Technical Documentation of Cryptographic Implementations

RSA, AES, and Diffie-Hellman Exchange Code Walkthrough

UTPB-COSC-4470-Project3-4

Generated using Claude 3.7

April 4, 2025

Contents

Chapter 1

Introduction

This technical documentation provides a detailed walkthrough of three cryptographic algorithm implementations in Java. The code was generated with the assistance of Claude 3.7 AI and represents educational implementations of RSA, AES, and Diffie-Hellman key exchange algorithms. This document focuses on explaining the code structure, function implementations, and algorithmic processes that make each cryptographic operation work.

Chapter 2

RSA Implementation (RSA.java)

2.1 Class Structure and Fields

The RSA implementation is organized as a Java class with the following key fields:

```
private static final int DEFAULT_KEY_SIZE = 1024;
private static final SecureRandom RANDOM = new
    ↪ SecureRandom();

private BigInteger n; // modulus
private BigInteger e; // public exponent
private BigInteger d; // private exponent
private int keySize;
```

These fields store the RSA key parameters and configuration. The `DEFAULT_KEY_SIZE` determines the strength of the RSA key (1024 bits by default). The `RANDOM` field provides a cryptographically secure source of randomness.

2.2 Constructor Implementation

The class has two constructors:

```
public RSA() {
    this(DEFAULT_KEY_SIZE);
```

```
}

public RSA(int keySize) {
    this.keySize = keySize;
    generateKeyPair();
}
```

The default constructor creates an RSA instance with the default key size, while the parameterized constructor allows specifying a custom key size. Both constructors call `generateKeyPair()` to create the key pair.

2.3 Key Generation

The `generateKeyPair()` method implements the RSA key generation algorithm:

```
public void generateKeyPair() {
    // Generate two large prime numbers
    BigInteger p = generateLargePrime(keySize / 2);
    BigInteger q = generateLargePrime(keySize / 2);

    // Calculate n = p * q
    n = p.multiply(q);

    // Calculate   (n) = (p-1) * (q-1)
    BigInteger phi = p.subtract(BigInteger.ONE)
                      .multiply(q.subtract(BigInteger.ONE
                         ↪ ));

    // Choose e such that 1 < e <   (n) and gcd(e,   (n)
       ↪ ) = 1
    e = BigInteger.valueOf(65537); // Common value for e

    // Ensure e and phi are coprime
    while (phi.gcd(e).compareTo(BigInteger.ONE) != 0) {
        e = e.add(BigInteger.TWO);
    }

    // Calculate d such that (d * e) %   (n) = 1
    d = e.modInverse(phi);
```

```
}
```

This method follows these steps:

1. Generate two large primes `p` and `q` of half the key size
2. Calculate the modulus `n = p * q`
3. Calculate Euler's totient function `(n) = (p-1) * (q-1)`
4. Set the public exponent `e` to 65537 (a common choice)
5. Ensure `e` and `(n)` are coprime
6. Calculate the private exponent `d` as the modular multiplicative inverse of `e` modulo `(n)`

The method `generateLargePrime()` uses Java's `BigInteger.probablePrime()` to generate prime numbers:

```
private BigInteger generateLargePrime(int bitLength) {
    return BigInteger.probablePrime(bitLength, RANDOM);
}
```

2.4 Encryption and Decryption

The encryption and decryption methods implement the RSA algorithm's core operations:

```
public BigInteger encrypt(BigInteger message) {
    if (message.compareTo(n) >= 0) {
        throw new IllegalArgumentException("Message must
            ↪ be less than n");
    }
    // c = m^e mod n
    return message.modPow(e, n);
}

public BigInteger decrypt(BigInteger ciphertext) {
    if (ciphertext.compareTo(n) >= 0) {
```

```
        throw new IllegalArgumentException("Ciphertext must be less than n");
    }
    // m = c^d mod n
    return ciphertext.modPow(d, n);
}
```

These methods perform:

- Input validation to ensure the message/ciphertext is less than `n`
- Modular exponentiation using `modPow` method from `BigInteger`
- Encryption: `c = m^e mod n` (using public key)
- Decryption: `m = c^d mod n` (using private key)

