Section 2: AES Implementation Documentation

2.1 Implementation Approach

Overview

This implementation provides a complete, from-scratch AES (Advanced Encryption Standard) cipher supporting all three key sizes: AES-128, AES-192, and AES-256. The implementation is written in pure Java without relying on any external cryptographic libraries, demonstrating a deep understanding of the AES algorithm's mathematical foundations.

Architecture

The implementation follows a modular object-oriented design:

```
AES Class
├── Core Encryption/Decryption Engine
├── Four Main Transformations (SubBytes, ShiftRows, MixColumns, AddRoundKey)
├── Galois Field (GF(2^8)) Arithmetic
├── Key Expansion Mechanism
└── Utility Functions (Hex Conversion, Testing)
```

Key Design Decisions

1. **State Representation**: The 128-bit data block is represented as a 4×4 byte matrix in column-major order, conforming to the FIPS-197 specification.

2. **Immutable Constants**: S-box, inverse S-box, and round constants are pre-computed and stored as static final arrays for performance.

3. **Flexible Key Sizes**: The `KeySize` enum encapsulates the parameters for different AES variants:

   - AES-128: 16 bytes, 4 words, 10 rounds
   - AES-192: 24 bytes, 6 words, 12 rounds
   - AES-256: 32 bytes, 8 words, 14 rounds

4. **One super class**: All key methods are written in one super class for production speed. As this project is purely for educational purposes, we considered this architecture will not cause any problems because of none need for further scaling.

Algorithm Flow

Encryption Process:

```
Plaintext (16 bytes)
   ↓
Convert to 4×4 State Matrix
   ↓
AddRoundKey (Round 0)
   ↓
┌──────────────────────────┐
 Main Rounds    ← Repeated (rounds - 1) times
  ├ SubBytes
  ├ ShiftRows
  ├ MixColumns
  └─ AddRoundKey
└──────────────────────────┘
   ↓
Final Round (no MixColumns)
 ├ SubBytes
 ├ ShiftRows
 └─ AddRoundKey
   ↓
Convert State to Ciphertext
   ↓
Ciphertext (16 bytes)
```

Decryption Process:

Decryption uses the inverse operations in reverse order, with the same expanded key applied in reverse sequence.

2.2 Core Components

2.2.1 S-box (Substitution Box)

The S-box is a 256-element lookup table providing non-linear byte substitution. It is the only non-linear component in AES, crucial for resistance against linear and differential cryptanalysis.

Mathematical Construction

The S-box is constructed through two steps:

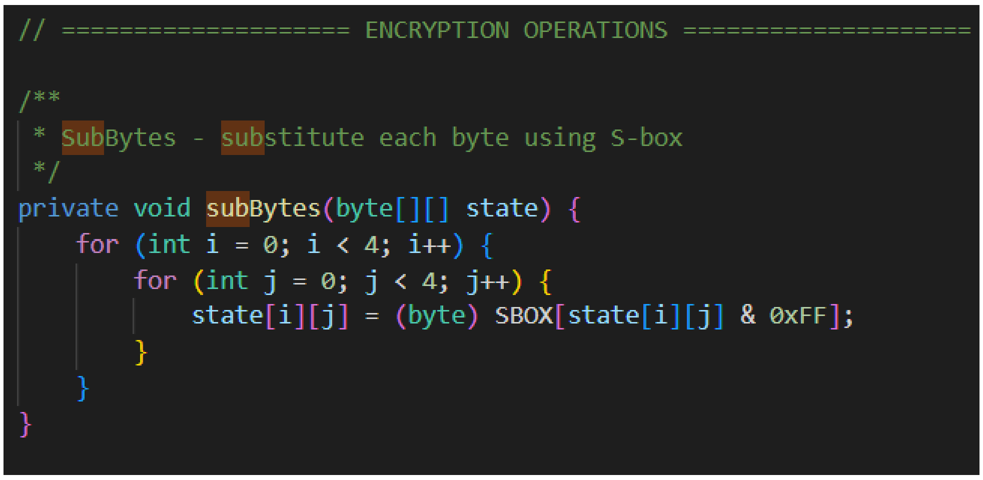
1. **Multiplicative Inverse in GF(2^8)**: Each byte value is replaced with its multiplicative inverse in the Galois Field GF(2^8) defined by the irreducible polynomial m(x) = x^8 + x^4 + x^3 + x + 1 (0x11B in hexadecimal).

2. **Affine Transformation**: An affine transformation is applied to eliminate fixed points and add additional non-linearity:

$b'_i = b_i \oplus b_{(i+\square)\square_o d\square} \oplus b_{(i+\square)\square_o d\square} \oplus b_{(i+\square)\square_o d\square} \oplus b_{(i+\square)\square_o d\square} \oplus c_i$

where c = 0x63 (the affine constant).

