Moustique Cipher

EE720 An Introduction to Number Theory and Cryptography

Rathour Param Jitendrakumar | 190070049
Prathamesh Pradip Dhake | 190070048

Autumn Semester 2021-22

Contents

1 Introduction

Moustique is updated version of Mosquito stream cipher. Moustique is a Self-synchronizing stream cipher which was broken in the final round of the eStream project. Here we present it's reduced size analog[1].

2 Encryption and Decryption

For encryption, each plaintext bit is added (over GF(2)) to a keystream bit to get ciphertext bit.

$$c(i) = m(i) + z(i) \tag{1}$$

For decrpytion, the same keystream bit should be added (over GF(2)) to the corresponding ciphertext bit.

$$m(i) = z(i) + c(i) \tag{2}$$

For a self-synchronizing stream encryption, the keystream bit $z(i)$ is same as applying a cipher function f_c to a range of ciphertext stream where n_m is input memory, b_s is cipher function delay.

$$z(i) = f_c[K](c(t - n_m) \dots c(t - (b_s + 1))) \tag{3}$$

For this assignment as our interest is in generating keystream bit, we focus our analysis on the cipher function.

3 The Reduced Moustique Cipher Function

In our reduced version of this cipher function, we take the key and IV size as 48 bits (96 in original paper) and we use 4 register sizes of 64, 53 (2 registers), 12 and 3 bits using up a total 185 states compared to 408 in original.

- The 64 bit register called as conditional complementing shift register (CCSR) is the only 1-indexed register. Also, it is organized a bit differently, CCSR(x) is written as $q_{i,j}$ and the pair of i, j uniquely maps to $x \in \{1, 2, \dots, 64\}$ given in Table 1. (i indices are given vertically, j's are horizontal and the corresponding cell gives x in CCSR)
- There are 2 53 bit registers a_{11} and a_{12} collectively called as a_1
- 12 bit register a_2
- 3 bit register a_3

[1] Also available at https://github.com/paramrathour/Moustique-Cipher

							64	7
							63	6
							62	5
							61	4
					58	59	60	3
					55	56	57	2
		49	50	51	52	53	54	1
...	42	43	44	45	46	47	48	0
41	42	43	44	45	46	47	48	

Table 1: CCSR at higher bits

3.1 State update map

The update uses the following three functions

$$\begin{aligned} g_0(a,b,c,d) &= a+b+c+d \\ g_1(a,b,c,d) &= a+b+c(d+1)+1 \\ g_2(a,b,c,d) &= a(b+1)+c(d+1) \end{aligned} \tag{4}$$

We can consider the state as a tuple (CCSR, a_1, a_2, a_3). The update to it's each element is given by[2]

CCSR Update For $j <= 2$,

$$q(i,j) = g_x(q(i \ \% \ n_{j-1}, j-1), K(j-1), q(i,0), q(i,0)) \tag{5}$$

For other $j \neq 48$,

$$q(i,j) = g_x(q(i \ \% \ n_{j-1}, j-1), K(j-1), q(i \ \% \ n_v, v), q(i \ \% \ n_w, w)) \tag{6}$$

For $j = 48$,

$$q(i,48) = g_x(q(i \ \% \ n_{47}, 47), q(0, 47-i), q(i \ \% \ n_{46}, 46), q(0, 46-i)) \tag{7}$$

Here n_j means the number of cells in column j of Table 1 and the values x, v, w are chosen according to Table 2.

Index (= $(j-i)$ % 6)	Function	v	w
1,4	g_0	$2 \cdot (j-i-1)/3$	$j-2$
2,5	g_1	$j-4$	$j-2$
3	g_1	0	$j-2$
0	g_1	$j-5$	0

