MULTIMEDIA UNIVERSITY

CCS6314 Cryptography and Data Security Assignment

Group B

Group Member Lists:

- Wong Zhi Lin 1201203299
- Cheah Wei Yan 1201203703
- Chay Chun Keat 1211200378
- Dylan Cheng Zhi Xin 1211200579

Introduction

Cryptography: From Past to Present

- Historically protected communication/data.
- Evolved from simple classical ciphers to advanced systems.
- Classical methods (substitution, transposition) now vulnerable.
- Modern cryptography uses complex math & dual encryption.

Project Overview: Cryptography CLI Tool

- CLI tool for experimenting with encryption/decryption.
- Implements Playfair, Rail Fence, and Product Ciphers.
- RSA key exchange for secure communication.
- AES/Triple DES for symmetric encryption.

Investigation Focus

- Testing speed and security of encryption/decryption protocols.
- Analyzing weaknesses of classical ciphers.
- Evaluating security of modern hybrid encryption.
- Studying impact of data errors on encrypted data.

Key Benefit

- Provides a practical, real-world understanding of cryptography.

Background Study

Background study in cryptography involves understanding its evolution from classical methods to modern standards.

Classical Cryptography:

- Relies on substitution and transposition ciphers.
- Substitution ciphers replace plaintext characters with different symbols based on a key.
- Transposition ciphers shuffle the characters of the message.
- Examples include the Playfair cipher and Rail Fence cipher.
- Classical methods are vulnerable to frequency analysis and brute-force attacks due to advances in computing.

Playfair Cipher:

- A substitution cipher that encrypts pairs of letters using a 5x5 key matrix.
- More secure than simple monoalphabetic ciphers but still vulnerable to modern computational techniques.

Rail Fence Cipher:

- A transposition cipher that writes plaintext in a zigzag pattern and reads it row by row.
- Disrupts character placement but can be decrypted relatively easily.

Product Ciphers:

- Combine Playfair and Rail Fence ciphers for stronger protection using both substitution and transposition.

Modern Cryptography:

- Uses symmetric and asymmetric encryption methods for stronger security.
- Symmetric encryption uses the same key for encryption and decryption, examples include AES and Triple DES[1].
- Asymmetric encryption (public-key cryptography) uses a pair of keys: a public key for encryption and a private key for decryption; an example is RSA.
- RSA solves the key distribution problem by securely transporting symmetric keys.

Data Integrity:

- Bit errors during ciphertext transmission can compromise security.
- Encryption methods like AES in CBC mode can be significantly affected by single-bit errors.
- Secure communication systems must be designed to handle transmission errors.

Project Application:

- CLI-based system to implement and evaluate cryptographic techniques.
- Interactive user testing of encryption, decryption, and key exchange processes.
- Analysis of classical and modern methods to understand their strengths, weaknesses, and practical uses.

Description of the Concepts, Methods, and Algorithms Used

Classical Symmetric Ciphers :

Playfair Cipher (Substitution Cipher)

1. **Matrix**: 5×5 matrix (e.g., keyword MONARCHY), excludes 'J.'
2. **Encryption Process**:
   - Convert plaintext into letter pairs; add extra letters if needed.
   - Same row: Replace letters with next in row.
   - Same column: Replace letters with the one below.
   - Rectangle: Swap diagonally.
3. **Decryption**: Reverse the encryption steps.
4. **Security**:
   - More secure than monoalphabetic ciphers.
   - Vulnerable to digraph frequency analysis and brute-force attacks.

Rail Fence Cipher (Transposition Cipher)

- **Encryption Process**:
  - Choose rail depth (number of rows).
  - Write plaintext in zigzag pattern across rows.
  - Read ciphertext row-wise.
- **Decryption**:
  - Recreate zigzag pattern and read letters in the same sequence.
- **Security**:
  - Harder to crack than simple ciphers.
  - Vulnerable to pattern recognition attacks.
  - Can be combined with substitution ciphers for better security.

Product Cipher (Playfair + Rail Fence)

- **Process**: Combines Playfair (substitution) and Rail Fence (transposition).
- **Security Benefits**:
  - Playfair disguises letter frequencies.
  - Rail Fence disrupts character order.
  - More resistant to traditional cryptanalysis.

Modern Cryptograpy: Hybrid Encryption Approach :

RSA Algorithm (Asymmetric Key Exchange)

1. **Key Generation**:
   - Choose large primes (p, q); compute $n = p \times q$.
   - Calculate $\varphi(n) = (p-1)(q-1)$.
   - Public key: Choose e (commonly 65537).
   - Private key: $d = e^{-1} \bmod \varphi(n)$.
2. **Encryption**: C = M^e mod n.
3. **Decryption**: M = C^d mod n.
4. **Security**:
   - Based on the difficulty of factoring large numbers.
   - Used for secure key exchange (not ideal for large data encryption).

