RSA ENCRYPTION

alexandrubrabete@yahoo.com

TEHNICAL UNIVERSITY OF CLUJ- NAPOCA, SOFTWARE ENGINEERING DEPARTMENT

RSA Encryption Project Documentation

Overview

This project implements a full **RSA encryption** system in Java, utilizing **PKCS#1 v1.5 padding** for secure message encryption and decryption. RSA is a widely-used asymmetric cryptographic algorithm where two keys are involved: a public key (for encryption) and a private key (for decryption). The project focuses on securely generating keys, encrypting and decrypting messages, and ensuring security using padding schemes and cryptographically secure random numbers.

Table of Contents

1. Introduction

This Java implementation demonstrates how to securely encrypt and decrypt messages using RSA. The implementation includes essential features such as:

- RSA public and private key pair generation.
- **PKCS#1 v1.5** padding for secure encryption.
- Modular exponentiation for encryption and decryption.
- Cryptographically secure random numbers to ensure strong key generation and padding.

The code is designed for educational and practical use, with high attention to security practices.

2. System Requirements

- **Java Development Kit (JDK)**: The project is written in Java and requires JDK version 8 or later.
- **IDE**: You can use any Java-compatible Integrated Development Environment (IDE) like **IntelliJ IDEA**, **Eclipse**, or **NetBeans**.
- **Operating System**: The project is cross-platform and works on all major operating systems, including Windows, macOS, and Linux.

3. Key Concepts

RSA Algorithm

- **Public Key (e, n)**: The public key consists of an exponent e and a modulus n. The modulus n is the product of two large prime numbers p and q, and e is a value typically chosen as 65537 for efficient encryption.
- **Private Key (d, n)**: The private key consists of the exponent d and the modulus n. The value of d is the modular inverse of e with respect to the Euler's totient function phi(n) of n.
- **Encryption**: Ciphertext = Plaintext^e mod n
- **Decryption**: Plaintext = Ciphertext^d mod n

PKCS#1 v1.5 Padding

- Padding is applied before encryption to prevent attacks like the padding oracle attack.
- PKCS#1 v1.5 padding involves adding random data between the message and a separator byte (0x00), ensuring the message length fits within the key size.

Modular Exponentiation

- Efficient modular exponentiation is performed using Java's BigInteger.modPow() function. This ensures that large integer powers are computed securely and efficiently.

4. Detailed Implementation

Key Generation

- **Generate Two Large Primes**: The method generatePrime(int bitLength) generates cryptographically secure prime numbers of a specified bit length. The prime numbers p and q are used to compute the modulus n = p * q.
- **Compute the Modulus n**: The modulus n is a part of both the public and private keys and is calculated as the product of the two primes p and q.
- **Compute Euler's Totient Function phi(n)**: phi(n) is calculated as (p-1) * (q-1). This value is used to compute the modular inverse of the public exponent e.
- **Choose the Public Exponent e**: A typical value for e is 65537, which is chosen for efficiency and cryptographic safety.
- **Compute the Private Exponent d**: The private exponent d is calculated as the modular inverse of e modulo phi(n), i.e., d ≡ e^(-1) mod phi(n).

Encryption and Decryption

- **Encrypt**: The message is padded using **PKCS#1 v1.5 padding** before encryption. The padded message is then raised to the power of e and reduced modulo n to obtain the ciphertext.
- **Decrypt**: The ciphertext is decrypted by raising it to the power of d modulo n. The padding is then removed to retrieve the original message.

PKCS#1 v1.5 Padding

- **Padding Before Encryption**: The message is padded with random bytes and an end delimiter (0x00) to ensure it fits the block size. The total length of the padded message must be less than or equal to the key size in bytes.
- **Unpadding After Decryption**: After decryption, the padding is removed, and the original message is retrieved.

5. Classes and Methods

AdvancedRSA Class

- **generateKeys(int bitLength)**: Generates an RSA key pair (public and private keys). Takes the bit length of the key as an argument.
- **generatePrime(int bitLength)**: Generates a large cryptographically secure prime number of the specified bit length.
- **encrypt(BigInteger message, BigInteger e, BigInteger n)**: Encrypts a given message using the public key (e, n). The message is padded before encryption.

- **decrypt(BigInteger ciphertext, BigInteger d, BigInteger n)**: Decrypts a given ciphertext using the private key (d, n). The padded message is unpadded after decryption.
- **pkcs1Padding(byte[] messageBytes, int keyLength)**: Applies PKCS#1 v1.5 padding to the given message. Returns the padded byte array.
- **pkcs1UnPadding(byte[] paddedMessage)**: Removes PKCS#1 v1.5 padding from the decrypted message.

Helper Methods

- **SecureRandom random**: A SecureRandom instance used for generating random padding bytes.

6. How to Run the Code

1. **Clone the Project**: Clone the project from the repository or copy the code into your Java IDE.
2. **Set Up the Environment**: Ensure you have the JDK installed (version 8 or later).
3. **Run the Program**: Execute the AdvancedRSA class. The main method demonstrates the RSA key generation, encryption, and decryption process.

7. Security Considerations

- **Key Size**: The default key size is set to **2048 bits**, which is the recommended size for secure RSA encryption. Increasing the key size (e.g., to 3072 or 4096 bits) provides higher security at the cost of performance.
- **Padding**: The **PKCS#1 v1.5 padding** used here ensures that the message is securely padded, preventing potential attacks like **padding oracle attacks**.
- **Prime Generation**: The project uses cryptographically secure prime generation using **BigInteger.nextProbablePrime()**, which ensures the primes are difficult to predict.

8. Limitations and Further Improvements

Limitations:

- **Performance**: RSA encryption is slow compared to symmetric encryption algorithms like AES, especially with larger key sizes.
- **Message Size**: The size of the message must be less than the key size in bytes, after padding. This can limit the size of data that can be encrypted directly using RSA.

Future Improvements:

- **Hybrid Encryption**: To improve performance, RSA can be combined with a symmetric encryption algorithm like **AES** to encrypt the message, while RSA is used to securely exchange the AES key.
- **Digital Signatures**: The project can be extended to support **digital signatures** for message integrity verification.
- **Key Management**: The implementation can include secure key storage and management practices, such as generating keys securely and rotating keys periodically.

Conclusion

This RSA implementation demonstrates a secure, efficient, and cryptographically sound approach to public-key encryption, with attention to modern security practices like **PKCS#1 padding** and **cryptographically secure random number generation**. It provides a solid foundation for learning about asymmetric encryption and can be extended with more features for real-world use.