### **1. Hàm subBytes**

#### **Chức năng:**

Thay thế từng byte trong state bằng một giá trị tương ứng từ bảng S-box (một bảng ánh xạ được sử dụng trong AES).

#### **Logic:**

java

state[i][j] = (byte) ((state[i][j] & 0x0F) ^ 0x63);

* state[i][j] & 0x0F: Lấy 4 bit thấp của byte trong state.
* ^ 0x63: XOR với hằng số 0x63 để tạo ra giá trị thay thế.

Đây là một cách đơn giản hóa logic của S-box thật sự trong AES.

#### **Vai trò:**

* Tăng độ phức tạp của dữ liệu đầu vào, làm cho dữ liệu sau khi mã hóa khó đoán.

### **2. Hàm shiftRows**

#### **Chức năng:**

Dịch chuyển các hàng của ma trận state theo một quy tắc cụ thể. Đây là một bước trong thuật toán AES nhằm tăng độ khuếch tán.

#### **Logic:**

java

for (int i = 1; i < 4; i++) {  
 byte[] row = Arrays.copyOf(state[i], 4); // Sao chép hàng hiện tại  
 for (int j = 0; j < 4; j++) {  
 state[i][j] = row[(j + i) % 4]; // Dịch hàng i sang trái i vị trí  
 }  
}

* Hàng thứ nhất (hàng 0) không thay đổi.
* Hàng thứ hai (hàng 1) dịch 1 vị trí.
* Hàng thứ ba (hàng 2) dịch 2 vị trí.
* Hàng thứ tư (hàng 3) dịch 3 vị trí.

#### **Vai trò:**

* Tạo ra sự khác biệt giữa các hàng của ma trận state, tăng độ khuếch tán của dữ liệu.

### **3. Hàm mixColumns**

#### **Chức năng:**

Trộn các cột của ma trận state bằng cách nhân trong trường Galois. Bước này giúp các byte trong cùng một cột ảnh hưởng lẫn nhau.

#### **Logic:**

java

state[0][c] = (byte) (galoisMult(col[0], 2) ^ galoisMult(col[1], 3) ^ col[2] ^ col[3]);  
state[1][c] = (byte) (col[0] ^ galoisMult(col[1], 2) ^ galoisMult(col[2], 3) ^ col[3]);  
state[2][c] = (byte) (col[0] ^ col[1] ^ galoisMult(col[2], 2) ^ galoisMult(col[3], 3));  
state[3][c] = (byte) (galoisMult(col[0], 3) ^ col[1] ^ col[2] ^ galoisMult(col[3], 2));

* Hàm galoisMult thực hiện phép nhân trên trường Galois (GF(2^8)).
* Mỗi giá trị mới của cột là sự kết hợp tuyến tính của các byte trong cột, với trọng số xác định bởi thuật toán AES.

#### **Vai trò:**

* Tăng cường tính khuếch tán giữa các byte trong cùng một cột.

### **4. Hàm galoisMult**

#### **Chức năng:**

Thực hiện phép nhân của hai giá trị trong trường Galois GF(2^8).

#### **Logic:**

java

for (int i = 0; i < 8; i++) {  
 if ((b & 1) != 0) {  
 p ^= a; // Nếu bit thấp nhất của b là 1, XOR giá trị a vào kết quả  
 }  
 hiBitSet = (byte) (a & 0x80); // Kiểm tra bit cao nhất của a  
 a <<= 1; // Dịch trái a 1 bit  
 if (hiBitSet != 0) {  
 a ^= 0x1b; // XOR với hằng số giảm để đảm bảo kết quả nằm trong GF(2^8)  
 }  
 b >>= 1; // Dịch phải b 1 bit  
}

* Phép nhân trong GF(2^8) là không giống với phép nhân thông thường, và cần đảm bảo kết quả nằm trong trường bằng cách dùng phép XOR với hằng số giảm 0x1b.

#### **Vai trò:**

* Cung cấp phép toán cơ bản cho bước mixColumns.

### **5. Hàm addRoundKey**

#### **Chức năng:**

XOR ma trận state với khóa vòng (round key). Đây là bước duy nhất trong AES trực tiếp sử dụng khóa.

#### **Logic:**

java

state[i][j] ^= key[i \* 4 + j];

* Mỗi byte trong state được XOR với byte tương ứng trong round key.

#### **Vai trò:**

* Thêm tính bí mật vào mỗi vòng của AES bằng cách trộn dữ liệu với round key.

### **6. Hàm generateRoundKey**

#### **Chức năng:**

Sinh round key cho từng vòng dựa trên khóa gốc (giản lược trong mã này).

#### **Logic:**

java

for (int i = 0; i < BLOCK\_SIZE; i++) {  
 roundKey[i] ^= round; // XOR mỗi byte của key với chỉ số vòng  
}

* Thực hiện phép XOR giữa khóa ban đầu với số vòng hiện tại (round).

#### **Vai trò:**

* Trong AES thực sự, round key được tạo qua quá trình phức tạp hơn (bao gồm bước thay thế byte, dịch hàng, và XOR). Ở đây, mã chỉ đơn giản XOR để minh họa.