2.5 Digital Signatures

The digital signature functions implement signing and verification:

```
public BigInteger sign(BigInteger message) {
    if (message.compareTo(n) >= 0) {
        throw new IllegalArgumentException("Message must be less than n");
    }
    // signature = message^d mod n (same as decryption)
    return message.modPow(d, n);
}

public boolean verify(BigInteger message, BigInteger signature) {
    if (signature.compareTo(n) >= 0) {
        throw new IllegalArgumentException("Signature must be less than n");
    }
    // Verify by checking if message == signature^e mod n
    BigInteger computedMessage = signature.modPow(e, n);
    return message.equals(computedMessage);
}
```

These methods perform:

- Signing: Apply the private key to the message (similar to decryption)
- Verification: Apply the public key to the signature and compare with the original message

2.6 Test Method

The `main()` method demonstrates the usage of the RSA class:

```
public static void main(String[] args) {
    // Create a new RSA instance with a 1024-bit key
    RSA rsa = new RSA(1024);

    // Test message
    BigInteger message = new BigInteger("123456789");
    System.out.println("Original␣message:␣" + message);

    // Test encryption and decryption
    BigInteger ciphertext = rsa.encrypt(message);
    System.out.println("Encrypted:␣" + ciphertext);

    BigInteger decrypted = rsa.decrypt(ciphertext);
    System.out.println("Decrypted:␣" + decrypted);

    // Test signature and verification
    BigInteger signature = rsa.sign(message);
    System.out.println("\nSignature:␣" + signature);

    boolean isValid = rsa.verify(message, signature);
    System.out.println("Signature␣valid:␣" + isValid);

    // Testing with a different message to show invalid
       ↪ signature
    BigInteger differentMessage = new BigInteger("
       ↪ 987654321");
    boolean invalidTest = rsa.verify(differentMessage,
       ↪ signature);
    System.out.println("Different␣message␣with␣same␣
       ↪ signature␣valid␣" +
```

```
27                          "(should be false): " +
                               ↪ invalidTest);
28  }
```

This method demonstrates:

- Creating an RSA instance with a 1024-bit key
- Encrypting and decrypting a test message
- Signing a message and verifying the signature
- Testing signature verification with an incorrect message

Chapter 3

AES Implementation (AES.java)

3.1 Class Structure and Constants

The AES implementation is structured as a Java class with several constants:

```
private static boolean DEBUG = false;

// AES constants
private static final int BLOCK_SIZE = 16; // 128 bits = 16 bytes
private static final int ROUNDS = 10;     // 10 rounds for AES-128

// S-box for SubBytes operation
private static final byte[] S_BOX = { ... }; // 256-byte substitution box

// Inverse S-box for inverse SubBytes operation
private static final byte[] INV_S_BOX = { ... }; // 256-byte inverse box

// Rcon values for key expansion
private static final int[] RCON = {
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36
```

```
};
```

These constants define:

- The AES block size (16 bytes or 128 bits)
- The number of rounds for AES-128 (10 rounds)
- The substitution box (S-box) for the SubBytes transformation
- The inverse S-box for decryption
- Round constants (Rcon) used in the key expansion

3.2 Encryption Process

The main encryption method processes a plaintext string using either ECB or CBC mode:

```
public static String encrypt(String plaintext, String
    ↪ keyString,
                                boolean isCBC) {
    // Convert inputs to byte arrays
    byte[] input = plaintext.getBytes(StandardCharsets.
       ↪ UTF_8);
    byte[] key = prepareKey(keyString);

    // Pad input to multiple of 16 bytes (PKCS#7 padding
       ↪ )
    input = padPKCS7(input);

    // Generate key schedule
    byte[][] keySchedule = keyExpansion(key);

    // Process each block
    byte[] output = new byte[input.length];
    byte[] iv = new byte[BLOCK_SIZE];  // Initialization
       ↪  vector for CBC

    for (int i = 0; i < input.length; i += BLOCK_SIZE) {
        byte[] block = new byte[BLOCK_SIZE];
```

```
19          System.arraycopy(input, i, block, 0, BLOCK_SIZE)
               ↪ ;
20
21          // For CBC mode, XOR with previous ciphertext
               ↪ block (or IV)
22          if (isCBC) {
23              xorBlocks(block, iv);
24          }
25
26          // Encrypt the block
27          block = encryptBlock(block, keySchedule);
28
29          // Copy to output
30          System.arraycopy(block, 0, output, i, BLOCK_SIZE
               ↪ );
31
32          // For CBC mode, update IV for next block
33          if (isCBC) {
34              System.arraycopy(block, 0, iv, 0, BLOCK_SIZE
                   ↪ );
35          }
36      }
37
38      // Encode result as Base64
39      return Base64.getEncoder().encodeToString(output);
40  }
```