Table 2: Function and v & w values

a_1 **Update**

$$\begin{aligned} &\text{for } i \in \{0,1,\ldots,52\} \quad a_{11}(4i \ \% \ 53) = g_1(\text{CCSR}(64-i), \text{CCSR}(i+9), \text{CCSR}(57-i), \text{CCSR}(i+1)) \\ &\text{for } i \in \{0,1,\ldots,52\} \quad a_{12}(4i \ \% \ 53) = g_1(a_{11}(i), a_{11}(i+3), a_{11}(i+1), a_{11}(i+2)) \end{aligned} \tag{8}$$

a_2 **Update**

$$\text{for } i \in \{0,1,\ldots,12\} \quad a_2(i) = g_1(a_{12}(4i), a_{12}(4i+3), a_{12}(4i+1), a_{12}(4i+2)) \tag{9}$$

a_3 **Update**

$$\text{for } i \in \{0,1,\ldots,3\} \quad a_3(i) = g_0(a_2(4i), a_2(4i+1), a_2(4i+2), a_2(4i+3)) \tag{10}$$

3.2 Output map

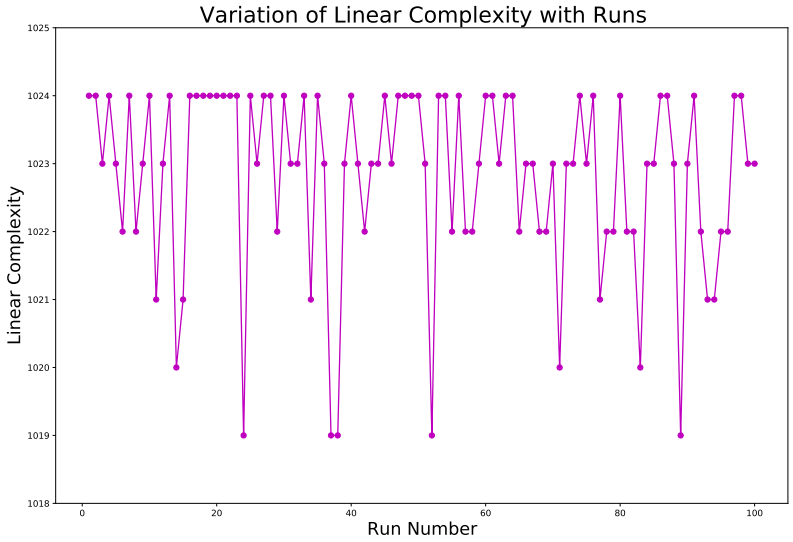
The keystream bit is given by

$$z = a_3(0) + a_3(1) + a_3(2) \tag{11}$$

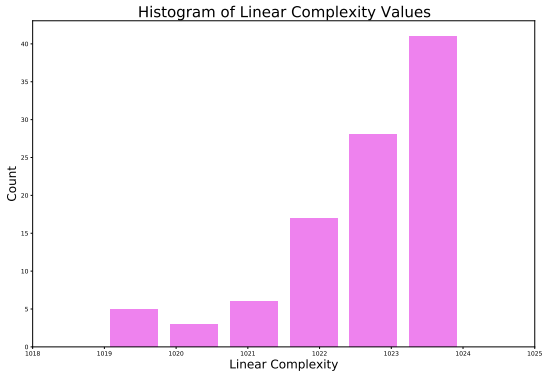
[2] if any index values goes out of range we consider them to be zero

4 Plots and Observations

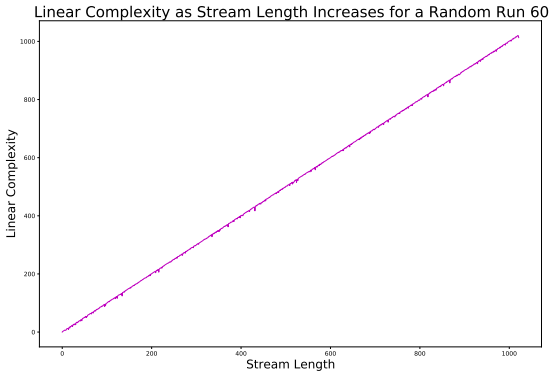
To obtain these plots, we took a random key and 100 random initialising vectors and calculated the first 1024 keystream bits for each key, IV pair. (initial 48 bits were = key + IV)



(a)



(b)



(c)

Figure 1: Linear Complexity Plots

- Figure 1a shows the variation of Linear Complexity (LC) for each of the 100 runs. The minimum LC is 1019 and the maximum is 1024. The LC is very high, for every run as it should be for a random keystream.
- Figure 1b is a histogram of LC values. We see that higher LC values are more frequent than lower values. This shows that our cipher is achieving the optimal LC for large number of run ($\sim 40\%$)
- In Figure 1c, a random sample from the 100 runs (run 60) was taken and the LC variation with it's length was plotted. It is clear that for every length of keystreams (¡=1024), the achieved LC is very close to that length.