AES (Advanced Encryption Standard)

1. **Process**:
   - Key Expansion: Generates multiple round keys.
   - Initial Round: Adds first-round key.
   - Main Rounds (10 rounds for AES-128):
     - **SubBytes**: Byte substitution using S-box.
     - **ShiftRows**: Left shift for diffusion.
     - **MixColumns**: Column mixing for further diffusion.
     - **AddRoundKey**: XOR with the round key.
   - Final Round: No MixColumns.
2. **Security**:
   - Resistant to brute-force due to large key sizes (128, 192, 256 bits).
   - Efficient and widely used in SSL/TLS and other applications.

Implementation Details

- Development Environment and Tools
  - Language – Java
  - Java Version – JDK 8 or later
- Libraries used – import java.security.SecureRandom;
  - import java.util.Base64;
  - import java.util.Scanner;
  - import javax.crypto.SecretKey;
  - import java.time.Duration;
  - import java.time.Instant;
  - import java.math.BigInteger;
  - import javax.crypto.spec.SecretKeySpec;
  - import javax.crypto.Cipher;
  - import javax.crypto.spec.IvParameterSpec;

Implementation of Classical Symmetric Ciphers

- Playfair Cipher Implementation

```
// Playfair Cipher Encryption
public static String playfairCipherEncrypt(String text, String key) {
    text = text.replaceAll("[^A-Za-z]", "").toUpperCase().replace("J", "I");
    key = key.replaceAll("[^A-Za-z]", "").toUpperCase().replace("J", "I");

    char[][] matrix = generatePlayfairMatrix(key);
    StringBuilder encryptedText = new StringBuilder();
    StringBuilder formattedText = new StringBuilder();

    long startTime = System.nanoTime(); // Start timing

    // Format text: prevent identical letter pairs & ensure even length
    for (int i = 0; i < text.length(); i++) {
        formattedText.append(text.charAt(i));
        if (i + 1 < text.length() && text.charAt(i) == text.charAt(i + 1)) {
            formattedText.append('X');  // Insert 'X' for duplicate letters
        }
    }
    if (formattedText.length() % 2 != 0) formattedText.append('X'); // Ensure even length

    System.out.println("Formatted Text for Playfair Encryption: " + formattedText.toString());
```

```java
for (int i = 0; i < formattedText.length(); i += 2) {
    char a = formattedText.charAt(i), b = formattedText.charAt(i + 1);
    int[] posA = findPosition(matrix, a), posB = findPosition(matrix, b);

    if (posA[0] == -1 || posB[0] == -1) {
        System.err.println("Error: Character not found in Playfair Matrix.");
        return null;
    }

    if (posA[0] == posB[0]) {  // Same row: shift right
        encryptedText.append(matrix[posA[0]][(posA[1] + 1) % 5])
                     .append(matrix[posB[0]][(posB[1] + 1) % 5]);
    } else if (posA[1] == posB[1]) { // Same column: shift down
        encryptedText.append(matrix[(posA[0] + 1) % 5][posA[1]])
                     .append(matrix[(posB[0] + 1) % 5][posB[1]]);
    } else { // Rectangle swap
        encryptedText.append(matrix[posA[0]][posB[1]])
                     .append(matrix[posB[0]][posA[1]]);
    }
}

long endTime = System.nanoTime(); // End timing
long timeElapsed = (endTime - startTime) / 1_000_000; // Convert nanoseconds to milliseconds

System.out.println("Encrypted Text: " + encryptedText.toString());
System.out.println("Encryption Time: " + timeElapsed + " ms");
return encryptedText.toString();
```

```java
    char[][] matrix = generatePlayfairMatrix(key);

    // Ensure the text length is even
    if (text.length() % 2 != 0) {
        System.err.println("Error: Ciphertext length is not even. Possible corruption.");
        return null;
    }

    Instant startTime = Instant.now(); // Start timing
    String decryptedText = processPlayfair(text, matrix, false);
    Instant endTime = Instant.now(); // End timing
    long timeElapsed = Duration.between(startTime, endTime).toMillis(); // Calculate time in milliseconds
    System.out.println("Decryption Output: " + decryptedText);
    System.out.println("Time taken: " + timeElapsed + " ms");

    return decryptedText;
}

private static String processPlayfair(String text, char[][] matrix, boolean encrypt) {
    StringBuilder result = new StringBuilder();
    int shift = encrypt ? 1 : -1; // Right for encryption, left for decryption

    for (int i = 0; i < text.length(); i += 2) {
        char a = text.charAt(i), b = text.charAt(i + 1);
        int[] posA = findPosition(matrix, a), posB = findPosition(matrix, b);

        // SAFETY CHECK: Ensure valid character positions
        if (posA[0] == -1 || posB[0] == -1) {
            System.err.println("Error: One or both characters not found in Playfair Matrix.");
```

```java
        if (posA[0] == posB[0]) {  // Same row: shift LEFT for decryption
            result.append(matrix[posA[0]][(posA[1] + shift + 5) % 5])
                  .append(matrix[posB[0]][(posB[1] + shift + 5) % 5]);
        } else if (posA[1] == posB[1]) { // Same column: shift UP for decryption
            result.append(matrix[(posA[0] + shift + 5) % 5][posA[1]])
                  .append(matrix[(posB[0] + shift + 5) % 5][posB[1]]);
        } else { // Rectangle swap
            result.append(matrix[posA[0]][posB[1]])
                  .append(matrix[posB[0]][posA[1]]);
        }
    }

    String decryptedText = result.toString();

    // **Fix Artificial 'X' Removal**
    decryptedText = removeArtificialX(decryptedText);

    return decryptedText;
}
```