The encryption process follows these steps:

1. Convert the plaintext string and key to byte arrays
2. Apply PKCS#7 padding to ensure the input is a multiple of 16 bytes
3. Generate the key schedule (round keys) from the initial key
4. Process each 16-byte block:
    - For CBC mode, XOR the block with the previous ciphertext block or IV
    - Encrypt the block using the AES block cipher
    - For CBC mode, update the IV with the current ciphertext block

5. Return the Base64-encoded ciphertext

3.3 Block Encryption

The `encryptBlock()` method implements the AES block cipher:

```
private static byte[] encryptBlock(byte[] block, byte
    ↪ [][] keySchedule) {
    byte[] state = new byte[BLOCK_SIZE];
    System.arraycopy(block, 0, state, 0, BLOCK_SIZE);

    // Initial round: AddRoundKey
    addRoundKey(state, keySchedule[0]);

    // Main rounds
    for (int round = 1; round < ROUNDS; round++) {
        // SubBytes
        subBytes(state);
        // ShiftRows
        shiftRows(state);
        // MixColumns
        mixColumns(state);
        // AddRoundKey
        addRoundKey(state, keySchedule[round]);
    }

    // Final round (no MixColumns)
    subBytes(state);
    shiftRows(state);
    addRoundKey(state, keySchedule[ROUNDS]);

    return state;
}
```

The block encryption process follows these steps:

1. Initialize the state with the input block
2. Apply the initial round key (AddRoundKey)
3. For rounds 1 to 9:

- Apply SubBytes transformation (substitute each byte using S-box)
- Apply ShiftRows transformation (cyclically shift rows)
- Apply MixColumns transformation (mix data within columns)
- Apply AddRoundKey transformation (XOR with round key)

4. For the final round (round 10):
   - Apply SubBytes
   - Apply ShiftRows
   - Apply AddRoundKey (no MixColumns in the final round)
5. Return the encrypted state

3.4 AES Transformation Functions

The AES implementation includes methods for each of the four main transformations:

3.4.1 SubBytes / InvSubBytes

```
private static void subBytes(byte[] state) {
    for (int i = 0; i < state.length; i++) {
        state[i] = S_BOX[state[i] & 0xFF];
    }
}

private static void invSubBytes(byte[] state) {
    for (int i = 0; i < state.length; i++) {
        state[i] = INV_S_BOX[state[i] & 0xFF];
    }
}
```

These methods perform byte substitution using the S-box or inverse S-box.

3.4.2 ShiftRows / InvShiftRows

```
private static void shiftRows(byte[] state) {
    byte[] temp = new byte[16];

    // Row 0: No shift
    temp[0] = state[0];
    temp[4] = state[4];
    temp[8] = state[8];
    temp[12] = state[12];

    // Row 1: Shift left by 1
    temp[1] = state[5];
    temp[5] = state[9];
    temp[9] = state[13];
    temp[13] = state[1];

    // Row 2: Shift left by 2
    temp[2] = state[10];
    temp[6] = state[14];
    temp[10] = state[2];
    temp[14] = state[6];

    // Row 3: Shift left by 3
    temp[3] = state[15];
    temp[7] = state[3];
    temp[11] = state[7];
    temp[15] = state[11];

    System.arraycopy(temp, 0, state, 0, 16);
}
```

This method performs the ShiftRows transformation by cyclically shifting the bytes in each row.

3.4.3 MixColumns / InvMixColumns

```
private static void mixColumns(byte[] state) {
    for (int i = 0; i < 4; i++) {
        byte a = state[i * 4];
```

```
        byte b = state[i * 4 + 1];
        byte c = state[i * 4 + 2];
        byte d = state[i * 4 + 3];

        state[i * 4] = (byte) (gmul(a, 2) ^ gmul(b, 3) ^
            ↪  c ^ d);
        state[i * 4 + 1] = (byte) (a ^ gmul(b, 2) ^ gmul
            ↪ (c, 3) ^ d);
        state[i * 4 + 2] = (byte) (a ^ b ^ gmul(c, 2) ^
            ↪ gmul(d, 3));
        state[i * 4 + 3] = (byte) (gmul(a, 3) ^ b ^ c ^
            ↪ gmul(d, 2));
    }
}
```

This method performs the MixColumns transformation by multiplying each column in the GF(2^8) field.