It is clear from above that the reduced size cipher generated satisfactory keystreams.

5 SageMath Code

```
from sage.matrix.berlekamp_massey import berlekamp_massey
import numpy as np
import bisect
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

# Function definitions
def init(key, F, register_lengths, row_count, r_1_num):
    CCSR = list(random_matrix(F, 1, register_lengths[0]+1).row(0))    # 1-indexed
    r_1 = random_matrix(F, r_1_num, register_lengths[1]+3)
    r_2 = list(random_matrix(F, 1, register_lengths[2]).row(0))
    r_3 = list(random_matrix(F, 1, register_lengths[3]).row(0))

    sum = np.zeros(len(row_count), dtype = int)
    sum[0] = row_count[0]
    for i in range(1, len(row_count)):
        sum[i] = row_count[i] + sum[i-1]

    mapping = {}
    mapping[0,0] = 0
    column_count = np.zeros(len(key)+1, dtype = int)
    column_count[0] = 1
    for i in range(1,register_lengths[0]+1):
        idx = bisect.bisect_left(sum, i)
        j = 48 + i - sum[idx]
        mapping[idx, j] = i
        column_count[j] += 1
    for i in range(2 + column_count[len(key)]):
        j = len(key) - i
        if (1, j) not in mapping:
            mapping[1, j] = 0

    return CCSR, r_1, r_2, r_3, mapping, column_count

def g(i, a, b, c, d):
    if i == 0:
        return a + b + c + d
    elif i == 1:
        return 1 + a + b + c * (d + 1)
    elif i == 2:
        return a * (1 + b) + c * (1 + d)
    else:
        return 0

def parameters(i,j):
    value = (j-i) % 6
    if value == 1 or value == 4:
        return 0, 2*(j-i-1)/3, j-2
    elif value == 2 or value == 5:
        return 1, j-4, j-2
    elif value == 3:
        return 1, 0, j-2
    else:
        return 1, j-5, 0

def state_update(F, key, CCSR, r_1, r_2, r_3, mapping, column_count,
                 register_lengths, r_1_num, row_count):
    CCSR_new = list(random_matrix(F, 1, register_lengths[0]+1).row(0))    # 1-indexed
```

```
    for j in range(1, len(key)+1):
        for i in range(column_count[j]):
            x, v, w = parameters(i, j)
            if j < 3:
                v = 0
                w = 0
                c = 0
                d = 0
            a = CCSR[mapping[i%column_count[j-1], j-1]]
            b = key[j-1]
            c = CCSR[mapping[i%column_count[v], v]]
            d = CCSR[mapping[i%column_count[w], w]]
            if j == len(key):
                x = 2
                b = CCSR[mapping[0, j-1-i]]
                c = CCSR[mapping[i%column_count[j-2], j-2]]
                d = CCSR[mapping[1, j-2-i]]
            CCSR_new[mapping[i, j]] = g(x, a, b, c, d)
    CCSR = CCSR_new;
    for i in range(register_lengths[1]):
        a = CCSR[register_lengths[0] - i]
        b = CCSR[i + int(register_lengths[0] * 18/128)]
        c = CCSR[int(register_lengths[0] * 113/128) - i]
        d = CCSR[i + 1]
        r_1[0, (4*i) % register_lengths[1]] = g(1, a, b, c, d)
    for j in range(1, r_1_num):
        for i in range(register_lengths[1]):
            a = r_1[j-1, i]
            b = r_1[j-1, i+3]
            c = r_1[j-1, i+1]
            d = r_1[j-1, i+2]
            r_1[j, (4*i) % register_lengths[1]] = g(1, a, b, c, d)
    for i in range(register_lengths[2]):
        a = r_1[r_1_num-1, 4*i]
        b = r_1[r_1_num-1, 4*i+3]
        c = r_1[r_1_num-1, 4*i+1]
        d = r_1[r_1_num-1, 4*i+2]
        r_2[i] = g(1, a, b, c, d)
    for i in range(register_lengths[3]):
        a = r_2[4*i]
        b = r_2[4*i+1]
        c = r_2[4*i+2]