- Rail Fence Cipher Implementation

```java
// Rail Fence Cipher Encryption (Fixed Zigzag Implementation)
public static String railFenceEncrypt(String text) {
    int depth = 3;
    if (text.length() <= 1) return text;

    Instant startTime = Instant.now(); // Start timing

    StringBuilder[] rails = new StringBuilder[depth];
    for (int i = 0; i < depth; i++) rails[i] = new StringBuilder();

    int row = 0, direction = 1;
    for (char c : text.toCharArray()) {
        rails[row].append(c);
        row += direction;
        if (row == 0 || row == depth - 1) direction *= -1;
    }

    StringBuilder encryptedText = new StringBuilder();
    for (StringBuilder sb : rails) encryptedText.append(sb);

    Instant endTime = Instant.now(); // End timing
    long timeElapsed = Duration.between(startTime, endTime).toMillis(); // Calculate time in milliseconds
    System.out.println("Encrypted Text: " + encryptedText.toString());
    System.out.println("Time taken: " + timeElapsed + " ms");
    return encryptedText.toString();
}
```

```java
// Rail Fence Cipher Decryption
public static String railFenceDecrypt(String text) {
    int depth = 3;
    if (text.length() <= 1) return text;

    Instant startTime = Instant.now(); // Start timing

    char[] decryptedText = new char[text.length()];
    int[] pattern = new int[text.length()];
    int row = 0, direction = 1;

    // Step 1: Determine the zig-zag pattern positions
    for (int i = 0; i < text.length(); i++) {
        pattern[i] = row;
        row += direction;
        if (row == 0 || row == depth - 1) direction *= -1;
    }

    // Step 2: Count characters in each row
    int[] rowCounts = new int[depth];
    for (int r : pattern) rowCounts[r]++;

    // Step 3: Fill rows with correct characters
    StringBuilder[] rows = new StringBuilder[depth];
    for (int i = 0; i < depth; i++) rows[i] = new StringBuilder();

    int index = 0;
    for (int i = 0; i < depth; i++) {
        for (int j = 0; j < rowCounts[i]; j++) {
            rows[i].append(text.charAt(index++));
        }
    }
```

```java
    // Step 4: Read characters from zig-zag pattern to decrypt
    row = 0;
    direction = 1;
    index = 0;
    for (int i = 0; i < text.length(); i++) {
        decryptedText[i] = rows[pattern[i]].charAt(0);
        rows[pattern[i]].deleteCharAt(0);
    }

    Instant endTime = Instant.now(); // End timing
    long timeElapsed = Duration.between(startTime, endTime).toMillis(); // Calculate time in milliseconds
    System.out.println("Time taken: " + timeElapsed + " ms");
    System.out.println("Decrypted Text: " + new String(decryptedText));
    return new String(decryptedText);
}
```