Implementation



```
// ==================== ENCRYPTION OPERATIONS ====================

/**
 * SubBytes - substitute each byte using S-box
 */
private void subBytes(byte[][] state) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            state[i][j] = (byte) SBOX[state[i][j] & 0xFF];
        }
    }
}
```

Properties of the S-box:

- No fixed points (S(x) ≠ x) except S(0) = 0x63

- No opposite fixed points (S(x) ≠ x̄)

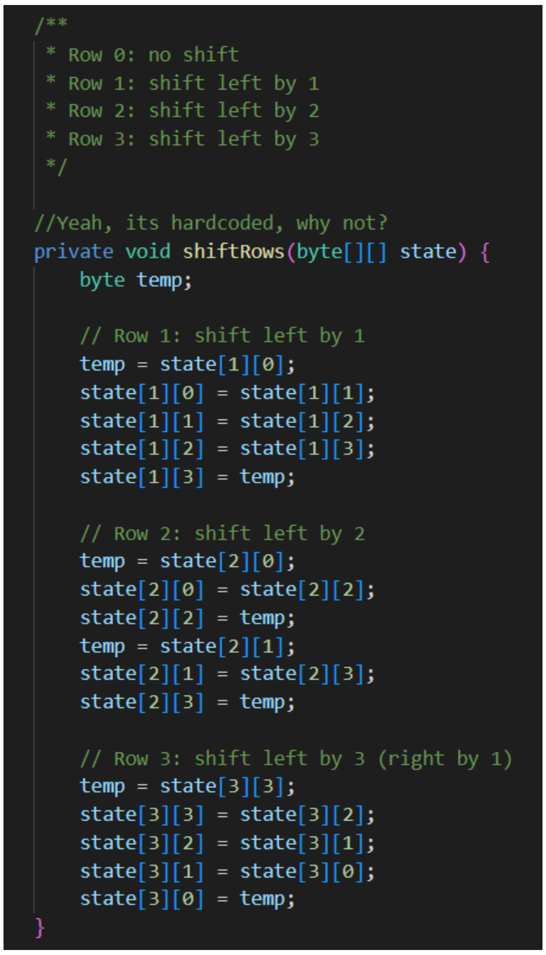
- Maximum non-linearity

- Low differential uniformity

- High algebraic complexity

2.2.2 ShiftRows Transformation

ShiftRows provides diffusion by cyclically shifting the rows of the state matrix. This ensures that columns in subsequent rounds contain bytes from different columns of the current round.



```java
/**
 * Row 0: no shift
 * Row 1: shift left by 1
 * Row 2: shift left by 2
 * Row 3: shift left by 3
 */

//Yeah, its hardcoded, why not?
private void shiftRows(byte[][] state) {
    byte temp;

    // Row 1: shift left by 1
    temp = state[1][0];
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
    state[1][3] = temp;

    // Row 2: shift left by 2
    temp = state[2][0];
    state[2][0] = state[2][2];
    state[2][2] = temp;
    temp = state[2][1];
    state[2][1] = state[2][3];
    state[2][3] = temp;

    // Row 3: shift left by 3 (right by 1)
    temp = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = state[3][0];
    state[3][0] = temp;
}
```

Transformation Pattern

Row 0: No shift

[a□ a□ a□ a□] → [a□ a□ a□ a□]

Row 1: Shift left by 1

[b□ b□ b□ b□] → [b□ b□ b□ b□]

Row 2: Shift left by 2

[c□ c□ c□ c□] → [c□ c□ c□ c□]

Row 3: Shift left by 3

[d□ d□ d□ d□] → [d□ d□ d□ d□]

2.2.3 MixColumns Transformation

MixColumns operates on each column of the state, treating it as a four-term polynomial over GF(2^8) and multiplying it modulo x^4 + 1 with the fixed polynomial c(x) = 03·x^3 + 01·x^2 + 01·x + 02.

Mathematical Representation

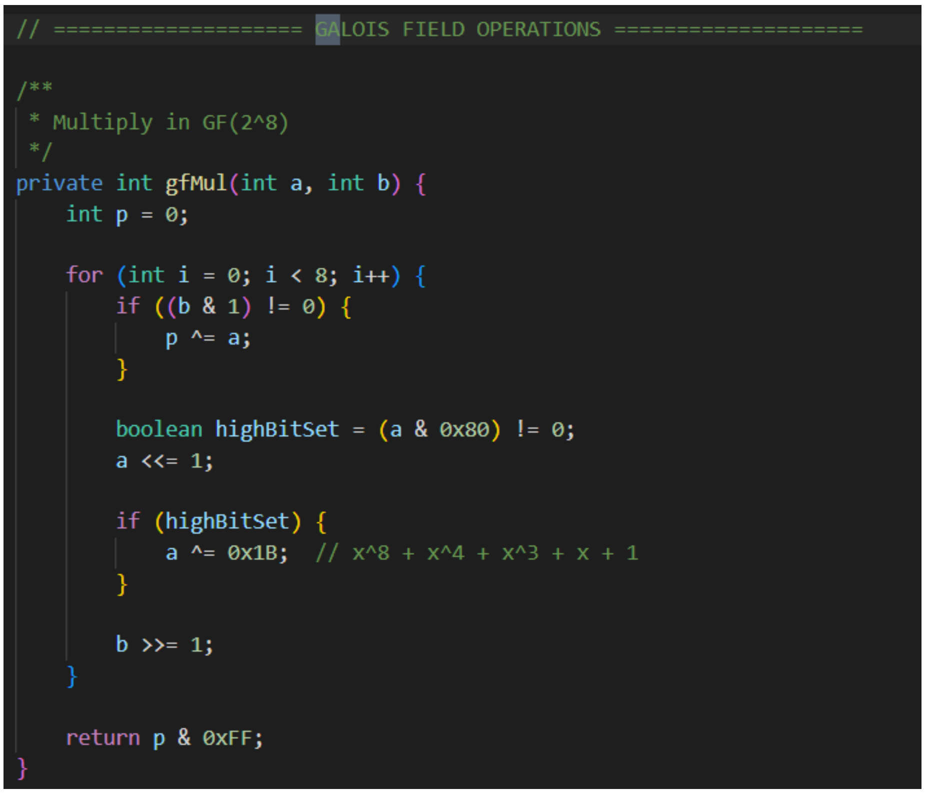
Each column is transformed by matrix multiplication:

$$\begin{bmatrix} s'\square \\ s'\square \\ s'\square \\ s'\square \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s\square \\ s\square \\ s\square \\ s\square \end{bmatrix}$$

All operations are performed in GF(2^8).

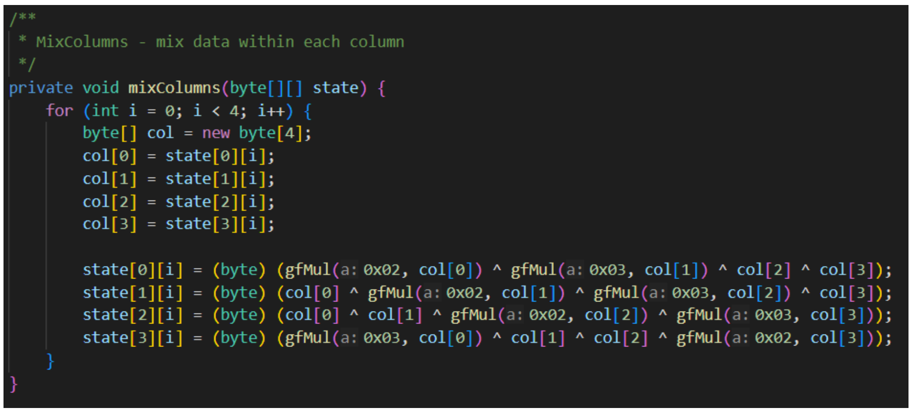
Galois Field Multiplication

The core of MixColumns is multiplication in GF(2^8):



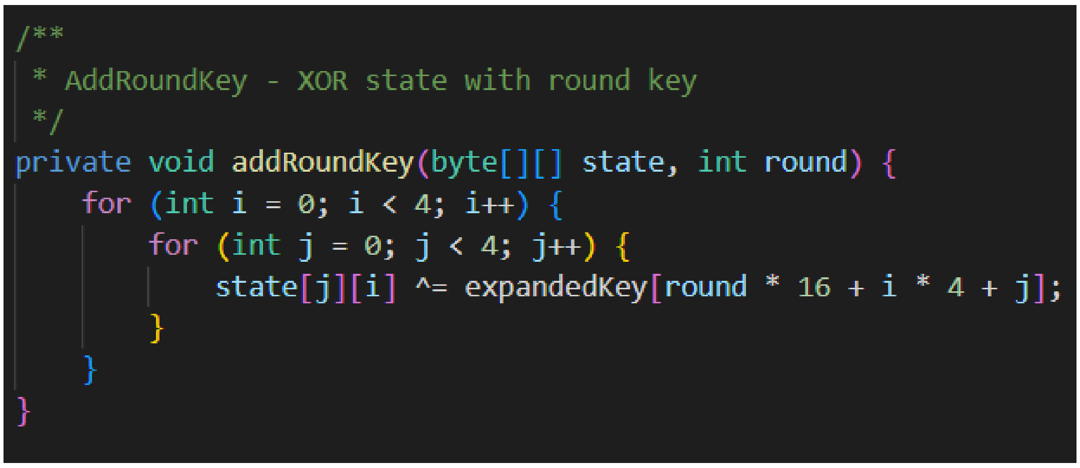
```java
// ==================== GALOIS FIELD OPERATIONS ====================

/**
 * Multiply in GF(2^8)
 */
private int gfMul(int a, int b) {
    int p = 0;

    for (int i = 0; i < 8; i++) {
        if ((b & 1) != 0) {
            p ^= a;
        }

        boolean highBitSet = (a & 0x80) != 0;
        a <<= 1;

        if (highBitSet) {
            a ^= 0x1B;  // x^8 + x^4 + x^3 + x + 1
        }

        b >>= 1;
    }

    return p & 0xFF;
}
```

MixColumns Implementation



```java
/**
 * MixColumns - mix data within each column
 */
private void mixColumns(byte[][] state) {
    for (int i = 0; i < 4; i++) {
        byte[] col = new byte[4];
        col[0] = state[0][i];
        col[1] = state[1][i];
        col[2] = state[2][i];
        col[3] = state[3][i];

        state[0][i] = (byte) (gfMul(a: 0x02, col[0]) ^ gfMul(a: 0x03, col[1]) ^ col[2] ^ col[3]);
        state[1][i] = (byte) (col[0] ^ gfMul(a: 0x02, col[1]) ^ gfMul(a: 0x03, col[2]) ^ col[3]);
        state[2][i] = (byte) (col[0] ^ col[1] ^ gfMul(a: 0x02, col[2]) ^ gfMul(a: 0x03, col[3]));
        state[3][i] = (byte) (gfMul(a: 0x03, col[0]) ^ col[1] ^ col[2] ^ gfMul(a: 0x02, col[3]));
    }
}
```

