Final report of

|  |
| --- |
| ***Cryptanalytic Techniques for Sequence Generators with Cryptographic Application*** |

**Name: LI Bin**

**Student ID: 20296588**

**Date: 2016-12**

CONTENTS

[CONTENTS 2](#_Toc468356139)

[Abstract 3](#_Toc468356140)

[1 Introduction 4](#_Toc468356141)

[2 preliminaries 5](#_Toc468356142)

[3 technical contributions 7](#_Toc468356143)

[4 Detailed description of the results 9](#_Toc468356144)

[5 Personal comments 11](#_Toc468356145)

[ACKNOWLEDGEMENT 13](#_Toc468356146)

[References 14](#_Toc468356147)

[Appendix A. MATLAB Source code 15](#_Toc468356148)

[CRYPTANALYSIS 15](#_Toc468356149)

[FUNCTION: Modula Computation of the Variable d 21](#_Toc468356150)

[FUNCTION: Zech Logarithm 21](#_Toc468356151)

[Appendix B. MAPLE Source code 22](#_Toc468356152)

Abstract

This research mainly elaborates the feasibility and the efficiency about a cipher sequence being analyzed and attacked by algorithms implemented in software. A variety of sequence generators are introduced, including their characteristics, period and linear complexity. Keystream sequences generated by shrinking generators are emphasized in terms of the attack techniques. Some numerical results of recovering the shrunken sequences generated by shrinking generators over different degrees will be demonstrated and explained in detail. All the experiments and simulations are conducted on one laptop and one computer. So it is possible that the results could be slightly different under other circumstance.

*Keywords: Keystream Sequence, Zech Logarithm, Cryptanalysis, Numerical Results*

1 Introduction

All information can be represented and transported in binary sequence form. The most important component of sequence generation is linear feedback shift register (LFSR). By bitwise XOR-ing, the original text could be encrypted by key stream sequence and become cipher-text. However, not all those sequences are qualified to be used in encryption due to the fact that some of those sequences have very short linear complexity.

In this article, shrinking generator is mainly discussed because the cryptanalysis is based on sequences generated by shrinking generators. Some simulations based on MATLAB will be demonstrated in terms of attacking time, number of intercepted bits, number of correct candidates and different periods over different degrees.

This report is organized as follows: In section 2, a variety of generators, the basic computations and algorithm are introduced. In section 3, some technical contributions of efficiently processing cryptanalysis are clarified. Relevant statistic results are shown and explained in section 4. Finally, conclusions and comments are included in section 5.

2 preliminaries

A **linear feedback shift register** (LFSR) is an electronic device with **L** cells. It can also be represented by the form of a polynomial. If a polynomial is primitive, the sequence generated by this LFSR has the maximal period of 2N-1, which can be also called PN-sequence. Another important characteristic of sequences is **linear complexity** (LC). LC is the minimal length of the LFSR that is able to generate such a sequence. The greater LC is, the more secure the sequence is.

The **shrinking generator** consists of two registers. L1 and L2 are the lengths of each register respectively. The period **T** of the output sequence is (2L2-1)2(L1-1) and the **Linear Complexity** (**LC**) is L2\*2(L1-2)<LC≤L2\*2(L1-1). Define {} as the sequence generated by register 1 and {} as the sequence generated by register 2. {Zi} is the output sequence. The decimation rule is as follows:

If {} is 1, then {} is selected, otherwise {} is discarded.

{}: 0 1 0 1 0 0 1 1 1 0 1 0 ……

{}: 1 0 1 0 1 0 1 0 0 1 1 1 ……

{Zi}: 0 0 1 0 0 1

i = 0, 1, 2, 3, 4 …

The **self-shrinking generator** consists of only one LFSR. If L is the length of the register, then the period **T** of the output sequence is 2[L/2]<**T**≤2(L-1) and LC is 2[L/2]-1<**LC**≤2L-1-(L-2). {Zi} is the output sequence. The output sequence is determined as:

{U­­i}: (1 0) (1 0) (0 1) (0 1) (1 1) (0 0) ……

{Zi}: 0 0 1…

i = 0, 1, 2, 3, 4 …

If U­­2i is 1, then U2i+1 is selected.