- Product Cipher Implementation – Playfair + Rail Fence

```java
// Product Cipher (Playfair + Rail Fence)
public static String productCipherEncrypt(String text, String key) {
    String playfairEncrypted = playfairCipherEncrypt(text, key);
    System.out.println("After Playfair Cipher: " + playfairEncrypted);

    String railFenceEncrypted = railFenceEncrypt(playfairEncrypted);
    System.out.println("After Rail Fence Cipher: " + railFenceEncrypted);

    return railFenceEncrypted;
}

// Product Cipher Decryption
public static String productCipherDecrypt(String text, String key) {
    String railFenceDecrypted = railFenceDecrypt(text);
    System.out.println("After Rail Fence Decryption: " + railFenceDecrypted);

    String playfairDecrypted = playfairCipherDecrypt(railFenceDecrypted, key);
    System.out.println("After Playfair Decryption: " + playfairDecrypted);

    return playfairDecrypted;
}
```

Implementation of Modern Cryptograhy

- RSA Key Exchange Implementation

```
public static class ManualRSAEncryption {
    private static final int BIT_LENGTH = 1024;
    private static final SecureRandom random = new SecureRandom();

    private BigInteger n, e, d;

    public ManualRSAEncryption() {
        BigInteger p = BigInteger.probablePrime(BIT_LENGTH / 2, random);
        BigInteger q = BigInteger.probablePrime(BIT_LENGTH / 2, random);
        n = p.multiply(q);
        BigInteger phi = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));

        e = new BigInteger(val:"65537");
        d = e.modInverse(phi);
    }

    public BigInteger encryptRSA(BigInteger message) {
        return message.modPow(e, n);
    }

    public BigInteger decryptRSA(BigInteger ciphertext) {
        return ciphertext.modPow(d, n);
    }

    public BigInteger getPublicKey() {
        return e;
    }

    public BigInteger getModulus() {
        return n;
    }

    public static void generateRSAKeyPair(Scanner scanner) {
        System.out.println(x:"\n======================================================================");
        System.out.println(x:" SIMULATING PERSON A & B COMMUNICATION FOR RSA & AES HYBRID ENCRYPTION ");
        System.out.println(x:"=======================================================================");
```

```java
// Generate RSA Keys
ManualRSAEncryption rsa = new ManualRSAEncryption();
BigInteger publicKeyB = rsa.getPublicKey();
BigInteger modulusB = rsa.getModulus();

// Generate AES Key
SecretKey aesKey = generateAESKey();
String aesKeyBase64 = Base64.getEncoder().encodeToString(aesKey.getEncoded());
BigInteger aesKeyBigInt = new BigInteger(signum:1, aesKeyBase64.getBytes());
System.out.println("\n[Person A] AES Key (Base64): " + aesKeyBase64);

// Encrypt AES Key using RSA
BigInteger encryptedAESKey = aesKeyBigInt.modPow(publicKeyB, modulusB);
System.out.println("[Person A] Encrypted AES Key: " + encryptedAESKey);

// Decrypt AES Key using RSA
BigInteger decryptedAESKeyBigInt = encryptedAESKey.modPow(rsa.d, rsa.n);
byte[] decryptedBytes = decryptedAESKeyBigInt.toByteArray();
String decryptedAESBase64 = new String(decryptedBytes);
SecretKey originalAESKey = new SecretKeySpec(Base64.getDecoder().decode(decryptedAESBase6
System.out.println("[Person B] Decrypted AES Key (Base64): " + decryptedAESBase64);

// Encrypt Message Using AES
System.out.print(s:"\n[Person A] Enter message to encrypt: ");
String message = scanner.nextLine();
byte[] encryptedMessage = encryptAES(message.getBytes(), originalAESKey);
System.out.println(
        "[Person A] Encrypted Message (Base64): " + Base64.getEncoder().encodeToString(en

// Decrypt the Message
byte[] decryptedMessage = decryptAES(encryptedMessage, originalAESKey);
System.out.println("[Person B] Decrypted Message: " + new String(decryptedMessage));

simulateBitError(encryptedMessage, originalAESKey.getEncoded());
}
```