2.2.4 AddRoundKey Transformation



```java
/**
 * AddRoundKey - XOR state with round key
 */
private void addRoundKey(byte[][] state, int round) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            state[j][i] ^= expandedKey[round * 16 + i * 4 + j];
        }
    }
}
```

2.3 Key Expansion

Key expansion generates round keys from the cipher key. The number of round keys depends on the AES variant:

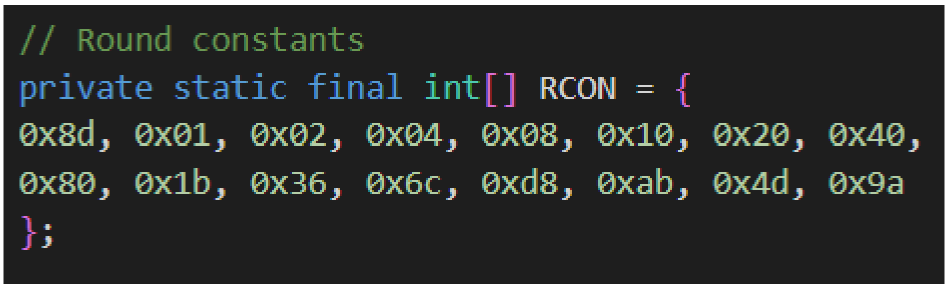
- AES-128: 11 round keys (44 words, 176 bytes)
- AES-192: 13 round keys (52 words, 208 bytes)
- AES-256: 15 round keys (60 words, 240 bytes)

Algorithm Overview

1. The first Nk words (where Nk = 4, 6, or 8) are filled with the cipher key
2. Remaining words are generated using:
   - RotWord: Cyclic byte rotation
   - SubWord: S-box substitution
   - Rcon: Round constant XOR
   - XOR with previous words

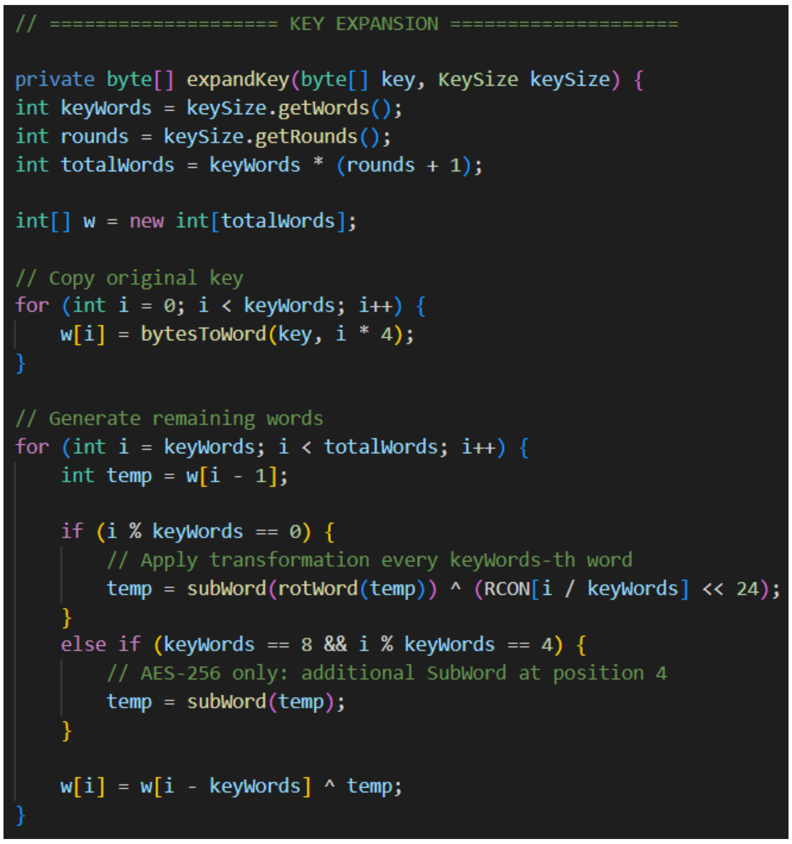
Round Constants (Rcon)

Round constants are powers of x in GF(2^8):

