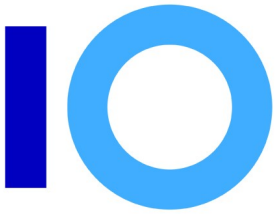


"Ss. Cyril and Methodius" University in Skopje
FACULTY OF COMPUTER
SCIENCE AND ENGINEERING

Лабораториска вежба бр. 2 по предметот

"Криптографија"

Trivium cipher implementation

Изработил:

Мартин Костадинов

Број на индекс:

161159

ВОВЕД

Во оваа документација ќе биде прикажан начинот на кој имплементиран Trivium алгоритмот за шифрирање и дешифрирање на пораки. Trivium е проточен алгоритам за криптирање кој генерира стрим од битови и овој стрим подоцна се користи како клуч за шифрирање и дешифрирање на дадена порака. Генерирањето на стримот од битови потребен за шифрирање и дешифрирање е тесно поврзан и директно зависи од клучот и иницијалниот вектор. Овој алгоритам во својата имплементација користи 80 битен клуч, 80 битен иницијален вектор и 288 битен internal state. Trivium бил создаден како пример за тоа колку еден проточен шифрувач може да биде олеснет без да се става на ризик безбедноста, брзината или флексибилноста. Програмскиот јазик избран за оваа имплементација, во мојот случај, е Java.

1. Key and IV setup

Делот со key and IV setup е од голема значајност бидејќи преку оваа метода се иницијализира почетната состојба (initial state) која пак подоцна се користи за генерирање на стрим од битови потребни за шифрирање и дешифрирање. Најнапред, првите 80 бита од клучот се сместуваат на првите 80 места во почетната состојба. Потоа од 93 бит па до 93+80 бит во иницијалната состојба се сместуваат вредностите на иницијалниот вектор и последните три бита од почетната состојба (288, 287, 286) се сетираат на 1. Сите останати битови кои што не ги спомнавме се сетираат на 0.

```
String[] keyArray = key.split( regex: "");
String[] vectorArray = IV.split( regex: "");

IntStream.range(0, keyArray.length).forEach(i -> initial_State[i] = Integer.parseInt(keyArray[i]));
IntStream.range(0, vectorArray.length).forEach(i -> initial_State[i+93] = Integer.parseInt(vectorArray[i]));

initial_State[285] = 1;
initial_State[286] = 1;
initial_State[287] = 1;
```