The **modified self-shrinking generator** is similar to the self-shrinking generator. If L is the degree of the register, the period **T** of the output sequence is 2[L/3]≤**T**≤2(L-1) (if L is an odd number between 3 and 20, the value is the maximum, 2(L-1)) and **LC** is 2[L/3]-1≤LC≤2L-1-(L-2). {U­­i} is the sequence generated by the LFSR. {Zi} is the output sequence. The decimation rule is as follows:

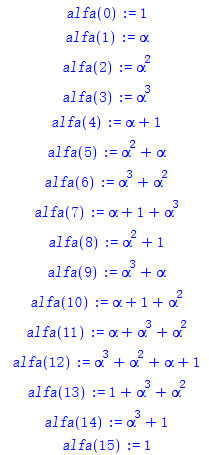
{U­­i}: (1 1 *1*) (0 1 **0**) (1 0 **0**) (0 0 *1*) (1 1 *0*) (1 0 **1**) (0 1 **1**) ……

{Z­­i}: 0 0 1 1…

i = 0, 1, 2, 3, 4 …

If U­­2i+ U­­2i+1=1, then U­­2i+2 is selected, otherwise U­­2i+2 is rejected.

The **Generalized self-shrinking generator** generates a sequence family. Every row of the family set is a shifted version of the original PN-sequence. And all the rows are combined with original PN-sequence respectively, working as shrinking generators. Those selected bits compose a new family set. Every row of the new family set is an output sequence. And they have their own **Linear Complexities** and periods in each line.

**Definition 1. Zech logarithm** is a function of finding value through the particular set. The assigning positions of intercepted bits are positons over the PN-sequence generated by characteristic polynomial **p(x)**, which always has the same degree with the characteristic polynomial **P2(x)** of register2. The general expression of Zech logarithm is ****, where **ɑ** is the root of the characteristic polynomial **p(x)**. If Zɑ(n)=p, that means **1+ɑn=ɑp**. And after the exhaustive search, p is the wanted number satisfying the expression **1+ɑn=ɑp**. It is very important to get assigning positions of recovery sequence from the logarithm. For example, if **p(x)** is **x4+x+1**, whose degree **L** is **4**, according to the general expression, it could be written as **ɑ4+ɑ+1=0**, also **ɑ4=ɑ+1.** The whole set could be iteratively represented by bases **ɑ(0)**, **ɑ(1)**, **ɑ(2)** and **ɑ(3)**, for example, **ɑ5=ɑ2+ɑ**. So **ɑ(*j*)**= **ɑ(*j-3*)**+**ɑ(*j-4*),** *j*=N,N+1,N+2,…,2N-1.

(As shown above)

If **5** and **7** were inputs, **output**=**Z(5-7)=Z(-2)=Z(-2+24-1)=Z(13)=ɑ(*13*)+1=ɑ(*6*)=6**. The **output position** is calculated by **Zɑ(5-7)+7**,which is **13**. If the value of the **output position** is greater than **2N-1**, the **output position** would be changed to **Z(ɑ1-ɑ2)+ ɑ2 - (2N-1).** Obviously, when degree becomes large, the whole set will exponentially increase and the exhaustive search for the position will cost much more time.

**Cryptanalysis** procedure is complicated and comprehensive. This report gives the whole procedure of attacking the keystream sequence generated by a shrinking generator. As introduced, shrinking generator consists of two registers with different lengths. The known parameters include intercepted bits from the keystream sequence, the lengths of two registers, the PN-sequence of the first register, **p(x)** and also Zech logarithm. First we get **N** intercepted bits as well as the same number of positions **{P\_Oi}**of **1**s from the PN-sequence. Computation of first row of assigning position **{P1i}** (i=1,2,3,…,N) only needs **d**(only related to the lengths of both registers). The second line of intercepted bits is computed by XOR-ing the adjacent bits of the first line of intercepted bits, thus getting **N-1** intercepted bits at the second line. The second line of assigning positions **{P2i}** also has **N-1** factors. By entering two adjacent numbers of positions to Zech logarithm, it outputs one position number, as described above.

If a position were assigned with different bits, that would be one **contradiction**, which means that the chosen initial state is incorrect. If no **contradiction** were found, that initial state would be potentially correct, thus narrowing down the range of total initial states. That is the whole procedure of **Cryptanalysis**.

The reason why I choose to use MATLAB is because MATLAB is an excellent tool allowing matrix manipulations and implementations of algorithms. Also it can save all the parameters after the program finishes, such as running time and every meaningful value we need, without entering additional instructions.