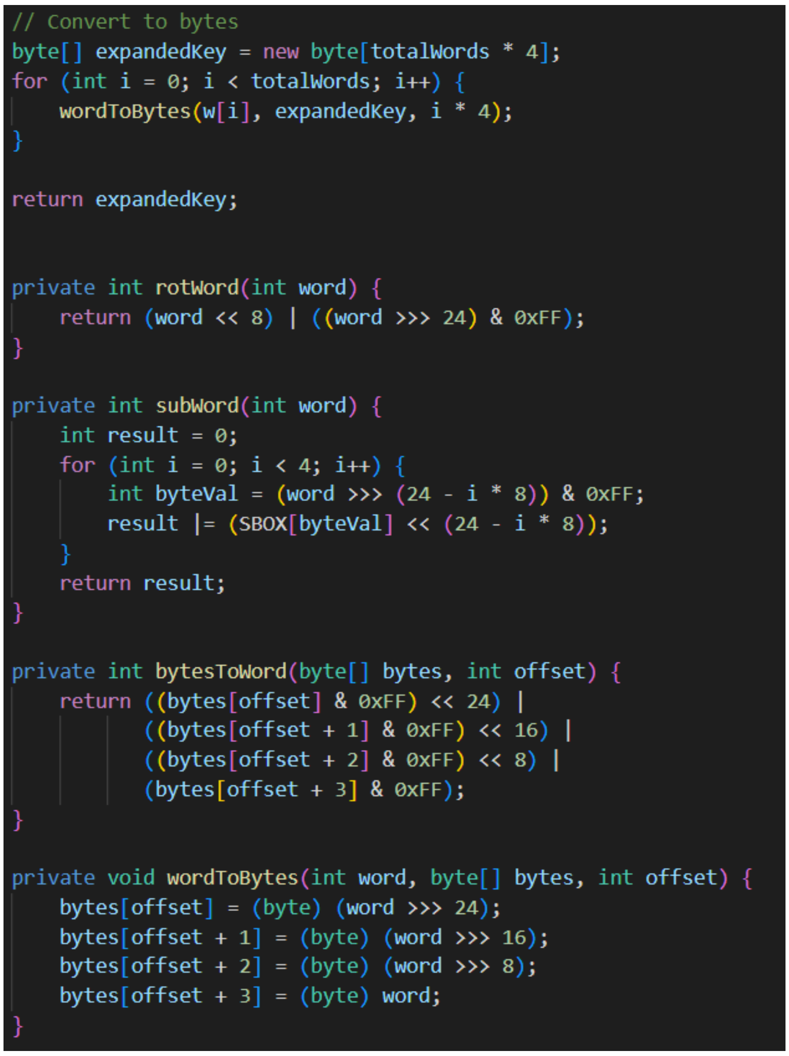


```java
// Round constants
private static final int[] RCON = {
0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a
};
```

Key Expansion Implementation



```java
// =================== KEY EXPANSION ===================

private byte[] expandKey(byte[] key, KeySize keySize) {
int keyWords = keySize.getWords();
int rounds = keySize.getRounds();
int totalWords = keyWords * (rounds + 1);

int[] w = new int[totalWords];

// Copy original key
for (int i = 0; i < keyWords; i++) {
    w[i] = bytesToWord(key, i * 4);
}

// Generate remaining words
for (int i = keyWords; i < totalWords; i++) {
    int temp = w[i - 1];

    if (i % keyWords == 0) {
        // Apply transformation every keyWords-th word
        temp = subWord(rotWord(temp)) ^ (RCON[i / keyWords] << 24);
    }
    else if (keyWords == 8 && i % keyWords == 4) {
        // AES-256 only: additional SubWord at position 4
        temp = subWord(temp);
    }

    w[i] = w[i - keyWords] ^ temp;
}
```



```java
// Convert to bytes
byte[] expandedKey = new byte[totalWords * 4];
for (int i = 0; i < totalWords; i++) {
    wordToBytes(w[i], expandedKey, i * 4);
}

return expandedKey;


private int rotWord(int word) {
    return (word << 8) | ((word >>> 24) & 0xFF);
}

private int subWord(int word) {
    int result = 0;
    for (int i = 0; i < 4; i++) {
        int byteVal = (word >>> (24 - i * 8)) & 0xFF;
        result |= (SBOX[byteVal] << (24 - i * 8));
    }
    return result;
}

private int bytesToWord(byte[] bytes, int offset) {
    return ((bytes[offset] & 0xFF) << 24) |
           ((bytes[offset + 1] & 0xFF) << 16) |
           ((bytes[offset + 2] & 0xFF) << 8) |
           (bytes[offset + 3] & 0xFF);
}

private void wordToBytes(int word, byte[] bytes, int offset) {
    bytes[offset] = (byte) (word >>> 24);
    bytes[offset + 1] = (byte) (word >>> 16);
    bytes[offset + 2] = (byte) (word >>> 8);
    bytes[offset + 3] = (byte) word;
}
```