3.4.4 AddRoundKey

```
private static void addRoundKey(byte[] state, byte[]
    ↪ roundKey) {
    for (int i = 0; i < 16; i++) {
        state[i] ^= roundKey[i];
    }
}
```

This method XORs the state with the round key.

3.5 Key Expansion

The `keyExpansion()` method generates the round keys:

```
private static byte[][] keyExpansion(byte[] key) {
    byte[][] expandedKey = new byte[ROUNDS + 1][16];

    // First round key is the original key
    System.arraycopy(key, 0, expandedKey[0], 0, 16);

    // Generate the remaining round keys
```

```
    for (int round = 1; round <= ROUNDS; round++) {
        // Start with the previous round key
        byte[] prevKey = expandedKey[round - 1];
        byte[] newKey = new byte[16];

        // Copy first 12 bytes from previous key
        System.arraycopy(prevKey, 4, newKey, 0, 12);

        // Process the last 4 bytes
        byte[] temp = new byte[4];
        System.arraycopy(prevKey, 12, temp, 0, 4);

        // Rotate word
        byte t = temp[0];
        temp[0] = temp[1];
        temp[1] = temp[2];
        temp[2] = temp[3];
        temp[3] = t;

        // Apply S-box
        for (int i = 0; i < 4; i++) {
            temp[i] = S_BOX[temp[i] & 0xFF];
        }

        // XOR with Rcon
        temp[0] ^= RCON[round - 1];

        // XOR with previous words
        for (int i = 0; i < 4; i++) {
            newKey[i] = (byte) (prevKey[i] ^ temp[i]);
        }

        for (int i = 4; i < 16; i++) {
            newKey[i] = (byte) (newKey[i - 4] ^ prevKey[
                ↪ i]);
        }

        expandedKey[round] = newKey;
    }

```

```
47      return expandedKey;
48  }
```

This method expands the 16-byte key into 11 round keys (for AES-128) following these steps:

1. The first round key is the original key
2. For each subsequent round:
    - Take the last 4 bytes of the previous round key
    - Rotate the bytes left by one position
    - Apply the S-box to each byte
    - XOR the first byte with the round constant (Rcon)
    - XOR this transformed word with the first word of the previous round key
    - Generate the remaining words by XORing with the corresponding word from the previous round key

3.6 Decryption Process

The decryption process reverses the encryption operations:

```
1  public static String decrypt(String ciphertext, String
       ↪ keyString,
2                               boolean isCBC) {
3      // Decode Base64 input
4      byte[] input = Base64.getDecoder().decode(ciphertext
           ↪ );
5      byte[] key = prepareKey(keyString);
6
7      // Generate key schedule
8      byte[][] keySchedule = keyExpansion(key);
9
10     // Process each block
11     byte[] output = new byte[input.length];
12     byte[] iv = new byte[BLOCK_SIZE];  // Initialization
           ↪  vector for CBC
```

```
13
14      for (int i = 0; i < input.length; i += BLOCK_SIZE) {
15          byte[] block = new byte[BLOCK_SIZE];
16          System.arraycopy(input, i, block, 0, BLOCK_SIZE)
                ↪ ;
17
18          // Store current block for CBC mode (before
                ↪ decryption)
19          byte[] prevBlock = new byte[BLOCK_SIZE];
20          if (isCBC) {
21              System.arraycopy(block, 0, prevBlock, 0,
                    ↪ BLOCK_SIZE);
22          }
23
24          // Decrypt the block
25          block = decryptBlock(block, keySchedule);
26
27          // For CBC mode, XOR with previous ciphertext
                ↪ block (or IV)
28          if (isCBC) {
29              if (i == 0) {
30                  xorBlocks(block, iv);
31              } else {
32                  byte[] temp = new byte[BLOCK_SIZE];
33                  System.arraycopy(input, i - BLOCK_SIZE,
                        ↪ temp, 0, BLOCK_SIZE);
34                  xorBlocks(block, temp);
35              }
36          }
37
38          // Copy to output
39          System.arraycopy(block, 0, output, i, BLOCK_SIZE
                ↪ );
40      }
41
42      // Remove PKCS#7 padding
43      output = removePKCS7Padding(output);
44
45      // Convert to string
46      return new String(output, StandardCharsets.UTF_8);
```

```
}
```