3 technical contributions

All the simulations are based on MATLAB R2014a(Intel(R) Core(TM) i5-3230M, 2.6GHz\*4, 8G, 64bits) or 2015a(Intel(R) Core(TM) i7-4790M, 3.6GHz\*8, 16G, 64bits) versions. Only basic library is utilized. The whole idea of algorithm is as following:

**Algorithm 1** Recover **P(x)** sequence

**Input**: the number of intercepted bits

01: Load **keystream sequence** and **PN sequence of R1**

02: Generate the whole set of **Intercepted Bits**

03: ***for* initial state** **1** to **end**

04:Generate and store positions of 1s of **PN sequence** in **pos\_original**

05: Generate and store actual assigning **position 1**

06: Assign first line **Intercepted bits** according to **position 1**

07: ***for*** line 2 to length (**Intercepted Bits**)

08: Use **Zech-logarithm** to generate the rest of assigning positions

10: Assign every bit to their corresponding position and find out contradiction

11: ***if***contradiction is found

12: output contradiction position and corresponding line

13: ***end if***

14: Output and store correct **initial state**

15: ***end for***

16: ***end for***

17: **Output**: correct candidates, number of correct candidates, running time, number of how many times contradictions were found in every line,

18: save workspace

The tricky parts are included in the whole programming procedure.

First of all, once the degrees and polynomials of two LFSRs are determined, the PN sequence of the LFSR1 (lower degree) and shrunken sequence are determined, so as any number of intercepted bits. If intercepted bits are determined, the set of intercepted bits are determined, no matter how initial states change, which can save memory while running the program.

Secondly, in order to perform an exhaustive search over the initial state of the LFSR1, those initial states starting with ‘0’ should be rejected. While those states starting with ‘1’, which could be found in PN sequence generated by the LFSR1, can be considered as all the potential initial states over L (the length of LFSR1) bits.

Meanwhile, selecting initial states by shifting bits of PN sequence is an efficient way to reduce Zech-logarithm computations, thus saving running time. This case could be explained by a given example: Characteristic polynomials of two LFSRs are X4+X3+1 and X5+X3+1, whose shrunken sequence is ‘101011100111101…’. PN sequence of LFSR 1 is ‘100011110101100…’. When number of intercepted bits is 6, ‘101011’, six 1s are selected from PN sequence 1 with their respective positions ‘0, 4, 5, 6, 7, 9’. Assigning positions computed by Zech logarithm are ‘0, 23, 21, 19, 17, 13’. Next initial state starting by the second 1 is ‘1111’. Same as before, six 1s are selected from the PN sequence, whose positions are ‘0, 1, 2, 3, 5, 7’. Assigning positions computed by Zech logarithm are ‘0, 29, 27, 25, 21, 17’. Apparently those assigning positions between consecutive initial states have same relative positions, which means we can assign intercepted bits according to assigning positions without computing them every time initial states change, thus avoiding computing the most time consuming process, Zech logarithm. Because when degrees become higher, for example 19 and 20, it is impossible for computers to do the exhaustive search among 107 numbers to find the correct position every time the computer computes the Zech logarithm.

Besides finding correct potential initial states, such useful information as positions where contradictions are found or running time is also collected. Details are shown in the next section.