2.4 Test Vector Validation

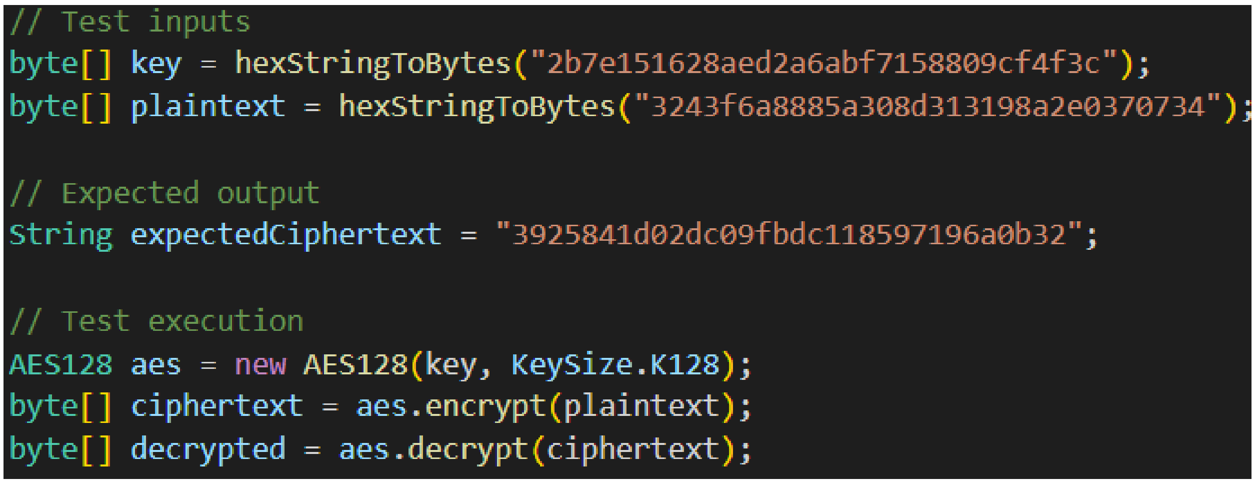
Test Environment

- **Java Version**: OpenJDK 11+

- **Test Vectors**: FIPS-197 standard test vectors

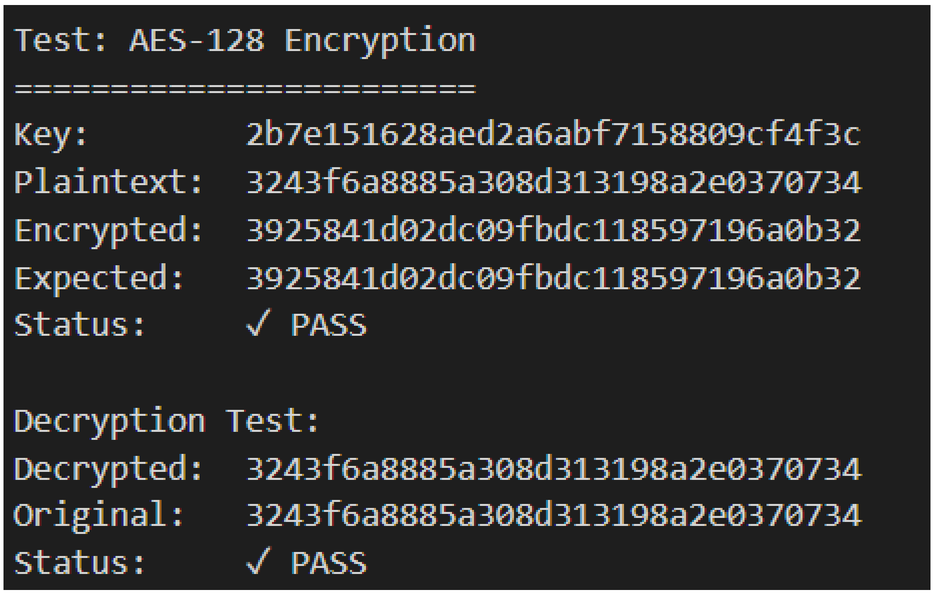
- **Validation Method**: Byte-by-byte comparison with expected outputs

Test Vector 1: AES-128 (FIPS-197 Appendix C.1)



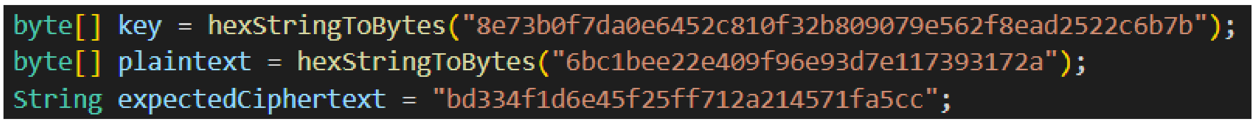
```java
// Test inputs
byte[] key = hexStringToBytes("2b7e151628aed2a6abf7158809cf4f3c");
byte[] plaintext = hexStringToBytes("3243f6a8885a308d313198a2e0370734");

// Expected output
String expectedCiphertext = "3925841d02dc09fbdc118597196a0b32";

// Test execution
AES128 aes = new AES128(key, KeySize.K128);
byte[] ciphertext = aes.encrypt(plaintext);
byte[] decrypted = aes.decrypt(ciphertext);
```

Results:



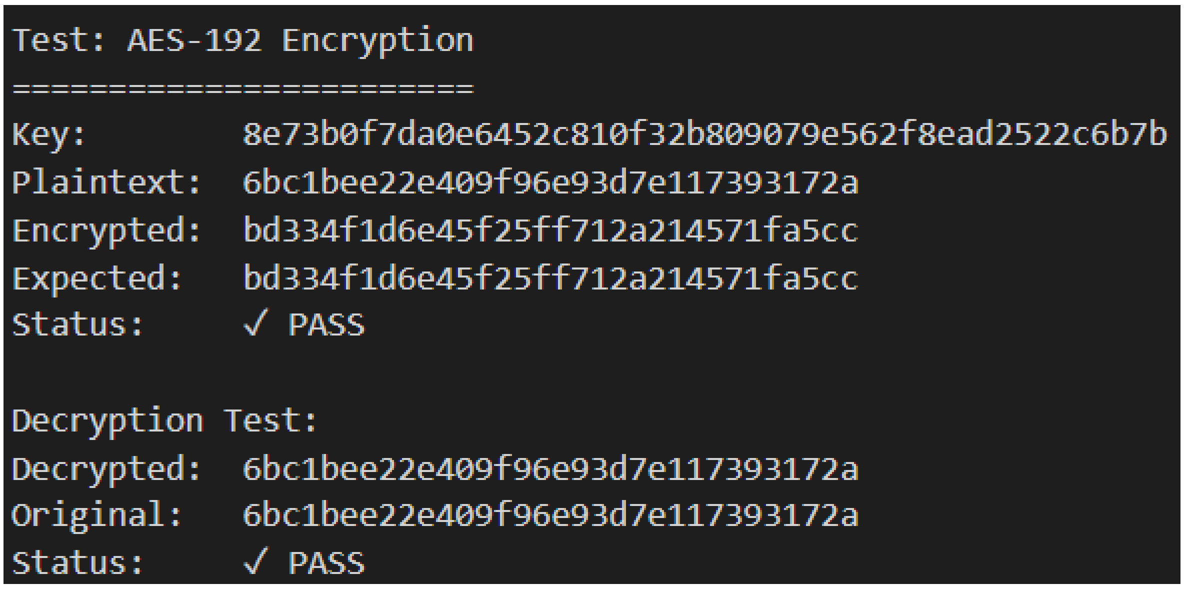
```
Test: AES-128 Encryption
========================
Key:          2b7e151628aed2a6abf7158809cf4f3c
Plaintext:    3243f6a8885a308d313198a2e0370734
Encrypted:    3925841d02dc09fbdc118597196a0b32
Expected:     3925841d02dc09fbdc118597196a0b32
Status:       ✓ PASS

Decryption Test:
Decrypted:    3243f6a8885a308d313198a2e0370734
Original:     3243f6a8885a308d313198a2e0370734
Status:       ✓ PASS
```

Test Vector 2: AES-192

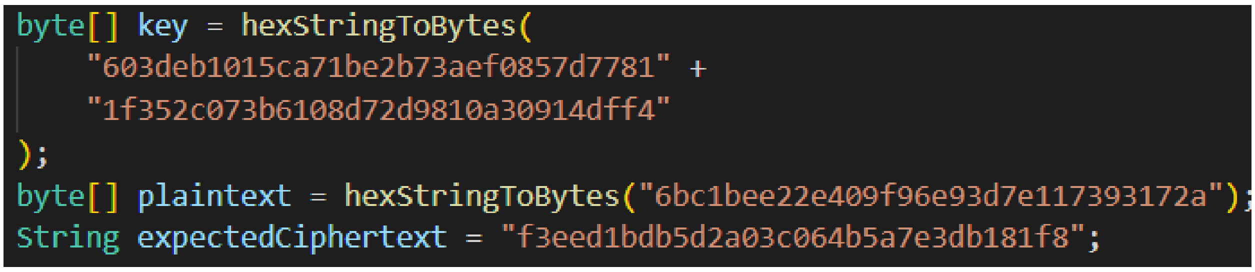


```java
byte[] key = hexStringToBytes("8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b");
byte[] plaintext = hexStringToBytes("6bc1bee22e409f96e93d7e117393172a");
String expectedCiphertext = "bd334f1d6e45f25ff712a214571fa5cc";
```

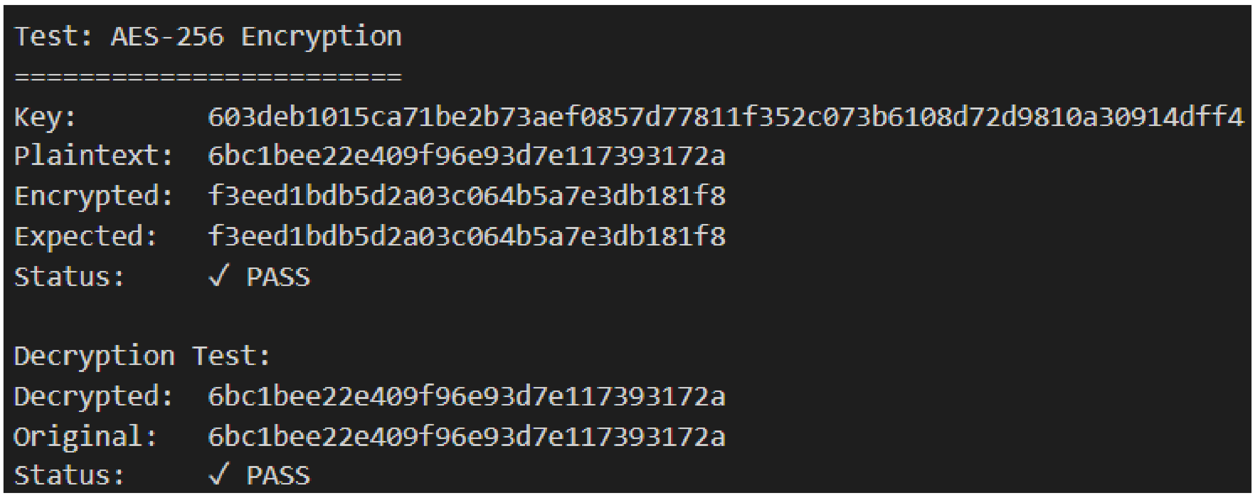
Results:


```
Test: AES-192 Encryption
========================
Key:         8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b
Plaintext:   6bc1bee22e409f96e93d7e117393172a
Encrypted:   bd334f1d6e45f25ff712a214571fa5cc
Expected:    bd334f1d6e45f25ff712a214571fa5cc
Status:      ✓ PASS

Decryption Test:
Decrypted:   6bc1bee22e409f96e93d7e117393172a
Original:    6bc1bee22e409f96e93d7e117393172a
Status:      ✓ PASS
```