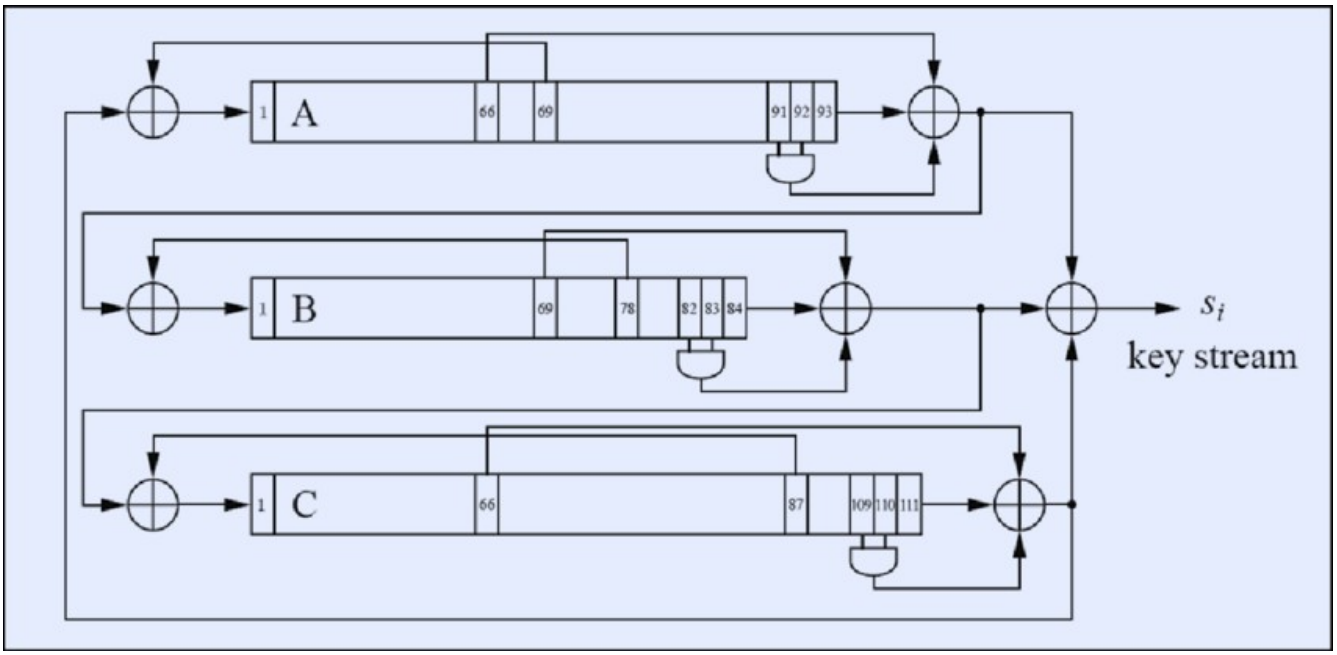
Во случајот не ги полниме нулите после 80 бит на клучот па се до 93 бит и тие после иницијалниот вектор бидејќи низата (initial_State) има зададена фиксна должина од 288 бита па тие сами се сетирани на 0, по default. Откако ќе ја наполниме почетната состојба со вредностите на клучот и иницијалниот вектор, потребно е да извршиме едно “мешање” со што ќе ја доведеме иницијалната состојба до посакуваните вредности. Потребно е да се извршат логички операции на одредени битови во иницијалната состојба т.ш добиваме три нови бита, ја ротираме низата циклично во десно за едно место и ги заменуваме овие битови на соодветно дефинирани места во иницијалната состојба. Овој процес се извршува 4*288 пати или вкупно 1152 пати.

```
for(int i = 0; i < 4*288; i ++) {

    int t1 = initial_State[65] ^ initial_State[90] & initial_State[91] ^ initial_State[92] ^ initial_State[170];
    int t2 = initial_State[161] ^ initial_State[174] & initial_State[175] ^ initial_State[176] ^ initial_State[263];
    int t3 = initial_State[242] ^ initial_State[285] & initial_State[286] ^ initial_State[287] ^ initial_State[68];

    rotateArray(initial_State);

    initial_State[0] = t3;
    initial_State[93] = t1;
    initial_State[177] = t2;
}
```

Со ова се завршува полнењето на иницијалната состојба и се преминува на генерирање на клучот од иницијалната состојба.

2. Keystream generation

Протокот од битови потребни за клучот, или keystream, користи слични методи како **key and IV setup** за да се генерираат. Во овој случај ја користиме иницијалната состојба (претходно пополнета со клучот, иницијалниот вектор, соодветно изротирана и со сите потребни замени).



Нека претпоставиме дека имаме три регистри (A, B и C) и секој е соодветно пополнет т.ш A се првите 93 бита од иницијалната состојба (првите 93 бита со клучот), B е вториот дел со иницијалниот вектор (наредните 84 бита) и третиот дел е оној што останува каде што последните три бита се 1. За да се генерира еден бит од keystreamot, потребно е да извршиме логичко XOR на 66 и 93 бит од првиот регистер, 69 и 84 бит од вториот и 66 и 111 од третиот. Потоа потребно е да извршиме XOR на трите добиени вредности од претходниот чекор. Со овој начин добивме еден бит од keystreamot а доколку сакаме да добиеме онолку битови колку што ни се потребни, истата постапка ја повторуваме онолку пати колку што е долга пораката (макс. до 2^{64}).

```
private int generate_Keystream(){
    int t1 = initial_State[65] ^ initial_State[92];
    int t2 = initial_State[161] ^ initial_State[176];
    int t3 = initial_State[242] ^ initial_State[287];

    int z = t1 ^ t2 ^ t3;

     t1 = initial_State[65] ^ initial_State[90] & initial_State[91] ^ initial_State[92] ^ initial_State[170];
     t2 = initial_State[161] ^ initial_State[174] & initial_State[175] ^ initial_State[176] ^ initial_State[263];
     t3 = initial_State[242] ^ initial_State[285] & initial_State[286] ^ initial_State[287] ^ initial_State[68];

    rotateArray(initial_State);

    initial_State[0] = t3;
    initial_State[93] = t1;
    initial_State[177] = t2;


    return z;
```

3. Encrypt

Делот со енкрипција е прилично едноставен. Потребно е да генерираме keystream како што беше опишано во претходниот параграф и потоа соодветно бит по бит да се изврши XOR на оригиналната порака со keystreamot.