4 Detailed description of the results

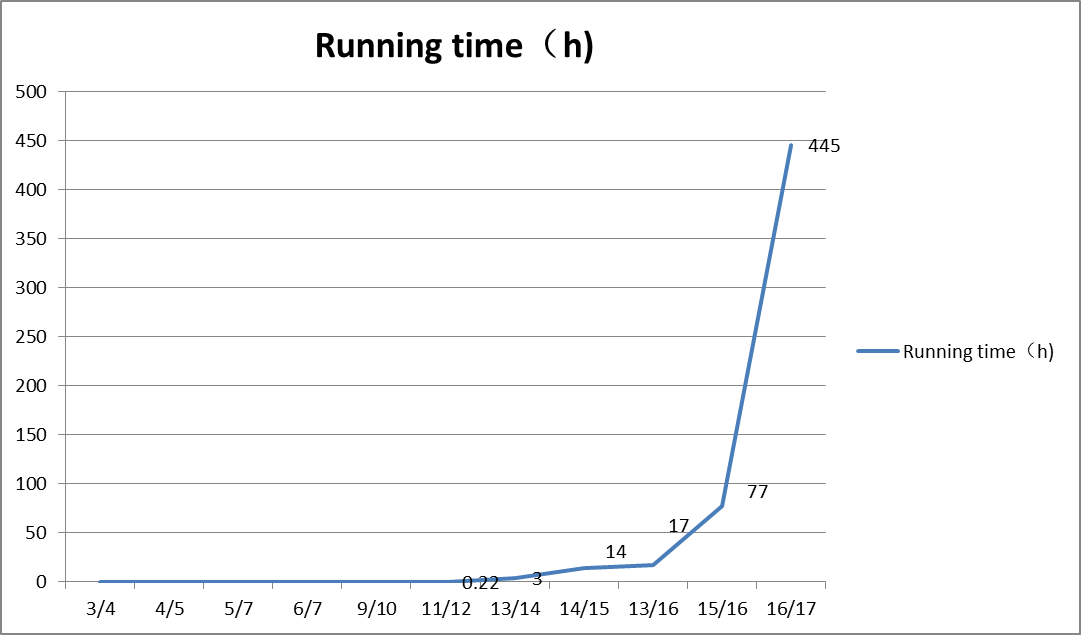
By considering running time of different degrees on the same computer, a rough estimation over running time of higher degrees of shrinking generators could be made.

**Figure 1**

According to Figure 1, with the increase of degrees of both registers, larger number of intercepted bits is needed in order to narrow down the range of potential correct initial states, and the proportions of number of intercepted bits and period are becoming smaller. That is to say we need a very small number of intercepted bits to track down the correct initial state.

Not all these number of intercepted bits are the minimum number of selecting one correct initial state. For example, with the degree pair of 13 and 16, 52 might not be the smallest number to select one correct initial state.

That the best a computer could do now is to process degree pair 16 and 17. The processing time is about three weeks.



**Figure 2**

According to Figure 2, the processing time is increasing rapidly with the increase of the degrees. That is because the Zech set is exponentially increasing and also the number of intercepted bits increases slightly. The main factor of long running time could be the highest degree of all registers. This can be concluded by pairs of 14/15 and 13/16.

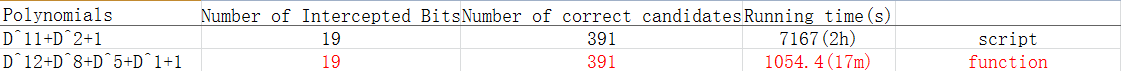
Regarding the same pair of degree, greater number of intercepted bits would cause longer processing time.

The pair of degrees such as 14/16 cannot be analyzed because 14 and 16 have the common factor, 2, resulting that **d** cannot be computed

The computation limit of the personal computer is the pair of degrees 13/14. The estimated time of analysis over 30 intercepted bits is 37 hours. While the same analysis (degree 16/17, 58 intercepted bits, 10 states analyzed) was running at laptop and high-configured computer, the running times are 6997s and 648s respectively. The processing ability of the computer is roughly ten times faster than that of the laptop.

5 Personal comments

Due to the large searching set and long processing time, it would be disappointing if accidents happened and caused program crash. The modification is to save the data of program periodically. This can effectively prevent data from accidentally disappearing.

If we use **functions** instead of scripts, running time could be dramatically reduced. Because scripts always load unnecessary intermediate parameters to the memory every time they use those parameters. However functions use ‘fake code’. They load parameters only once during the whole process. Here is a given example of comparison between running a function and script:

**Preallocation** of matrix is very necessary in terms of saving running time, especially for a large matrix inside a loop. For example,

tic;

a(1) = 1;

b(1) = 0;

for k = 2:8000

a(k) = 0.99803 \* a(k-1)-0.06279 \* b(k-1);

b(k) = 0.06279 \* a(k-1) + 0.99803 \* b(k-1);

end

toc

Elapsed time is 0.030076 seconds.

tic;

a=zeros(1,8000);

b=zeros(1,8000);

a(1) = 1;

b(1) = 0;

for k = 2:8000

a(k) = 0.99803 \* a(k-1)-0.06279 \* b(k-1);

b(k) = 0.06279 \* a(k-1) + 0.99803 \* b(k-1);

end

toc

Elapsed time is 0.025812 seconds.

0.030076/0.025812=1.165. This may not be a desirable boost, but when the size of matrix is large enough, the cost time would be much less.

