IMPLEMENTATION AND FAULT ATTACK ON SALSA20 CIPHER

Project Report

Submitted by:

Brij Bhushan

Roll No. 1301016

Under the Guidance of

Dr. Sandip Karmakar

Department of Computer Science and Engineering

Indian Institute of Information Technology Guwahati

Guwahati 781001, India

April, 2016

CONTENTS

1. Objective of the project

The problem statement of the project is:

"**To implement Salsa20 cipher and to perform fault attack on it.**"

The purpose of the project is to understand the working of the Salsa20 cipher and also to implement it. After that introducing a fault attack on Salsa20 cipher based on realistic fault model and then exploring possible improvements of the attack.

2. Introduction

i. Cipher

In cryptography, a cipher is an algorithm for performing encryption or decryption – a series of well-defined steps that can be followed as a procedure.

Most modern ciphers can be categorized in several ways.

- Based on they work on blocks of symbols usually of a fixed size(**block ciphers**), or on a continuous stream of symbols(**stream ciphers**)
- Based on the key used for encryption and decryption whether the same key (**symmetric key algorithms**) is used for both or different key (**asymmetric key algorithms**) is used for each.

ii. Salsa20 Cipher

- Salsa20 is a stream cipher and works on 64-bytes blocks of data. For each 64-byte block of data, **Salsa20 expansion** function is used.
- The Salsa20 encryption function is a long chain of three simple operations on 32-bit words:
  - 32-bit addition, producing the sum a + b mod 232 of two 32-bit words a, b;
  - 32-bit exclusive-or, producing the xor of two 32-bit words a, b; and
  - constant distance 32-bit rotation, producing the rotation a <<< b of a 32-bit word a by b bits to the left, where b is constant.

iii. Mathematical Description of Salsa20 Cipher

In Salsa20, the encryption functions which are used are mentioned below:

❖ **Quarterround Function:**

- The quarterround function takes 4-words as input and returns another 4-word sequence.

- Input: $x = (x_0, x_1, x_2, x_3)$

quarterround(x) = (y_0, y_1, y_2, y_3)

where: $y_1 = x_1 * ((x_0 + x_3) <<< 7)$

$y_2 = x_2 * ((y_1 + x_0) <<< 9)$

$y_3 = x_3 * ((y_2 + y_1) <<< 13)$

$y_0 = x_0 * ((y_3 + y_2) <<< 18)$

* → XOR operation

2. **Rowround Function:**
- The rowround function takes 16 words as input and returns 16-word sequence.
- Input: $x = (x_0, x_1, x_2, ..., x_{15})$

rowround(x) = $(y_0, y_1, y_2, ..., y_{15})$

where:

(y_0, y_1, y_2, y_3) = quarterround(x_0, x_1, x_2, x_3)

(y_5, y_6, y_7, y_4) = quarterround(x_5, x_6, x_7, x_4)

$(y_{10}, y_{11}, y_8, y_9)$ = quarterround$(x_{10}, x_{11}, x_8, x_9)$

$(y_{15}, y_{12}, y_{13}, y_{14})$ = quarterround$(x_{15}, x_{12}, x_{13}, x_{14})$

3. **Columnround Function:**
- The columnround function takes 16 words as input and returns 16-word sequence.
- Input: $x = (x_0, x_1, x_2, ..., x_{15})$

columnround(x) = $(y_0, y_1, y_2, ..., y_{15})$

where:

(y_0, y_4, y_8, y_{12}) = quarterround(x_0, x_4, x_8, x_{12})

(y_5, y_9, y_{13}, y_1) = quarterround(x_5, x_9, x_{13}, x_1)

$(y_{10}, y_{14}, y_2, y_6)$ = quarterround$(x_{10}, x_{14}, x_2, x_6)$

$(y_{15}, y_3, y_7, y_{11})$ = quarterround$(x_{15}, x_3, x_7, x_{11})$

4. **Doubleround Function:**
- The doubleround function takes 16 words as input and returns 16-word sequence.

doubleround(x) = rowround(columnround(x));

