Chosen Ciphertext Attack On RSA

Aim:

To demonstrate a Chosen Ciphertext Attack (CCA) on an RSA-based encryption scheme by manipulating an encrypted message and using a decryption oracle to retrieve the original plaintext.

Description:

This program simulates RSA encryption and decryption and showcases how an attacker can exploit a decryption oracle to recover the original message. It performs the following steps:

1. Generates an RSA key pair (public and private keys).
2. Encrypts a plaintext message using the public key.
3. Simulates a Chosen Ciphertext Attack by modifying the ciphertext with a random multiplier and querying the decryption oracle.
4. Uses the decrypted modified message to recover the original plaintext.

Code:

```
import java.math.BigInteger;
import java.security.SecureRandom;

public class CCAttack {

    // Generate a large prime number
    private static BigInteger generatePrime(int bitLength) {
        SecureRandom rand = new SecureRandom();
        return BigInteger.probablePrime(bitLength, rand);
    }

    // Generate RSA keys
    public static BigInteger[] generateKeyPair(int bitLength) {
        BigInteger p = generatePrime(bitLength / 2);
        BigInteger q = generatePrime(bitLength / 2);
        BigInteger n = p.multiply(q);
        BigInteger phi = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));

        // Choose public exponent e (commonly 65537)
        BigInteger e = BigInteger.valueOf(65537);

        // Ensure e and phi(n) are coprime (gcd(e, phi(n)) = 1)
        while (e.gcd(phi).compareTo(BigInteger.ONE) != 0) {
            e = e.add(BigInteger.TWO); // Try next odd number
        }

        // Calculate the private exponent d such that e * d ≡ 1 (mod φ(n))
```

```java
        BigInteger d = e.modInverse(phi);

        // Public key is (e, n), Private key is (d, n)
        return new BigInteger[] { e, n, d };
    }

    // Encrypt a message using the public key (e, n)
    public static BigInteger encrypt(BigInteger message, BigInteger e, BigInteger n) {
        return message.modPow(e, n);
    }

    // Decrypt a message using the private key (d, n)
    public static BigInteger decrypt(BigInteger cipherText, BigInteger d, BigInteger n) {
        return cipherText.modPow(d, n);
    }

    public static void main(String[] args) {
        int bitLength = 512; // Key size in bits

        // Step 1: Generate RSA keys
        BigInteger[] keys = generateKeyPair(bitLength);
        BigInteger e = keys[0]; // Public exponent
        BigInteger n = keys[1]; // Modulus
        BigInteger d = keys[2]; // Private exponent

        System.out.println("Public Key (e, n): (" + e + ", " + n + ")");
        System.out.println("Private Key (d, n): (" + d + ", " + n + ")");

        // Step 2: Encrypt a secret message
        String message = "hello world!";
        BigInteger messageBigInt = new BigInteger(1, message.getBytes());
        System.out.println("Original Message: " + message);

        BigInteger cipherText = encrypt(messageBigInt, e, n);
        System.out.println("Encrypted Message: " + cipherText);

        // Step 3: Simulating a Chosen Ciphertext Attack
        System.out.println("\n=== Chosen Ciphertext Attack (CCA) Simulation ===");

        // The attacker chooses a random number r (mod n)
        BigInteger r = new BigInteger(bitLength / 2, new SecureRandom()).mod(n);
        System.out.println("Attacker chosen r: " + r);

        // Attacker creates a modified ciphertext: C' = C * r^e mod n
        BigInteger modifiedCipherText = cipherText.multiply(r.modPow(e, n)).mod(n);
        System.out.println("Modified Ciphertext (C'): " + modifiedCipherText);
```

```
        // The attacker queries the decryption oracle to get the decrypted C'
        BigInteger decryptedModifiedMessage = decrypt(modifiedCipherText, d, n);
        System.out.println("Decryption Oracle Output (M'): " + decryptedModifiedMessage);

        // The attacker recovers the original message using: M = M' / r mod n
        BigInteger recoveredMessage = decryptedModifiedMessage.multiply(r.modInverse(n)).mod(n);
        String recoveredText = new String(recoveredMessage.toByteArray());
        System.out.println("\nRecovered Message by Attacker: " + recoveredText);
    }
}
```

Output:

```
Public Key (e, n): (65537, 931550500064579878010018457229092125515233275161295602937725588762425745499149158226691448875580790571951177248551265624519564328424288623228580784017 0569)
Private Key (d, n): (901061221906615310844425134514344691100625109160090496382628141703582600037854027187351362455049198242042670493792979275484497306533996916505693544067 6993, 93155
0500064579878010018457229092125515233275161295602937725588762425745499149158226691448875580790571951177248551265624519564328424288623228580784017 0569)
Original Message: hello world!
Encrypted Message: 396256027017785059249911912502856719409775212906289065921866804157150183980586171462717047688423521160489804608346823590004044031348309359744131168094 3041

=== Chosen Ciphertext Attack (CCA) Simulation ===
Attacker chosen r: 102526967538661770521239333779796428193845186161762361697474690447778571926591
Modified Ciphertext (C'): 177702038628703520037766724950641582744823353845149723029493681816490205407888126106800239103620444762033812024784666263385903485764230529560528681371035
Decryption Oracle Output (M'): 331254938654386165655919655957675902166401836805795571602152952583313674381093906134635138316906544 6596639

Recovered Message by Attacker: hello world!
```

Code analysis:

1. **Key Generation:**
   - Two large prime numbers p and q are generated to compute n = p * q.
   - Euler's totient function phi(n) = (p-1) * (q-1) is calculated.
   - A common public exponent e = 65537 is chosen, ensuring it's coprime with phi(n).
   - The private exponent d is computed as d = e^(-1) mod phi(n).
2. **Encryption:**
   - Converts the plaintext message to a BigInteger.
   - Encrypts the message using cipherText = message^e mod n.
3. **Decryption:**
   - Decrypts ciphertext using message = cipherText^d mod n.
4. **Chosen Ciphertext Attack (CCA) Simulation:**
   - Attacker selects a random integer r.
   - Modifies the original ciphertext C' = C * r^e mod n.
   - Queries the decryption oracle for M' = C'^d mod n.
   - Recovers the original message using M = M' * r^(-1) mod n.

Time complexity:

- **Key Generation:** O(n^2 log n log log n) (Probable prime generation and modular inverse calculations)
- **Encryption:** O(log e * log n) (Modular exponentiation using square-and-multiply algorithm)
- **Decryption:** O(log d * log n) (Modular exponentiation)
- **CCA Attack:** O(log e * log n) + O(log n) (Ciphertext modification, modular exponentiation, and modular inverse)

Space complexity:

- **Key Storage:** O(log n) for p, q, n, e, d
- **Encryption/Decryption:** O(log n) for storing messages and ciphertexts
- **CCA Attack:** O(log n) for storing intermediate values like r, C', and M'