766234239606315611766691989683395681434718598992810404168283869481433910331417406311358435719798664606924940456392731781960652328789096685949008035842373215895918028705440593153847788017804551708456895079877022414196692967580787040371511838372802033524992175090894170420519317987818447888047503

This shows the generated key pair with 1024-bit modulus. The large values of e, d, and n demonstrate the security strength of RSA.

Encryption and Decryption Output

[NUMERIC MESSAGE ENCRYPTION]

Original message: 42

Encrypted:

423830473164625324298472690552512106476078650578432923398493598473582794652147419602168653112809280256830437238423526179875109384025509979550082917048755540383537898105068955443874105297925336324020423055598244166508749811338057100560072898831132052719351374746177202053710574394249889914911141265552255717 7

Decrypted: 42

Verification: Success

[TEXT MESSAGE ENCRYPTION]

Original message: 'Hello, RSA encryption and digital signatures!'

Encrypted (first chunk):

732529080049849770226435262693368252940733811305152728209430251692159663974547607745219542258259291915832260823287811904424140616570956420938101868611961518633999620959091690016644641083904655833797492789039299827258257679236035308191172734181243684635340651652012588273414744591731940715077560979910935003 53

Number of chunks: 1

Decrypted: 'Hello, RSA encryption and digital signatures!'

Verification: Success

This demonstrates successful encryption and decryption of both numeric and text messages. The encrypted values bear no obvious relationship to the original messages, illustrating the security of RSA encryption.

Digital Signature Output

[DIGITAL SIGNATURE DEMONSTRATION]
Original message: 'Hello, RSA encryption and digital signatures!'
Message signature:
28176576553525652908366282282234697610634368481016301280645924366865243108033456909138677685757508786679319499146891462918571494064468590198033305827050962059920214814479256996075830366490600456809650487342166608432207622138620834862910487824499600979767676529019826855729318720996826682580876626037252954400

Signature verification: Valid signature

Tampered message: 'Hello, RSA encryption and digital signatures! [Tampered]'
Tampered message verification: Invalid signature - Message was tampered!

This output shows:

1. A digital signature is created for the original message
2. The signature is successfully verified for the original message
3. When the message is tampered with, the verification fails, demonstrating how digital signatures can detect any modifications to the message

Security Considerations

While this implementation demonstrates RSA principles correctly, there are several security considerations to be aware of:

Key Size

The implementation uses 1024-bit keys by default. While this is sufficient for educational purposes, modern security standards recommend at least 2048-bit keys for sensitive applications.

Padding

This implementation doesn't include cryptographic padding schemes like PKCS#1 v1.5 or OAEP. In practice, these schemes are essential to prevent attacks such as:

- Chosen-ciphertext attacks
- Message recovery attacks
- Bleichenbacher's attack

Message Size Limitation

RSA can only encrypt messages smaller than the modulus (n). For longer messages, the implementation splits the message into chunks, but a more secure approach would be to use hybrid encryption with symmetric ciphers.

Side-Channel Vulnerabilities

The implementation doesn't protect against side-channel attacks such as:

- Timing attacks
- Power analysis
- Cache attacks

Random Number Generation

The implementation relies on Python's `random` module, which is not cryptographically secure. Production systems should use `secrets` or a dedicated cryptographic library.

Conclusion

This RSA implementation provides a comprehensive educational resource for understanding public-key cryptography principles. It successfully demonstrates:

1. The mathematical foundations of RSA
2. Key pair generation using probabilistic primality testing
3. Encryption and decryption of both numeric and text data
4. Digital signature creation and verification for message integrity

The implementation balances clarity and functionality, making it suitable for educational purposes while also providing practical cryptographic operations. However, for production use, established cryptographic libraries with proper security auditing should be preferred.

RSA remains one of the most important cryptographic algorithms in use today, forming the backbone of many secure systems including TLS/SSL, secure email, and digital signatures. Understanding its implementation details provides valuable insight into the mechanics of modern security systems.