3.7 Padding Functions

The AES implementation includes PKCS#7 padding functions:

```
private static byte[] padPKCS7(byte[] input) {
    int padLength = BLOCK_SIZE - (input.length %
       ↪ BLOCK_SIZE);
    if (padLength == 0) {
        padLength = BLOCK_SIZE;  // If already a
           ↪ multiple, add full block
    }

    byte[] padded = new byte[input.length + padLength];
    System.arraycopy(input, 0, padded, 0, input.length);
    // Fill padding bytes with the padding length value
    for (int i = input.length; i < padded.length; i++) {
        padded[i] = (byte) padLength;
    }

    return padded;
}

private static byte[] removePKCS7Padding(byte[] input) {
    int padLength = input[input.length - 1] & 0xFF;

    // Validate padding
    if (padLength <= 0 || padLength > BLOCK_SIZE) {
        throw new IllegalArgumentException("Invalid␣PKCS
           ↪ #7␣padding");
    }

    // Check that all padding bytes have the correct
       ↪ value
    for (int i = input.length - padLength; i < input.
       ↪ length; i++) {
        if ((input[i] & 0xFF) != padLength) {
```

```
28                throw new IllegalArgumentException("Invalid␣
                     ↪ PKCS#7␣padding");
29            }
30        }
31
32        byte[] unpadded = new byte[input.length - padLength
             ↪ ];
33        System.arraycopy(input, 0, unpadded, 0, unpadded.
             ↪ length);
34
35        return unpadded;
36    }
```

These methods handle PKCS#7 padding, which adds bytes with the value of the padding length to make the input a multiple of the block size.

Chapter 4

Diffie-Hellman Exchange Implementation (DHE.java)

4.1 Class Structure and Fields

The DHE implementation is organized as a Java class with the following key fields:

```
private static final SecureRandom RANDOM = new
    ↪ SecureRandom();

private BigInteger g; // generator
private BigInteger p; // prime modulus
private int gBits; // bit length of generator
private int pBits; // bit length of prime
```

These fields store the Diffie-Hellman parameters and configuration.

4.2 Constructor and Parameter Generation

The class has a constructor and parameter generation method:

```
public DHE(int gBits, int pBits) {
    this.gBits = gBits;
    this.pBits = pBits;
    generateParameters();
}
```

```
6
7  private void generateParameters() {
8      // Generate a prime p
9      p = BigInteger.probablePrime(pBits, RANDOM);
10
11     // Use a common generator (5 is often used for
           ↪ demonstration)
12     g = BigInteger.valueOf(5);
13 }
```

The constructor initializes the bit lengths and calls `generateParameters()`, which creates a random prime modulus `p` and sets the generator `g` to 5 (a common choice).

4.3 Key Exchange Methods

The DHE implementation includes methods for each step of the key exchange process:

```
1  public BigInteger generatePrivateBase(int bits) {
2      BigInteger base = new BigInteger(bits, RANDOM);
3      return base;
4  }
5
6  public BigInteger computePublicValue(BigInteger
       ↪ privateBase) {
7      // Calculate g^privateBase mod p
8      BigInteger publicValue = g.modPow(privateBase, p);
9      return publicValue;
10 }
11
12 public BigInteger computeSharedSecret(BigInteger
       ↪ privateBase,
13                                        BigInteger
                                             ↪ otherPublicValue
                                             ↪ ) {
14     // Calculate (otherPublicValue)^privateBase mod p
15     BigInteger sharedSecret = otherPublicValue.modPow(
           ↪ privateBase, p);
16     return sharedSecret;
```

```
}
```

These methods implement:

- `generatePrivateBase()`: Creates a random private value
- `computePublicValue()`: Calculates the public value as `g^privateBase mod p`
- `computeSharedSecret()`: Derives the shared secret from a private value and the other party's public value as `otherPublicValue^privateBase mod p`