Try to use the functions already embedded in MATLAB. For example, use **zeros** to define an all-zero vector, instead of using **[0,0,0,0,…,0]**.

However, there are still some programs with long processing time (as estimated more than 1 month) not being run. So theoretically, the limit a computer can analyze is the pair of degree 16/17, whose period is 4294934528. The amount of necessary intercepted bits is 65 or so. The computational complexity is much less than other cryptanalysis methodologies.

I would rewrite the programs with more-bottom-reaching languages, if time permits, because those languages can directly use memory without any language translating, thus more time-saving.

On the other hand, quantum technology is also one of my favorite topics. I believe that in the next ten years, this technology will be widely used in cryptosystems due to its high efficiency in encryption and decryption. I show a lot of interests in this technology and I would make an effort on quantum technology if there is any chance.

ACKNOWLEDGEMENT

I would like to thank Prof. A.Fuster and the staff in ITEFI for their helpful support and comments.

References

[1] Solomon W. Golomb. *Shift Register-Sequences*. Aegean Park Press, Laguna Hill, California, 1982.

on CA. *Journal of Cellular Automata*, Accepted, 2016.

[2] Sara D. Cardell and Amparo Fuster-Sabater. Recovering the MSS-sequence via CA. Procedia Computer Science, Volume 80, 2016, Pages 599–606, ICCS 2016. The International Conference on Computational Science

[3] Ali Kanso. Modified self-shrinking generator. Computers and Electrical Engineering 36 (2010) 993–1001

[4] Sara D. Cardell and Amparo Fuster-Sabater. Linear models for the self-shrinking generator based on CA. Journal of Cellular Automata, Accepted, 2016.

[5] Pino Caballero-Gil1, Amparo Fúster-Sabater2 and M. Eugenia Pazo-Robles3. New Attack Strategy for the Shrinking Generator. Journal of Research and Practice in Information Technology, Vol. 41, No. 2, May 2009

[6] CABALLERO-GIL, P. and FÚSTER-SABATER, A. (2006): Using Linear Hybrid Cellular Automata to Attack the Shrinking Generator, IEICE Transactions on Fundamentals of Electronics Communications and Computer, E89-A: 1166-1172.

[7] JOHANSSON, T. (1998): Complexity Correlation Attacks on Two Clock-Controlled Generators, *Lecture Notes in Computer Science* 1426, Springer-Verlag: 342-356.

[8] SHPARLINSKI, I. (2001): On Some Properties of the Shrinking Generator. Designs*, Codes and Cryptography* 23: 147-156.

[9] Golomb S. Shift register sequences. Laguna Hills, CA: Aegean Park Press; 1982.

[10] Willi Meier and Othmar Staffelbach. The self-shrinking generator. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 1994*, volume 950 of *Lecture Notes in Computer Science*, pages 205–214. Springer-Verlag, 1994.

Appendix A. MATLAB Source code

In this section, I present the simple example of shrinking-generator cryptanalysis over degree 4/5. This source code is written using MATLAB R2014a. The embedded functions are also included. Degrees and reciprocal polynomial p(x) should be changed manually. Also the PN sequence and the shrunken sequence should be generated before so that they could be loaded in this program.

CRYPTANALYSIS

function [runningtime]=TEST\_4\_5(~)

clear all

clc

% clock

tic

% Zech Logorithm

N2=31;

alpha0=[0,0,0,0,1];

alpha1=[0,0,0,1,0];

alpha2=[0,0,1,0,0];

alpha3=[0,1,0,0,0];

alpha4=[1,0,0,0,0]; %D^4+D^1+1

alpha5=[0,0,1,0,1];

alpha=cat(3,alpha0,alpha1,alpha2,alpha3,alpha4,alpha5); %

for al=6:N2

alpha(:,:,al)=xor(alpha(:,:,al-3),alpha(:,:,al-5)); % p(x) +1:-10

end

load Shrunken\_Sequence1000.mat; %load shrunken sequence

load Sequence1\_all1000.mat %load PN sequence of R1

num\_int=8; % numnber of intercepted bits

actual\_position=zeros(num\_int,1);

Sequence1\_all=Sequence1\_all';

N1=15;

state=1;

d=modula(N1,N2); %compute d

Correct\_state=cat(3); % Store correct initial states

correct\_state=[];

correct\_counter=1;

