Лабораторна робота №2

Симетричне шифрування

Виконав: Безпалий Марко Леонідович, КН-М922в

Завдання:

Реалізувати алгоритм симетричного шифрування AES (будь-якої версії - 128 або 256).

Довести коректність роботи реалізованого алгоритму шляхом порівняння результатів з існуючими реалізаціями (напр. сайтом-утилітою https://cryptii.com).

Опис роботи:

У цій роботі був реалізований алгоритм для роботи з 128-бітним ключем, шифрувальним режимом ECB та доповненням PKCS5. Нам знадобиться декілька константних таблиць:

S-BOX Таблиця

```
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
```

Константи для раундів алгоритму розширення ключей

```
 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xed, 0xc5
```

Фіксована матриця для перемішування стовпців

```
2 3 1 1
1 2 3 1
```

```
1 1 2 3
3 1 1 2
```

AES/ECB/PKCS5 (128)

Спочатку йде перевірка, що ключ містить 16 байтів, тобто 128 бітів.

```
if (key.length() != BYTES_BLOCK_SIZE) {
    throw new IllegalArgumentException("Length of secret key should be 16 for 128 bits key size");
}
```

Далі йде поділ тексту, що потрібно зашифрувати на блоки по 16 байтів

```
List<Byte> byteList = ByteUtils.toByteList(text.getBytes());
List<List<Byte>> blocks = ListUtils.partition(byteList, 128 / Byte.SIZE);
```

Останній блок може містити менше ніж 16 байтів, тому нам треба його доповнити за алгоритмом PKCS5. Цей алгоритм передбачає додавання байтів до останнього блоку, які є значенням довжини байтів, яких не вистачає. Тобто ящо в нас не вистачає три байти, то ми додаємо `0x03` до останнього блоку, поки він не буде розміром в 16 байтів.

```
int size = blocks.size();
List<Byte> lastBlock = blocks.get(size - 1);

int paddingData = BYTES_BLOCK_SIZE - lastBlock.size();

while (lastBlock.size() != BYTES_BLOCK_SIZE) {
    lastBlock.add((byte) paddingData);
}
```

Далі йде основний алгоритм, що виглядає наступним чином:

```
roundKeys = getRoundKeys(key)

for (block in blocks) {
    //first round
    addRoundKey(roundKeys[0], block);
    //intermediate rounds
    for (10 rounds) {
        subBytes(block);
        shiftRows(block);
        mixColumns(block);
        addRoundKey(roundKeys[round_number - 1], block);
    }
    //last round
    subBytes(block);
    shiftRows(block);
    addRoundKey(roundKeys[last], block);
}
    ciphertext = collectBytes(blocks);
    return ciphertext;
```

```
return cipherText;
```

Опишемо алгоритм кожного з методів

getRoundKeys

Метод `getRoundKeys` бере ключ, що передає користувач та створює на його основі додаткові ключі, що потрібні на кожному раунді. Алгоритм цього методу складається з наступних дій:

1. Ключ розбивається на частини по 4 байти (слова `w[0..3]`).
2. Для останнього слова виконуються наступні дії `g(w[3])` :
   - круговий зсув вліво
   - заміна всіх байтів на байти з таблиці `S-BOX`
   - Додавання (XOR) константи поточного раунду
3. Вирахувати нові байти для наступного ключа:
   - `w[4] = w[0] XOR g(w[3])`
   - `w[5] = w[4] XOR w[1]`
   - `w[6] = w[5] XOR w[2]`
   - `w[7] = w[6] XOR w[3]`
4. Новий ключ використовується для створення нових. У нашому випадку нам потрібно 11 ключей.

addRoundKey

Метод `addRoundKey` додає (XOR) до кожного значення матриці стану значення поточного ключа.

```
for (int i = 0; i < matrixColumns.size(); i++) {
    int added = matrixColumns.get(i) ^ roundKey.get(i);
    result.add((byte) added);
}
```

subBytes

Метод `subBytes` замінює кожен байт матриці на відповідний із таблиці `S-BOX` . В даній роботі ця таблиця реалізовна як одномірний масив, де значення вхідного байта є індексом вихідного. Але перед тим, як дізнатися індекс, потрібно перевести тип даних Byte в Integer, тому що нам потрібно, щоб наш перший біт був не знаковим, а додавав значення. Тобто нам потрібен `unsigned byte` .

```
List<Integer> intValues = stateMatrix.stream()
    .map(bValue -> new byte[]{0, 0, 0, bValue})
    .map(ByteBuffer::wrap)
    .map(ByteBuffer::getInt)
    .collect(Collectors.toList());

    return intValues.stream()
        .map(S_BOX_TABLE::get)
        .map(Integer::byteValue)
        .collect(Collectors.toList());
```

shiftRows

Метод `shiftRows` робить круговий зсув рядків матриці стану на число, що відповідає номеру рядку (0-3). В нас завжди всього 4 рядки, тому перший рядок не зсувається, другий зсувається на 1 байт, третій на 2 байти, четвертий на 3 байти.

В цій реалізації всі методи, окрім `shiftRows` виконують операції над стовпцями, тому для цього методу потрібно змінити логічне направлення, тобто оперувати над рядками, замість стовпців.

```
private static List<Byte> changeLogicalDirection(List<Byte> values) {
    List<Byte> viceVersaDirection = new ArrayList<>(values.size() / 4);
    List<List<Byte>> forwardDirection = ListUtils.partition(values, MATRIX_ROW_SIZE);

    for (int i = 0; i < MATRIX_ROW_SIZE; i++) {
        for (int j = 0; j < MATRIX_ROW_SIZE; j++) {
            Byte value = forwardDirection.get(j).get(i);
            viceVersaDirection.add(value);
        }

    }

    return viceVersaDirection;
}
```

mixColumns

Метод `mixColumns` перемножує матрицю стану на фіксовану матрицю, що містить тільки значення 1, 2 та 3. Замість звичайного складання потрібно використати XOR, а перемноження виконується з використанням полей Гауса у вигляді $GF(2^8)$.

В коді це виглядає наступним чином:

Множення на 2:

```
private static byte gMul2(byte value) {
    int highBit = value & 0x80;
    int result = value << 1;

    if (highBit == 0x80) {
        result = result ^ 0x1b;
    }

    return (byte) result;
}
```

Множення на 3:

```
private static byte gMul3(byte value) {
    return (byte) (value ^ gMul2(value));
}
```

Завершення алгоритму

У кінці алгоритму всі блоки з'єднуються й на виході ми отримуємо зашифровану послідовність.

Тест пройдено успішно