        d = r_2[4*i+3]
        r_3[i] = g(0, a, b, c, d)
    return CCSR, r_1, r_2, r_3

def output_stream(key, CCSR, r_1, r_2, r_3):
    return sum(r_3)

def generate_stream(F, key, IV, CCSR, r_1, r_2, r_3, mapping, column_count,
                    register_lengths, r_1_num, row_count, size):
    key_combined = [key[i] + IV[i] for i in range(len(key))]
    output = []
    for i in range(min(size, len(key_combined))):
        output.append(key_combined[i])
    for i in range(len(key_combined), size):
        CCSR, r_1, r_2, r_3 = state_update(F, key_combined, CCSR, r_1, r_2, r_3, mapping,
                                    column_count, register_lengths, r_1_num, row_count)
        output.append(output_stream(key, CCSR, r_1, r_2, r_3))
    return output
def generate_data(IV_num, size):
```

```
    F = GF(2)
    register_lengths = [64, 53, 12, 3]
    r_1_num = 2
    row_count = [48, 6 , 3, 3, 1, 1, 1, 1]
    output = []
    IVs = []
    key_length = 48
    key = list(random_matrix(F, 1, key_length).row(0))
    for i in range(IV_num):
        CCSR, r_1, r_2, r_3, mapping, column_count = init(key, F, register_lengths,
                                                          row_count, r_1_num)
        IV = list(random_matrix(F, 1, key_length).row(0))
        IVs.append(IV)
        output.append(generate_stream(F, key, IV, CCSR, r_1, r_2, r_3, mapping,
                                      column_count, register_lengths, r_1_num, row_count, size))
    return key, IVs, output

# Call functions
set_random_seed(0)
key, IVs, output = generate_data(100, 1024) # 100 is number of IVs, 1024 is stream length
linear_complexities = []
for o in output:
    linear_complexities.append(berlekamp_massey(o+o).degree())
# berlekamp_massey gives correct result when period <= half length, hence stream taken twice

# Figure 1
fig, ax = plt.subplots()
ax.plot(np.arange(1, len(linear_complexities)+1), linear_complexities, '-om')
ax.set_xlabel('Run Number', fontsize = 20)
ax.set_ylabel('Linear Complexity', fontsize = 20)
fig.set_size_inches(15, 10, forward=True)
ax.set_ylim([1018, 1025])
ax.set_title('Variation of Linear Complexity with Runs', fontsize = 25)
plt.savefig('1.pdf')

# Figure 2
bins_num = len(np.unique(linear_complexities))
fig, ax = plt.subplots()
ax.hist(linear_complexities, bins = bins_num, rwidth = 0.8, color = 'violet')
ax.set_xlabel('Linear Complexity', fontsize = 20)
ax.set_ylabel('Count', fontsize = 20)
fig.set_size_inches(15, 10, forward=True)
ax.set_xlim([1018, 1025])
ax.set_title('Histogram of Linear Complexity Values', fontsize = 25)
plt.savefig('2.pdf')

# Figure 3
r = ZZ.random_element(0,99)
linear_complexities_one = []
for i in range(2, len(output[r])):
    linear_complexities_one.append(berlekamp_massey(o[1:i]+o[1:i]).degree())
fig, ax = plt.subplots()
ax.plot(linear_complexities_one, '-m')
ax.set_xlabel('Stream Length', fontsize = 20)
ax.set_ylabel('Linear Complexity', fontsize = 20)
fig.set_size_inches(15, 10, forward=True)
ax.set_title('Linear Complexity as Stream Length Increases for a Random Run {}'.
             format(r), fontsize = 25)
plt.savefig('3.pdf')
```

6 References

Matthew Robshaw (auth.), *New Stream Cipher Designs The eSTREAM Finalists*, Lecture Notes in Computer Science
Daemen, J., Kitsos, P.: The self-synchonizing stream cipher MOUSTIQUE (2006)
Andreas Klein (auth.), *Stream Ciphers*, Springer
`bisect.bisect_left` for efficient mapping of CCSR
Sage Immutable Vector Error