- AES Encryption Implementation – CBC Mode

```java
// AES Encryption Method
private static byte[] encryptAES(byte[] plaintext, SecretKey key) throws Exception {
    byte[] ivBytes = new byte[16];
    new SecureRandom().nextBytes(ivBytes); // Use random IV for security
    IvParameterSpec iv = new IvParameterSpec(ivBytes);

    Cipher cipher = Cipher.getInstance(transformation:"AES/CBC/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, key, iv);

    byte[] encrypted = cipher.doFinal(plaintext);

    // Combine IV and Ciphertext
    byte[] combined = new byte[ivBytes.length + encrypted.length];
    System.arraycopy(ivBytes, srcPos:0, combined, destPos:0, ivBytes.length);
    System.arraycopy(encrypted, srcPos:0, combined, ivBytes.length, encrypted.length);
    return combined;
}

// AES Decryption Method (with input validation)
private static byte[] decryptAES(byte[] ciphertext, SecretKey key) throws Exception {
    if (ciphertext.length < 16) {
        throw new IllegalArgumentException(s:"Ciphertext too short, cannot extract IV.");
    }

    byte[] iv = new byte[16];
    System.arraycopy(ciphertext, srcPos:0, iv, destPos:0, iv.length);
    IvParameterSpec ivSpec = new IvParameterSpec(iv);

    Cipher cipher = Cipher.getInstance(transformation:"AES/CBC/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);

    return cipher.doFinal(ciphertext, iv.length, ciphertext.length - iv.length);
}
```

```java
// Fix RSA-AES Key Exchange Conversion Issue
private static SecretKey generateAESKey() {
    try {
        javax.crypto.KeyGenerator keyGen = javax.crypto.KeyGenerator.getInstance(algorithm:"A
        keyGen.init(keysize:128);
        return keyGen.generateKey();
    } catch (Exception e) {
        throw new RuntimeException(message:"Error generating AES key", e);
    }
}
```

```java
// Generate AES Key
SecretKey aesKey = generateAESKey();
String aesKeyBase64 = Base64.getEncoder().encodeToString(aesKey.getEncoded());
BigInteger aesKeyBigInt = new BigInteger(signum:1, aesKeyBase64.getBytes());
System.out.println("\n[Person A] AES Key (Base64): " + aesKeyBase64);

// Encrypt AES Key using RSA
BigInteger encryptedAESKey = aesKeyBigInt.modPow(publicKeyB, modulusB);
System.out.println("[Person A] Encrypted AES Key: " + encryptedAESKey);

// Decrypt AES Key using RSA
BigInteger decryptedAESKeyBigInt = encryptedAESKey.modPow(rsa.d, rsa.n);
byte[] decryptedBytes = decryptedAESKeyBigInt.toByteArray();
String decryptedAESBase64 = new String(decryptedBytes);
SecretKey originalAESKey = new SecretKeySpec(Base64.getDecoder().decode(decryptedAESBase6
System.out.println("[Person B] Decrypted AES Key (Base64): " + decryptedAESBase64);

// Encrypt Message Using AES
System.out.print(s:"\n[Person A] Enter message to encrypt: ");
String message = scanner.nextLine();
byte[] encryptedMessage = encryptAES(message.getBytes(), originalAESKey);
System.out.println(
        "[Person A] Encrypted Message (Base64): " + Base64.getEncoder().encodeToString(en

// Decrypt the Message
byte[] decryptedMessage = decryptAES(encryptedMessage, originalAESKey);
System.out.println("[Person B] Decrypted Message: " + new String(decryptedMessage));

simulateBitError(encryptedMessage, originalAESKey.getEncoded());
```

Comparison and Discussion of the Result

PLAYFAIR CIPHER	RAIL FENCE CIPHER (DEPTH=3)	PRODUCT CIPHER (PLAYFAIR + RAIL FENCE)
• Substitution-based encryption using a 5×5 matrix • Fast encryption & decryption (~1 ms) • Weakness: Susceptible to frequency analysis	• Transposition-based encryption in a zigzag pattern • Very fast encryption & decryption (~0 ms) • Weakness: Easily broken by brute-force	• Combination of substitution & transposition • Improved security over individual ciphers • Weakness: Still vulnerable to modern cryptanalysis

Comparison and Discussion of the Result

RSA Key Exchange	AES (CBC Mode)
• Used for secure key distribution (asymmetric encryption) • Based on the difficulty of factoring large prime numbers • Weakness: Slow and not suitable for encrypting large data	• Symmetric encryption with high security and fast processing • Encrypts data in fixed 128-bit blocks • Weakness: Sensitive to bit errors (error propagation in CBC mode)

Comparison and Discussion of the Result

- Classical ciphers (Playfair, Rail Fence) are fast but weak
- Product Cipher improves security but still not practical for modern use
- RSA is great for key exchange but slow for large data
- AES is efficient & highly secure, but sensitive to bit errors
- Best practice: Use hybrid encryption (RSA for key exchange + AES for data encryption)

Conclusion

- Hybrid encryption (RSA for key exchange + AES for data encryption) is the best approach for secure digital communication
- Cryptographic methods must balance security, performance, and application needs
- Encryption research is crucial for enhancing security measures and combating emerging threats
- Understanding cryptography helps in making informed decisions for real-world cybersecurity applications