5. **Littleendian Function:**
- The littleendian function that changes the order of a 4-byte sequence.
- Input: $b = (b_0, b_1, b_2, b_3)$

littleendian(b) = $b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$

6. **Salsa20 Expansion Function:**

The Salsa20 expansion function takes two sequences of bytes. First sequence 32 bytes and second one (n) is always 16-byte. It returns a sequence of 64 bytes.

$$\text{Salsa20Expansion}_{k0,\,k1}(n) = \text{Salsa20Hash}(a_0, k_0, a_1, n, a_2, k_1, a_3)$$

where:

$a_0 = (101, 120, 112, 97)$

$a_1 = (110, 100, 32, 51)$

$a_2 = (50, 45, 98, 121)$

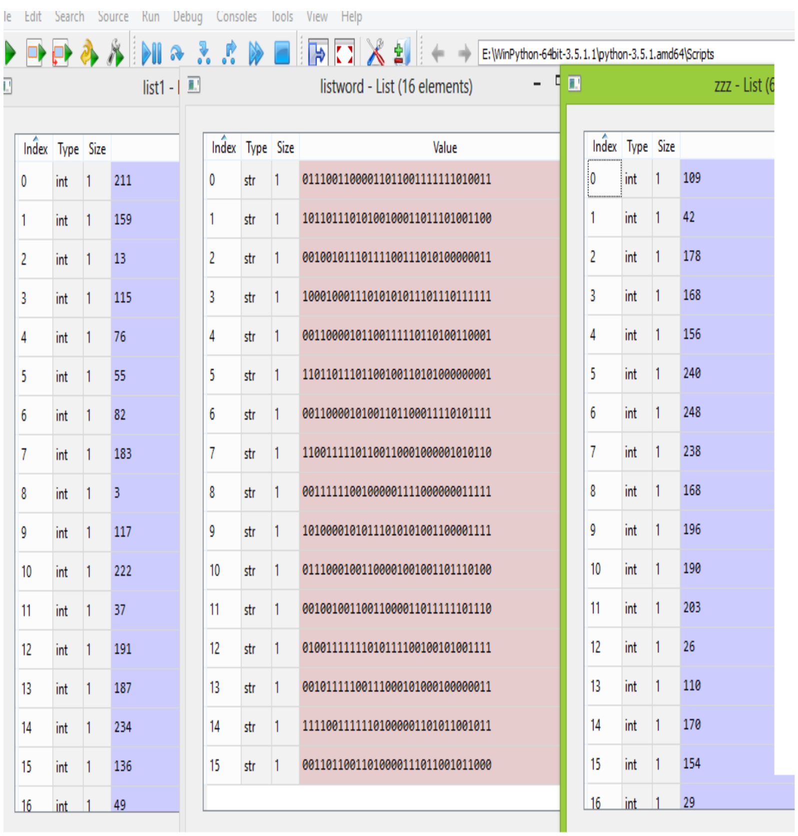
$a_3 = (116, 101, 32, 107)$

ko and k1 is the two part of 32-byte key (16 byte each).

3. Project Work

3.1. Implementation of Salsa20 Cipher:

In the implementation of the Salsa20 Cipher, all the function i.e. rowround, columnround, doubleround, quarterround, little-endian used in the Salsa20 were implemented in python. The key value which is 32byte, nonce which 8byte and 8byte block number is taken as input. The salsa expansion function uses 16byte by itself for the computations. That is total 32+8+8+16=64byte sequence is given as input to the Salsa20 hash function. In Salsa20 hash function 10 times doubleround function runs on the input.



Examples

Salsa20(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
= (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0).

Salsa20(211,159, 13,115, 76, 55, 82,183, 3,117,222, 37,191,187,234,136,
49,237,179, 48, 1,106,178,219,175,199,166, 48, 86, 16,179,207,
31,240, 32, 63, 15, 83, 93,161,116,147, 48,113,238, 55,204, 36,
79,201,235, 79, 3, 81,156, 47,203, 26,244,243, 88,118,104, 54)
= (109, 42,178,168,156,240,248,238,168,196,190,203, 26,110,170,154,
29, 29,150, 26,150, 30,235,249,190,163,251, 48, 69,144, 51, 57,
118, 40,152,157,180, 57, 27, 94,107, 42,236, 35, 27,111,114,114,
219,236,232,135,111,155,110, 18, 24,232, 95,158,179, 19, 48,202).