Times\_positions\_founded=zeros(N1+1,1);

% Generate Intercepted Bits set

Intercepted\_Bits1=Shrunken\_Sequence(1:num\_int);

for gen\_row=1:length(Intercepted\_Bits1)-1

for gen\_col=1:length( eval(strcat('Intercepted\_Bits',num2str(gen\_row))))-1 %coloum

eval(strcat('Intercepted\_Bits',num2str(gen\_row+1),'(gen\_col)=xor(Intercepted\_Bits',... % Bits

num2str(gen\_row),'(gen\_col),Intercepted\_Bits',num2str(gen\_row),'(gen\_col+1));'));

end

end

% Find positions of 1s in sequence1

nom\_ones=1;

pos\_original=zeros(num\_int,1);

counter\_ones=1;

while 1

if Sequence1\_all(counter\_ones)==1

actual\_position(nom\_ones)=counter\_ones-1;

pos\_original(nom\_ones)=counter\_ones-1;

nom\_ones=nom\_ones+1;

end

counter\_ones=counter\_ones+1;

if nom\_ones-1==num\_int

break;

end

end

counter\_ones=counter\_ones-1;

% Calculate position 1 (No Zech Logorithm)

pos1=zeros(num\_int,1);

for counter1=1:length(pos\_original)

pos1(counter1)=mod(pos\_original(counter1)\*d,N2);

end

% Zech Logorithm generating rest of positions

for row\_counter=1:num\_int

for col\_counter=1:length(eval(strcat('pos',num2str(row\_counter))))-1

if eval(strcat('pos',num2str(row\_counter),'(col\_counter)-pos',num2str(row\_counter),...

'(col\_counter+1)<0')) % increase

eval(strcat('pos',num2str(row\_counter),'(col\_counter)=pos',num2str(row\_counter),...

'(col\_counter)+N2;'));

end

eval(strcat('Zech\_counter=xor(alpha(:,:,(pos',num2str(row\_counter),...

'(col\_counter)-pos',num2str(row\_counter),'(col\_counter+1))+1),alpha(:,:,1));'))

for search\_counter=1:N2

if Zech\_counter==alpha(:,:,search\_counter)

break;

end

end

eval(strcat('zech=search\_counter-1+pos',num2str(row\_counter),'(col\_counter+1);')) % decrease the order

if zech>N2

zech=zech-N2;

end

eval(strcat('pos',num2str(row\_counter+1),'(col\_counter)=zech;'))

if eval(strcat('pos',num2str(row\_counter),'(col\_counter)-N2>=0')) % decrease

eval(strcat('pos',num2str(row\_counter),'(col\_counter)=pos',...

num2str(row\_counter),'(col\_counter)-N2;'))

end

end

end

% Assign Intercepted Bits and judge

fprintf('%d.\n',state);

tf=0;

Recovery=[];

CheckArray=zeros(N2+1,1);

for gen\_row=1:length(Intercepted\_Bits1)

for gen\_col=1:length( eval(strcat('Intercepted\_Bits',num2str(gen\_row))))

if eval(strcat('CheckArray(pos', num2str(gen\_row),'(gen\_col)+1)==1')) % compare

if eval(strcat('Recovery(pos',num2str(gen\_row),'(gen\_col)+1)~=Intercepted\_Bits',...

num2str(gen\_row),'(gen\_col);' ));

tf=1; % contradiction

fprintf('Wrong!\n');

fprintf('Contradiction found: line %d #%d, position: %d\n',gen\_row,...

gen\_col,eval(strcat('pos',num2str(gen\_row),'(gen\_col)')));

break;

end

end

eval(strcat('Recovery(pos',num2str(gen\_row),'(gen\_col)+1)=Intercepted\_Bits',... %

num2str(gen\_row),'(gen\_col);'));

eval(strcat('CheckArray(pos',num2str(gen\_row),'(gen\_col)+1)=1;' )); % check

end

if tf==1

break;

end

end

Recovery=Recovery';

if tf==0

Correct\_state(:,:,correct\_counter)=Sequence1\_all(pos\_original(1)+1:pos\_original(1)+log2(N1+1));

correct\_state(correct\_counter)=state;

correct\_counter=correct\_counter+1;

fprintf('Correct!\n');

fprintf('%d',Sequence1\_all(pos\_original(1)+1:pos\_original(1)+log2(N1+1)));

fprintf('\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n');

end

pos1=pos1';