4.4 Two-Party Key Exchange

The `main()` method demonstrates a two-party Diffie-Hellman key exchange:

```
// Two-party Diffie-Hellman key exchange
DHE dhe = new DHE(512, 2048);

// Alice's calculations
BigInteger a = dhe.generatePrivateBase(512);
BigInteger A = dhe.computePublicValue(a);

// Bob's calculations
BigInteger b = dhe.generatePrivateBase(512);
BigInteger B = dhe.computePublicValue(b);

// Shared secret calculation
BigInteger aliceSecret = dhe.computeSharedSecret(a, B);
BigInteger bobSecret = dhe.computeSharedSecret(b, A);
```

This demonstrates:

1. Creating a DHE instance with 512-bit private values and a 2048-bit prime
2. Alice generating a private value `a` and computing her public value `A = g^a mod p`
3. Bob generating a private value `b` and computing his public value `B = g^b mod p`

4. Alice computing the shared secret as `B^a mod p`
5. Bob computing the shared secret as `A^b mod p`
6. Verifying that both parties derive the same shared secret

4.5 Three-Party Key Exchange

The implementation also demonstrates a three-party Diffie-Hellman key exchange:

```
// Party X's calculations
BigInteger x = dhe3.generatePrivateBase(512);
BigInteger X = dhe3.computePublicValue(x);

// Party Y's calculations
BigInteger y = dhe3.generatePrivateBase(512);
BigInteger Y = dhe3.computePublicValue(y);

// Party Z's calculations
BigInteger z = dhe3.generatePrivateBase(512);
BigInteger Z = dhe3.computePublicValue(z);

// Intermediate values
BigInteger XY = dhe3.computeSharedSecret(x, Y);
BigInteger YZ = dhe3.computeSharedSecret(y, Z);
BigInteger ZX = dhe3.computeSharedSecret(z, X);

// Final shared secrets
BigInteger secretX = dhe3.computeSharedSecret(x, YZ);
BigInteger secretY = dhe3.computeSharedSecret(y, ZX);
BigInteger secretZ = dhe3.computeSharedSecret(z, XY);
```

This demonstrates:

1. Three parties (X, Y, Z) generating private values and public values
2. Computing intermediate values:
   - `XY = Y^x mod p` (X computes this using Y's public value)
   - `YZ = Z^y mod p` (Y computes this using Z's public value)

- `ZX = X^z mod p` (Z computes this using X's public value)

3. Computing final shared secrets:
   - X computes `secretX = YZ^x mod p = g^(y*z*x) mod p`
   - Y computes `secretY = ZX^y mod p = g^(z*x*y) mod p`
   - Z computes `secretZ = XY^z mod p = g^(x*y*z) mod p`
4. Verifying that all three parties derive the same shared secret

4.6 Alternative Three-Party Method

The implementation also demonstrates an alternative approach for three-party key exchange:

```
// Calculate g^xyz directly using a different order of operations
// Party X
BigInteger temp1 = dhe3.computePublicValue(y); // g^y
BigInteger temp2 = dhe3.computeSharedSecret(z, temp1); // (g^y)^z = g^yz
BigInteger YZ_X = dhe3.computeSharedSecret(x, temp2); // (g^yz)^x = g^xyz

// Party Y
temp1 = dhe3.computePublicValue(z); // g^z
temp2 = dhe3.computeSharedSecret(x, temp1); // (g^z)^x = g^zx
BigInteger ZX_Y = dhe3.computeSharedSecret(y, temp2); // (g^zx)^y = g^zxy

// Party Z
temp1 = dhe3.computePublicValue(x); // g^x
temp2 = dhe3.computeSharedSecret(y, temp1); // (g^x)^y = g^xy
BigInteger XY_Z = dhe3.computeSharedSecret(z, temp2); // (g^xy)^z = g^xyz
```

This demonstrates an alternative approach that yields the same result:

1. Party X computes the shared secret as `(g^yz)^x = g^xyz`
2. Party Y computes the shared secret as `(g^zx)^y = g^zxy`
3. Party Z computes the shared secret as `(g^xy)^z = g^xyz`
4. The code verifies that both approaches yield the same result

Chapter 5

Project 4: Implementation Details

5.1 Algorithm Selection and Implementation Decisions

5.1.1 RSA Implementation Decisions

The RSA implementation makes several important design choices:

- Using a default key size of 1024 bits as a balance between security and performance for educational purposes
- Selecting 65537 (0x10001) as the default public exponent `e`, which is a common choice in practice
- Using Java's `BigInteger` class for large integer arithmetic
- Implementing both encryption/decryption and signature/verification functions
- Using simple input validation to ensure messages are smaller than the modulus

5.1.2 AES Implementation Decisions

The AES implementation makes several important design choices:

- Implementing both ECB and CBC modes of operation
- Including a debug flag to enable detailed output for educational purposes
- Using PKCS#7 padding for proper block alignment
- Base64-encoding the ciphertext output for easier handling
- Including all internal AES operations without using Java's built-in crypto libraries
- Using a 128-bit key length (AES-128) with 10 rounds

5.1.3 DHE Implementation Decisions

The DHE implementation makes several important design choices:

- Using configurable bit lengths for the private values and prime modulus
- Using 5 as the generator value (a common choice for demonstrations)
- Implementing both two-party and three-party key exchange variants
- Using Java's `BigInteger` class for modular exponentiation
- Including comprehensive test cases in the main method

5.2 Technical Implementation Challenges

5.2.1 RSA Implementation Challenges

Several technical challenges were addressed in the RSA implementation:

- Generating large prime numbers efficiently
- Calculating the modular multiplicative inverse for the private key
- Ensuring the public and private exponents work correctly
- Handling large integers and modular arithmetic efficiently
- Validating input messages against the modulus size

5.2.2 AES Implementation Challenges

Several technical challenges were addressed in the AES implementation:

- Implementing the S-box and inverse S-box correctly
- Ensuring proper implementation of the ShiftRows operation
- Implementing Galois Field multiplication for MixColumns
- Managing the key expansion process with correct round constant application
- Implementing reverse operations for decryption
- Correct handling of PKCS#7 padding for arbitrary message lengths
- Proper CBC mode implementation with initialization vector

5.2.3 DHE Implementation Challenges

Several technical challenges were addressed in the DHE implementation:

- Generating a secure prime number for the modulus
- Ensuring efficient modular exponentiation for large values
- Implementing the three-party key exchange correctly
- Verifying that all parties arrive at the same shared secret
- Demonstrating alternative approaches to multi-party key exchange

5.3 Code Generation Process with Claude 3.7

This code was generated through an interaction with Claude 3.7, an AI language model by Anthropic. The process involved:

1. Specifying the cryptographic algorithms to be implemented
2. Requesting educational implementations with detailed comments

3. Refining the code structure and implementation details
4. Adding test cases to demonstrate functionality
5. Documenting the code with detailed explanations

Claude 3.7 was able to generate comprehensive implementations of these cryptographic algorithms with detailed explanations of the underlying mathematical operations. The model provided code that correctly implements the fundamental aspects of each algorithm, with thorough comments explaining each step of the process.

5.4 Implementation Insights

5.4.1 RSA Insights

The RSA implementation provides insights into:

- How public and private keys are mathematically related
- The role of prime numbers in cryptographic security
- How modular exponentiation enables encryption and decryption
- The mathematical basis for digital signatures
- The performance implications of different key sizes

5.4.2 AES Insights

The AES implementation provides insights into:

- The internal structure of a modern symmetric block cipher
- How substitution and permutation networks provide confusion and diffusion
- The importance of modes of operation for securing multiple blocks
- The role of key expansion in creating round keys
- How operations are reversed for decryption

5.4.3 DHE Insights

The DHE implementation provides insights into:

- How shared secrets can be established over insecure channels
- The mathematical principles behind key exchange
- How modular exponentiation creates one-way functions
- The extension of key exchange to multiple parties
- How different computational paths can lead to the same shared secret

Chapter 6

Conclusion

This technical documentation has provided a detailed walkthrough of the Java implementations of RSA, AES, and Diffie-Hellman key exchange algorithms. The documentation has focused on explaining the code structure, function implementations, and algorithmic processes that make each cryptographic operation work.

These implementations serve as educational tools for understanding the inner workings of fundamental cryptographic algorithms. Each implementation includes detailed comments, proper structure, and test cases that demonstrate functionality.

The code was generated with the assistance of Claude 3.7, which was able to produce comprehensive implementations that correctly implement the mathematical operations required for each algorithm. These implementations can serve as a foundation for understanding the principles behind modern cryptographic systems.