Во случајот го користиме клучот **0F62B5085BAE0154A7FA** и иницијалниот вектор **288FF65DC42B92F960C7** за да генерираме keystream. Најнапред потребно е да ја наместиме иницијалната вредност и откако ќе се извршат сите 1152 ротирања, иницијалната состојба ги има следните вредности:
011101001011100000011100100000110011010000100001000011110000000101101000110110011111111111110101100110011110110110011000001100001011110110000101101011100100110001110101111111011101110101111101111011011001001001010101001010011000001010110010101101100011001100010100000011010010101101101100101 (вкупно 288 бита).

За да ја декриптираме пораката “Kriptografija” потребно е да изгенерираме keystream со големина на зборот “Kriptografija” претставен во бинарен формат. Во прилог се дадени оригиналната порака во бинарен формат и keystreamot.

```
ORIGINAL:
100101111100101101001111000011101001101111110011111110010110000111001101101001110101011000001
KEYSTREAM:
101011001100110101000101010001010011001001101010010111111111110010100000111001000100111110011
```

Сега потребно е да извршиме соодветно XOR на секој бит и да ја добиеме енкриптираната порака која изгледа вака:

```
ENCRYPTED:
001110110000011000001010010010111010100110011001101011010011101011011010100001100011001001O
```

Шифрираната порака го добива следниот облик: `□AA$]&3-□6Pc□`

Шифрираната порака ја менува својата форма во зависност од иницијалниот вектор и вредноста на клучот.

```
String[] encrypt(String[] originalList) {
    StringBuilder sb = new StringBuilder();
    int k = 0;
    String[] s = convertArrayToStringMethod(originalList).split( regex: "");
    keyStream(convertArrayToStringMethod(originalList).length());
    int[] encryptedMessage = new int[keystream.length];
    IntStream.range(0, keystream.length).forEach(i -> encryptedMessage[i] = Integer.parseInt(s[i]) ^ keystream[i]);


    for (String s1 : originalList) {
        for (int j = k; j < s1.length() + k; j++) {
            sb.append(encryptedMessage[j]);
        }
        sb.append(" ");
        k += s1.length();
    }

    return sb.toString().split( regex: " ");
}
```

4. Decrypt

Постапката за декрипција е идентична со постапката за енкрипција само што во случајот наместо оригиналната порака, ги користиме битовите од шифрираната порака и правиме XOR со keystreamot.

```
String[] decrypt(String[] encryptedList) {

    int k = 0;
    StringBuilder sb = new StringBuilder();
    String[] s = convertArrayToStringMethod(encryptedList).split( regex: "");
    int[] decryptMessage = new int[keystream.length];

    IntStream.range(0, keystream.length).forEach(i -> decryptMessage[i] = Integer.parseInt(s[i]) ^ keystream[i]);


    for (String s1 : encryptedList) {
        for (int j = k; j < s1.length() + k; j++) {
            sb.append(decryptMessage[j]);
        }
        sb.append(" ");
        k += s1.length();
    }

    return sb.toString().split( regex: " ");
}
```

Доколку го искористиме стрингот “**Trivium is a synchronous stream cipher designed to provide a flexible trade-off between speed and gate count in hardware, and reasonably efficient software implementation.”** вака би изгледале енкрипцијата и декрипцијата:

```
□□I □□□ 11  0y□□MEt□gl+<$C7t
□t
□□HL□;+
□□□1□□Fd□3q
□□to/□m7gp□□QJ8h□j6E7p;z□z0k @9□
2#(.q□□Rm4F/&O3□

Trivium is a synchronous stream cipher designed to provide a flexible trade-off between speed and gate count in hardware, and reasonably efficient software implementation.
```