Implementation and Analysis of RSA Algorithm

Tejas Budhwal (2101AI42)

January 27, 2025

Contents

1 Introduction to RSA Algorithm

RSA (Rivest-Shamir-Adleman) is a widely used public-key cryptosystem that facilitates secure data transmission. It relies on the mathematical complexity of prime factorization.

2 Implementation of RSA Algorithm in Python

The following Python code demonstrates the implementation of the RSA algorithm, including key generation, encryption, and decryption.

2.1 Python Code for RSA Algorithm

Listing 1: RSA Algorithm in Python

```python
import random
import sympy

class RSA:
    """Function to initialize RSA and specify the key size"""
    def __init__(self, key_size=1024):
        self.key_size = key_size
        self.public_key, self.private_key = self.generate_key_pair()

    """Function for generating a large prime number of specified bit size"""
    def generate_large_prime(self, bits):
        return sympy.randprime(2**(bits-1), 2**bits)

    """Function to compute the GCD using Euclidean algorithm"""
    def gcd(self, a, b):
        while b:
            a, b = b, a % b
        return a

    """Function to compute modular inverse using Extended Euclidean Algorithm"""
    def mod_inverse(self, e, phi):
        def egcd(a, b):
```

```
            if a == 0:
                return b, 0, 1
            g, x, y = egcd(b % a, a)
            return g, y - (b // a) * x, x

        g, x, _ = egcd(e, phi)
        if g != 1:
            raise ValueError("Modular inverse does not exist")
        return x % phi

    """Generate RSA key pair"""
    def generate_key_pair(self):
        p = self.generate_large_prime(self.key_size // 2)
        q = self.generate_large_prime(self.key_size // 2)
        n = p * q
        phi = (p - 1) * (q - 1)

        e = 65537  # Common public exponent
        if self.gcd(e, phi) != 1:
            raise ValueError("e and phi(n) are not coprime, regenerate primes.")

        d = self.mod_inverse(e, phi)

        public_key = (e, n)
        private_key = (d, n)
        return public_key, private_key

    """Encryption function to encrypt a plaintext message using the public key"""
    def encrypt(self, plaintext):
        e, n = self.public_key
        numeric_representation = [ord(char) for char in plaintext]
        encrypted_blocks = [pow(char, e, n) for char in numeric_representation]
        return encrypted_blocks

    """Decryption function to decrypt an encrypted message using the private key"""
```

```python
    def decrypt(self, encrypted_blocks):
        d, n = self.private_key
        decrypted_chars = [chr(pow(block, d, n)) for
            block in encrypted_blocks]
        return "".join(decrypted_chars)

# Initialize RSA with a 1024-bit key size
rsa = RSA(key_size=1024)

# Test Cases: Different Message Sizes
test_messages = [
    "T",
    "Tejas",
    "RSA Encryption by me!",
    "This is a test message to check RSA encryption
       using a medium length message",
    "I am Tejas Budhwal and now I am testing for a
       larger message and hence I will write a longer
       sentence here so that the test can successfully
       be conducted to show the results.",
]

for i, message in enumerate(test_messages, start=1):
    print(f"Test {i}: Original Message -> {message}")

    encrypted_message = rsa.encrypt(message)
    print(f"Encrypted Message: {encrypted_message}")

    decrypted_message = rsa.decrypt(encrypted_message)
    print(f"Decrypted Message: {decrypted_message}")

    assert message == decrypted_message, "Decryption
       failed!\n"
    print("Encryption and Decryption Successful!\n")
```

3 Explanation of the Code

- **Key Generation:** Generates large prime numbers p and q, computes $n = p \times q$ and Euler's totient function $\phi(n) = (p-1)(q-1)$.
- **Public Key:** Composed of (e, n), where e is a common prime like 65537.
- **Private Key:** Computed as d, the modular inverse of e modulo $\phi(n)$.
- **Encryption:** Converts plaintext characters to their ASCII values and applies the encryption formula:

$$C = M^e \mod n$$

- **Decryption:** Recovers the original message using the formula:

$$M = C^d \mod n$$

4 Vulnerabilities of RSA and Mitigations

4.1 Vulnerabilities

1. **Insufficient Key Length:** Keys smaller than 1024 bits are susceptible to brute-force attacks.
2. **Common Public Exponent (e = 65537):** Although 65537 is widely adopted for efficiency, poorly chosen exponents can weaken encryption strength.
3. **Factorization Risks:** If the modulus n can be factored into its prime components p and q, the private key can be easily derived.
4. **Timing Attacks:** By measuring the time taken for decryption, attackers may infer details about the private key.
5. **Vulnerability to Chosen Ciphertext Attacks:** Manipulating ciphertext strategically can help attackers uncover information about the original plaintext.
6. **Weak Randomness in Prime Generation:** Predictable patterns in generating prime numbers p and q can significantly compromise security.

4.2 Mitigation Strategies

- Use Stronger Keys: Employ key sizes of at least 2048 bits to ensure resistance against brute-force attacks. Regularly update cryptographic standards as computational power advances.

- Secure Exponent Selection: While 65537 is secure due to its properties, avoid using extremely small or large exponents. Ensure the modulus is large enough to maintain security even with commonly used exponents.

- Generate Strong Primes: Use cryptographically secure random number generators (CSPRNGs) to create large, unpredictable prime numbers. Apply primality tests like the Miller-Rabin test to verify prime integrity.

- Implement Side-Channel Attack Protections: Design cryptographic operations to execute in constant time, regardless of input values, to prevent timing-based side-channel attacks.

- Padding Schemes: Use OAEP (Optimal Asymmetric Encryption Padding) to prevent chosen ciphertext attacks.

- Use Hybrid Encryption: Combine RSA with symmetric encryption (AES) to handle large data securely.

5 Analysis Based on Key Size and Input Size

- **Key Size:** Larger key sizes (2048 or 4096 bits) enhance security but increase computational time for encryption and decryption.

- **Input Size:** The plaintext message must be smaller than the modulus n. To encrypt larger messages, hybrid encryption (combining RSA with symmetric encryption like AES) is recommended.

- **Performance:** Encryption is faster with a small public exponent (e.g., 65537), while decryption is slower due to larger private exponents.

6 RSA Optimization Strategies

- **Use Hybrid Encryption:** Use RSA to encrypt symmetric keys (AES) instead of full messages.
- **Increase Key Size Selectively:** 2048-bit RSA is a good balance between security and speed.
- **Optimize Prime Number Generation:** Use efficient libraries like OpenSSL for prime generation.
- **Parallel Processing:** Leverage multi-core CPUs for encryption/decryption tasks.
- **Use Fast Modular Exponentiation:** Optimize mathematical operations using Montgomery Multiplication.

7 Conclusion

RSA remains one of the most robust public-key cryptographic algorithms. However, its security depends heavily on proper implementation, key size, and mitigation of known vulnerabilities.