Fig.1 Screenshot of the output (16 word) of the Salsa20 cipher along with theoretical result

3.2. Fault attack:

i. Assumptions:

- It is assumed that the fault induced is single bit fault.
- Fault is induced in the random cycle.

ii. Basic idea:

- In the quarterround function, we can see that while computing the z_1 value, y_2 is not involved. Therefore z_1 is independent of y_2.
- So if we are able to do fault on the y_2 position (which is input to quarterround function for columnround function) after the completion of the 9th round of the doubleround function.
- So, the output of the columnround function will have same value of z_1 as it has without faulting . And after the rowround, all the values will be altered.

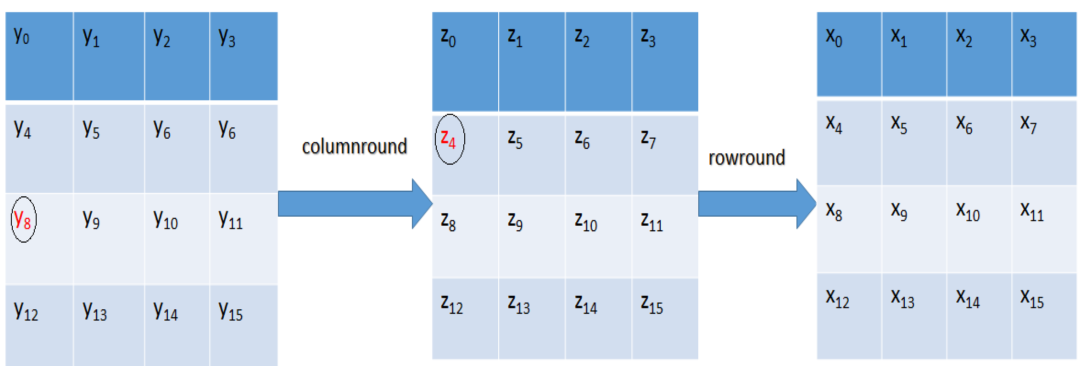


Fig.2 Representing the fault flow where z4 is fault free if fault is induced on y8

- If we do fault on y_2 after the completion of 9.5 round of double round. Then

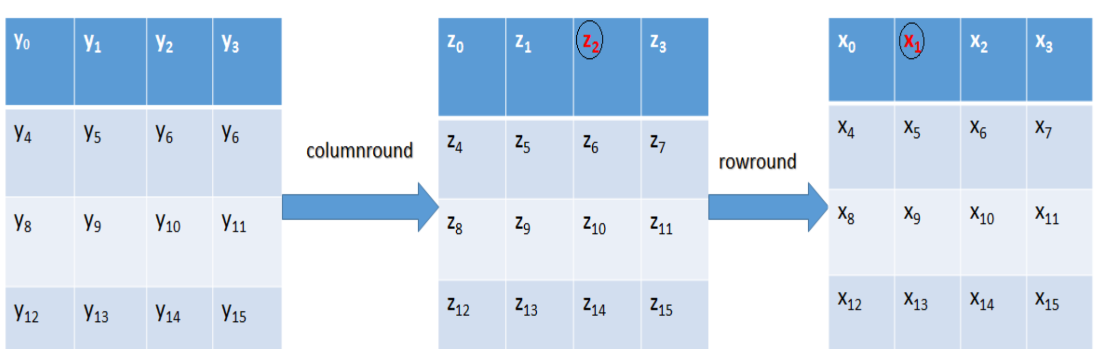


Fig.3Representing the fault flow where x1 is fault free if fault is induced on z2 after 9.5 round

- And now we can form 16 equation using faulty and without faulty output of the doubleround with XOR operation.
- Without fault Xi = f(y_0, y_1,y_2…….y_{15}). where i=0,1,2,….15
- After faulting Xi’ = f’(y_0, y_1,y_2…….y_{15}). where i=0,1,2,….15
- Therefore,

  Xi $\oplus$ Xi’ = f(y_0, y_1,y_2…….y_{15}) $\oplus$ f’(y_0, y_1,y_2…….y_{15}). where i=0,1,2,….15
- For example, if X_1, i=1

  X_1 $\oplus$ X_1’ = f(y_0, y_1,y_2…….y_{15}) $\oplus$ f’(y_0, y_1,y_2…….y_{15});
- By solving these equation we will get the key values.

iii. Complexity:

- In general, the brute force complexity for 64byte sequence is 2^{512}.
- But in the above discussed method, there will be 16 equation with six variables each.
- And all the variables are of 32bits.
- So complexity will be $2^{32*6} * 16 = 2^{196}$.