### **Tóm tắt**

Các hàm này mô phỏng cơ chế hoạt động của AES:

1. **subBytes**: Thay thế byte bằng S-box.
2. **shiftRows**: Dịch hàng để tăng khuếch tán.
3. **mixColumns**: Trộn cột để khuếch tán thêm.
4. **addRoundKey**: XOR với khóa vòng để tăng tính bí mật.
5. **galoisMult**: Hỗ trợ phép toán trong GF(2^8) cho bước mixColumns.
6. **generateRoundKey**: Sinh khóa vòng (giản lược).

Mặc dù mã này không tuân thủ chính xác tiêu chuẩn AES, nó cung cấp một cách tiếp cận dễ hiểu về các bước trong thuật toán.

package enc;  
  
import java.security.SecureRandom;  
import java.util.Arrays;  
  
public class AESCFB {  
 // 16 byte cho kích thước khối của AES (AES hoạt động trên từng khối 16 byte)  
 private static final int *BLOCK\_SIZE* = 16; // 16 bytes for AES block size  
  
 public static void main(String[] args) {  
 // Sinh khóa AES và IV ngẫu nhiên  
 byte[] key = *generateAESKey*();  
 byte[] iv = *generateIV*();  
  
 // Chuỗi plaintext  
 String plaintext = "This is a secret message.";  
 byte[] plaintextBytes = plaintext.getBytes();  
  
 // Encrypt  
 byte[] ciphertext = *encryptCFB*(plaintextBytes, key, iv);  
 System.*out*.println("Ciphertext: " + *bytesToHex*(ciphertext));// In ra ciphertext ở dạng hex  
  
 // Decrypt  
 byte[] decrypted = *decryptCFB*(ciphertext, key, iv);  
 System.*out*.println("Decrypted: " + new String(decrypted));  
 }  
  
 // Sinh một khóa AES ngẫu nhiên (128-bit, 16 byte)  
 // Generate a random AES key (128-bit)  
 public static byte[] generateAESKey() {  
 byte[] key = new byte[*BLOCK\_SIZE*];  
 new SecureRandom().nextBytes(key); // Dùng SecureRandom để đảm bảo độ an toàn  
 return key;  
 }  
  
 // Sinh một IV (vector khởi tạo) ngẫu nhiên (128-bit, 16 byte)  
 // Generate a random IV (128-bit)  
 public static byte[] generateIV() {  
 byte[] iv = new byte[*BLOCK\_SIZE*];  
 new SecureRandom().nextBytes(iv); // IV cũng cần ngẫu nhiên và duy nhất cho mỗi phiên  
 return iv;  
 }  
  
 // Hàm chuyển đổi byte array sang chuỗi hex để dễ đọc  
 // Convert bytes to a hex string for display  
 public static String bytesToHex(byte[] bytes) {  
 StringBuilder sb = new StringBuilder();  
 for (byte b : bytes) {  
 sb.append(String.*format*("%02x", b)); // Mỗi byte được biểu diễn bằng 2 ký tự hex  
 }  
 return sb.toString();  
 }  
  
 // Hàm mã hóa một khối (16 byte) bằng AES  
 // AES encryption of a single block (16 bytes)  
 public static byte[] aesEncryptBlock(byte[] block, byte[] key) {  
 // Đảm bảo kích thước của khối và khóa là 16 byte  
 if (block.length != *BLOCK\_SIZE* || key.length != *BLOCK\_SIZE*) {  
 throw new IllegalArgumentException("Block and key must be 16 bytes.");  
 }  
  
 // Bước 1: Chuyển đổi mảng block thành state (4x4)  
 // Example implementation of AES block encryption:  
 byte[][] state = new byte[4][4];  
  
  
 // Convert block to state array  
 for (int i = 0; i < 16; i++) {  
 state[i % 4][i / 4] = block[i];  
 }  
  
 // Bước 2: Thực hiện bước AddRoundKey ban đầu  
 // Initial AddRoundKey step  
 state = *addRoundKey*(state, key);  
  
 // Bước 3: Các vòng AES (giả sử 10 vòng cho AES-128)  
 // AES rounds (simplified here, typically 10 rounds for AES-128)  
 for (int round = 1; round <= 10; round++) {  
 state = *subBytes*(state);  
 state = *shiftRows*(state);  
 if (round < 10) {  
 state = *mixColumns*(state);  
 }  
 byte[] roundKey = *generateRoundKey*(round, key);  
 state = *addRoundKey*(state, roundKey);  
 }  
  