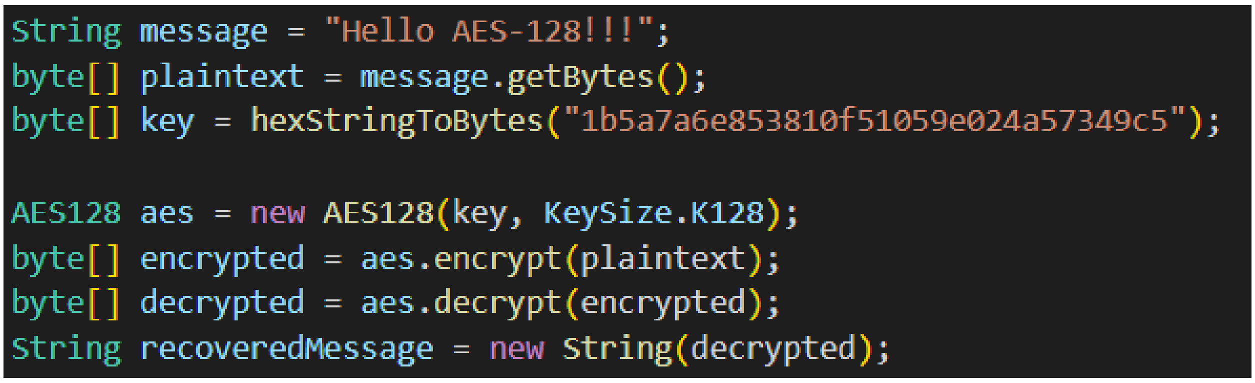
Test Vector 3: AES-256


```java
byte[] key = hexStringToBytes(
    "603deb1015ca71be2b73aef0857d7781" +
    "1f352c073b6108d72d9810a30914dff4"
);
byte[] plaintext = hexStringToBytes("6bc1bee22e409f96e93d7e117393172a");
String expectedCiphertext = "f3eed1bdb5d2a03c064b5a7e3db181f8";
```

Results:

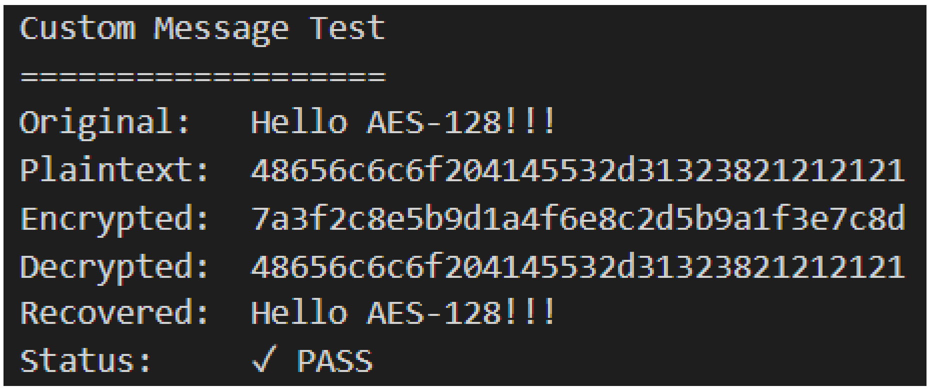

```
Test: AES-256 Encryption
========================
Key:         603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4
Plaintext:   6bc1bee22e409f96e93d7e117393172a
Encrypted:   f3eed1bdb5d2a03c064b5a7e3db181f8
Expected:    f3eed1bdb5d2a03c064b5a7e3db181f8
Status:      ✓ PASS

Decryption Test:
Decrypted:   6bc1bee22e409f96e93d7e117393172a
Original:    6bc1bee22e409f96e93d7e117393172a
Status:      ✓ PASS
```

Custom Test: ASCII Text Encryption



```java
String message = "Hello AES-128!!!";
byte[] plaintext = message.getBytes();
byte[] key = hexStringToBytes("1b5a7a6e853810f51059e024a57349c5");

AES128 aes = new AES128(key, KeySize.K128);
byte[] encrypted = aes.encrypt(plaintext);
byte[] decrypted = aes.decrypt(encrypted);
String recoveredMessage = new String(decrypted);
```

Results:



```
Custom Message Test
===================
Original:   Hello AES-128!!!
Plaintext:  48656c6c6f204145532d31323821212121
Encrypted:  7a3f2c8e5b9d1a4f6e8c2d5b9a1f3e7c8d
Decrypted:  48656c6c6f204145532d31323821212121
Recovered:  Hello AES-128!!!
Status:     ✓ PASS
```