3.3 Solving Equations:

- Initially above mentioned equations are solved by assuming them as a 4-bit number.
- In that case complexity reduces to $2^{4*6} * 16 = 2^{28}$.
- For solving these equation, implementation is done in python.

4. Future work:

- To improve the complexity of the proposed method.
- To work with more than one bit fault.
- To solve these equation with improved complexity.

5. Conclusion

The proposed method reduces the complexity of fault attack from 2^{512} to 2^{196} , which much better than only doing brute force. And also if the number is considered as 4 bit then the complexity reduces to 2^{28} . And also further the complexity of the proposed method can improved as future works.

6. References

1. Daniel J. Bernstein, The Salsa20 family of stream ciphers (2007). URL: http://cr.yp.to/papers.html#salsafamily.
2. Dipanwita Roy Choudhury, Vincent Rijmen, Abhijit Das(Eds.), Progress in Cryptology INDROCRYPT 2008. 9th International Conference on Cryptology in India. Kharagpur, India , December 14-17, 2008
3. Daniel J. Bernstein, Salsa20 Design. URL: https://cr.yp.to/snuffle/design.
4. Daniel J. Bernstein, Chacha, a variant of Salsa20. URL: https://cr.yp.to/chacha/chacha-20080128.
5. http://www.ecrypt.eu.org/stream/e2-salsa20.html

Appendix1

Python program of Salsa20 cipher:

```
list1 = [211,159,13,115,76,55,82,183,3,117,222,37,191,187,234,136,
        49,237,179,48,1,106,178,219,175,199,166,48,86,16,179,207,
        31,240,32,63,15,83,93,161,116,147,48,113,238,55,204,36,
        79,201,235,79,3,81,156,47,203,26,244,243,88,118,104,54];

list3 = [];
i=0;
while i<64:
    list3.append('{0:08b}'.format(list1[i]))
    i=i+1;

# rotation function
def rotation(strg,n):
    return strg[:2]+strg[n:] + strg[2:n]

# littteendian functiom
listword = [];
j=0;
while j<64:
        listword.append('{0:032b}'.format(list1[j]+pow(2,8)*list1[j+1]+pow(2,16)*list1[j+2]+po
w(2,24)*list1[j+3]));
        j=j+4;

zz=listword[:];

def quarterround(x,y,z,w):
    #z[1]=y1^((y1+y3)<<<7),        z[2]=y2^((z1+y0)<<<9)
    #z[3]=y3^((z2+y1)<<<13),       z[0]=y0^((z3+z2)<<<18)
    zz[y]='{0:032b}'.format(int(zz[y],2) ^ int((rotation(bin(int(zz[y],2)+int(zz[w],2)),-7)),2))
    zz[z]='{0:032b}'.format(int(zz[z],2) ^ int((rotation(bin(int(zz[y],2)+int(zz[x],2)),-9)),2))
    zz[w]='{0:032b}'.format(int(zz[w],2) ^ int((rotation(bin(int(zz[z],2)+int(zz[y],2)),-13)),2))
    zz[x]='{0:032b}'.format(int(zz[x],2) ^ int((rotation(bin(int(zz[w],2)+int(zz[z],2)),-18)),2))

def rowround():
    #(z0,z1,z2,z3)
    quarterround(x=0,y=1,z=2,w=3);
    #(z4,z5,z6,z7) = quarterround(y4,y5,y6,y7)
    quarterround(x=4,y=5,z=6,w=7);
    #(z8,z9,z10,z11) = quarterround(y8,y9,y10,y11)
    quarterround(x=8,y=9,z=10,w=11);
    #(z12,z13,z14,z15) = quarterround(y12,y13,y14,y15)
    quarterround(x=12,y=13,z=14,w=15);
```

```
def columnround():
    #(y0,y4,y8, y12) = quarterround(x0,x4,x8,x12)
    quarterround(x=0,y=4,z=8,w=12);
    #(y5,y9,y13, y1) = quarterround(x5,x9,x13,x1)
    quarterround(x=5,y=9,z=13,w=1);
    #(y10,y14,y2, y6) = quarterround(x10,x14,x2,x6)
    quarterround(x=10,y=14,z=2,w=6);
    #(y15,y3,y7, y11) = quarterround(x15,x3,x7,x11)
    quarterround(x=15,z=3,y=7,w=11);

def doubleround():
    #  doubleround(x)=
    rowround(columnround());

# Calling 10 time doubleround function
i=0;
while i<10
       columnround();
       rowround();
       i=i+1;

keybin=[];
keydec=[];

#Calculating back the 64 byte sequence
i=0;
while i<16:
    a=zz[i];
    j=0;
    while j<32:
         keybin.append(a[j:j+7]);
         keydec.append(int(a[j:j+7],2));
         j=j+8;
    i=i+1;
k=0;
while k<64:
    print((keydec[k]));
    k=k+1;
```