 // Bước 4: Chuyển lại state thành mảng byte  
 // Convert state array back to byte array  
 byte[] encryptedBlock = new byte[16];  
 for (int i = 0; i < 16; i++) {  
 encryptedBlock[i] = state[i % 4][i / 4];  
 }  
  
 return encryptedBlock;  
 }  
  
 // CFB encryption  
 public static byte[] encryptCFB(byte[] plaintext, byte[] key, byte[] iv) {  
 byte[] ciphertext = new byte[plaintext.length];  
 byte[] feedback = Arrays.*copyOf*(iv, iv.length); // IV ban đầu dùng làm feedback  
  
 for (int i = 0; i < plaintext.length; i += *BLOCK\_SIZE*) {  
 byte[] encryptedFeedback = *aesEncryptBlock*(feedback, key); // Mã hóa feedback  
 int blockSize = Math.*min*(*BLOCK\_SIZE*, plaintext.length - i); // Đảm bảo kích thước hợp lệ  
  
 // XOR từng byte của plaintext với encryptedFeedback  
 for (int j = 0; j < blockSize; j++) {  
 ciphertext[i + j] = (byte) (plaintext[i + j] ^ encryptedFeedback[j]);  
 }  
 // Cập nhật feedback với ciphertext vừa mã hóa  
 feedback = Arrays.*copyOfRange*(ciphertext, i, i + blockSize);  
 }  
  
 return ciphertext;  
 }  
  
 // Hàm giải mã CFB (tương tự như mã hóa, nhưng ngược lại)  
 // CFB decryption  
 public static byte[] decryptCFB(byte[] ciphertext, byte[] key, byte[] iv) {  
 byte[] plaintext = new byte[ciphertext.length];  
 byte[] feedback = Arrays.*copyOf*(iv, iv.length); // IV ban đầu dùng làm feedback  
  
 for (int i = 0; i < ciphertext.length; i += *BLOCK\_SIZE*) {  
 byte[] encryptedFeedback = *aesEncryptBlock*(feedback, key); // Mã hóa feedback  
 int blockSize = Math.*min*(*BLOCK\_SIZE*, ciphertext.length - i);  
  
 // XOR ciphertext với encryptedFeedback để khôi phục plaintext  
 for (int j = 0; j < blockSize; j++) {  
 plaintext[i + j] = (byte) (ciphertext[i + j] ^ encryptedFeedback[j]);  
 }  
  
 // Cập nhật feedback với ciphertext hiện tại  
 feedback = Arrays.*copyOfRange*(ciphertext, i, i + blockSize);  
 }  
  
 return plaintext;  
 }  
  
 // Substitute bytes using a simplified S-box  
 public static byte[][] subBytes(byte[][] state) {  
 for (int i = 0; i < 4; i++) {  
 for (int j = 0; j < 4; j++) {  
 state[i][j] = (byte) ((state[i][j] & 0x0F) ^ 0x63); // Simplified S-box logic  
 }  
 }  
 return state;  
 }  
  
 // Shift rows step  
 public static byte[][] shiftRows(byte[][] state) {  
 for (int i = 1; i < 4; i++) {  
 byte[] row = Arrays.*copyOf*(state[i], 4);  
 for (int j = 0; j < 4; j++) {  
 state[i][j] = row[(j + i) % 4];  
 }  
 }  
 return state;  
 }  
  
 // Mix columns step (fully implemented)  
 public static byte[][] mixColumns(byte[][] state) {  
 for (int c = 0; c < 4; c++) {  
 byte[] col = new byte[4];  
 for (int i = 0; i < 4; i++) {  
 col[i] = state[i][c];  
 }  
 state[0][c] = (byte) (*galoisMult*(col[0], 2) ^ *galoisMult*(col[1], 3) ^ col[2] ^ col[3]);  
 state[1][c] = (byte) (col[0] ^ *galoisMult*(col[1], 2) ^ *galoisMult*(col[2], 3) ^ col[3]);  
 state[2][c] = (byte) (col[0] ^ col[1] ^ *galoisMult*(col[2], 2) ^ *galoisMult*(col[3], 3));  
 state[3][c] = (byte) (*galoisMult*(col[0], 3) ^ col[1] ^ col[2] ^ *galoisMult*(col[3], 2));  
 }  
 return state;  
 }  
  
 // Galois field multiplication  
 private static byte galoisMult(byte a, int b) {  
 byte p = 0;  
 byte hiBitSet;  
 for (int i = 0; i < 8; i++) {  
 if ((b & 1) != 0) {  
 p ^= a;  
 }  
 hiBitSet = (byte) (a & 0x80);  
 a <<= 1;  
 if (hiBitSet != 0) {  
 a ^= 0x1b; // x^8 + x^4 + x^3 + x + 1  
 }  
 b >>= 1;  
 }  
 return p;  
 }  
  
 // Add round key step  
 public static byte[][] addRoundKey(byte[][] state, byte[] key) {  
 for (int i = 0; i < 4; i++) {  
 for (int j = 0; j < 4; j++) {  
 state[i][j] ^= key[i \* 4 + j];  
 }  
 }  
 return state;  
 }  
  
 // Generate round key (placeholder implementation)  
 public static byte[] generateRoundKey(int round, byte[] key) {  
 byte[] roundKey = Arrays.*copyOf*(key, *BLOCK\_SIZE*);  
 for (int i = 0; i < *BLOCK\_SIZE*; i++) {  
 roundKey[i] ^= round;  
 }  
 return roundKey;  
 }  
}