Appendix2

Program solving equation of 4 bit binary number:

```
list1 = [2,2,2,2,1,2,1,3,1,1,1,0,1,2,3,4];

list3 = [];

i=0;
while i<16:
    list3.append('{0:04b}'.format(list1[i]))
    i=i+1;

# rotation function

def rotation(strg,n):
    return strg[:2]+strg[n:] + strg[2:n]

#z1= '{0:032b}'.format(int(listword[0],2)+int(listword[3],2));
zz=list3[:];

def quarterround(x,y,z,w):
    zz[y]='{0:04b}'.format(int(zz[y],2) ^ (int(zz[y],2)+int(zz[w],2)))
    zz[z]='{0:04b}'.format(int(zz[z],2) ^ (int(zz[y],2)+int(zz[x],2)))
    zz[w]='{0:04b}'.format(int(zz[w],2) ^ (int(zz[z],2)+int(zz[y],2)))
    zz[x]='{0:04b}'.format(int(zz[x],2) ^ (int(zz[w],2)+int(zz[z],2)))

def rowround():
    #(z0,z1,z2,z3)
    quarterround(x=0,y=1,z=2,w=3);
    #(z4,z5,z6,z7) = quarterround(y4,y5,y6,y7)
    quarterround(x=4,y=5,z=6,w=7);
    #(z8,z9,z10,z11) = quarterround(y8,y9,y10,y11)
    quarterround(x=8,y=9,z=10,w=11);
    #(z12,z13,z14,z15) = quarterround(y12,y13,y14,y15)
    quarterround(x=12,y=13,z=14,w=15);

def columnround():
    #(y0,y4,y8, y12) = quarterround(x0,x4,x8,x12)
    quarterround(x=0,y=4,z=8,w=12);
    #(y5,y9,y13, y1) = quarterround(x5,x9,x13,x1)
    quarterround(x=5,y=9,z=13,w=1);
    #(y10,y14,y2, y6) = quarterround(x10,x14,x2,x6)
    quarterround(x=10,y=14,z=2,w=6);
    #(y15,y3,y7, y11) = quarterround(x15,x3,x7,x11)
    quarterround(x=15,z=3,y=7,w=11);
```

```
def doubleround():
   #  doubleround(x)=
   rowround(columnround());

columnround();
rowround();

listfault=list1[:];
a=b=c=d=e=f=0;
for a in 15:
   for b in 15:
      for b in 15:
         for d in 15:
            for e in 15:
               for f in 15:
                  int(list1[1],2)^int(listfault[a],2)='{0:04b}'.format((int(lis1t[a],2)                    ^
(int(list1[b],2)+int(list1[c],2))) ^ (int(listfault[d],2) ^ (int(listfault[e],2)+int(listfault[f],2))));

keybin=[];
keydec=[];

i=0;
while i<16:
   a=zz[i];
   j=0;
   while j<32:
       keybin.append(a[j:j+7]);
       keydec.append(int(a[j:j+7],2));
       j=j+8;
   i=i+1;
k=0;
while k<64:
   print((keydec[k]));
   k=k+1;
```