%% Other States !!!!!!!!!!!!

while state<=(N1-1)/2

state=state+1;

fprintf('%d.\n',state);

states=zeros(1,log2(N1+1));

%% Shift line 1

minus\_num=pos1(2);

pos1=pos1-minus\_num;

for counter\_once=1:length(pos1) % get positions

if pos1(counter\_once)<0

pos1(counter\_once)=pos1(counter\_once)+N2;

end

end

pos1(1:end-1)=pos1(2:end); % shift 1 position,

%% select the Original Position of the last bit

pos\_original(1:end)=pos\_original(1:end)-pos\_original(2);

pos\_original(1:end-1)=pos\_original(2:end);

actual\_position(1:end-1)=actual\_position(2:end);

while 1

counter\_ones=counter\_ones+1;

if Sequence1\_all(counter\_ones)==1

pos\_original(end)=counter\_ones-actual\_position(1)-1;

break;

end

end

actual\_position(end)=counter\_ones-1;

i=1;

while 1

if pos\_original(i)>=log2(N1+1)

break;

end

states(pos\_original(i)+1)=1;

i=i+1;

end

%% shift the rest of positions

for row\_counter=2:num\_int-1

for col\_counter=1:length(eval(strcat('pos',num2str(row\_counter))))-1

eval(strcat('pos',num2str(row\_counter),'(col\_counter)=pos',num2str(row\_counter),...

'(col\_counter+1)-minus\_num;'))

if eval(strcat('pos',num2str(row\_counter),'(col\_counter)<0'))

eval(strcat('pos',num2str(row\_counter),'(col\_counter)=pos',num2str(row\_counter),...

'(col\_counter)+N2;'))

end

end

end

%% calculate the last position in pos1, and rest of poitions

pos1(end)=mod(pos\_original(end)\*d,N2);

for row\_counter=2:num\_int

eval(strcat('pos',num2str(row\_counter),'(end)=ZechLog\_4\_5(pos',...

num2str(row\_counter-1),'(end),pos',...

num2str(row\_counter-1),'(end-1));'))

end

%% Assign Intercepted Bits to new positions

tf=0;

Recovery=[];

CheckArray=zeros(N2+1,1);

for gen\_row=1:length(Intercepted\_Bits1)

for gen\_col=1:length( eval(strcat('Intercepted\_Bits',num2str(gen\_row))))

if eval(strcat('CheckArray(pos', num2str(gen\_row),'(gen\_col)+1)==1')) % compare

if eval(strcat('Recovery(pos',num2str(gen\_row),'(gen\_col)+1)~=Intercepted\_Bits',...

num2str(gen\_row),'(gen\_col);' ));

tf=1; % contradiction

Times\_positions\_founded(gen\_row)=Times\_positions\_founded(gen\_row)+1;

fprintf('%d',states);

fprintf('\nWrong!\n');

fprintf('Contradiction found: line %d #%d, position: %d\n',gen\_row,...

gen\_col,eval(strcat('pos',num2str(gen\_row),'(gen\_col)')));

break;

end

end

eval(strcat('Recovery(pos',num2str(gen\_row),'(gen\_col)+1)=Intercepted\_Bits',... %

num2str(gen\_row),'(gen\_col);'));

eval(strcat('CheckArray(pos',num2str(gen\_row),'(gen\_col)+1)=1;')); % check

end

if tf==1

break;

end

end

if tf==0

Correct\_state(:,:,correct\_counter)=states;

correct\_state(correct\_counter)=state;

correct\_counter=correct\_counter+1;

fprintf('Correct!\n');

fprintf('%d',states);

fprintf('\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n');

end

end %while

runningtime=toc

Correct\_state

correct\_state

Times\_positions\_founded

save workspace\_4\_5\_19

end

FUNCTION: Modula Computation of the Variable d

function y=modula(L1,L2)

for d=1:L2

if mod(L1\*d,L2)==1

break;

end

end

y=d;

end

FUNCTION: Zech Logarithm

function y=ZechLog\_4\_5(d1,d0)

T2=5;

N2=2^T2-1;

alpha0=[0,0,0,0,1];

alpha1=[0,0,0,1,0];

alpha2=[0,0,1,0,0];

alpha3=[0,1,0,0,0];

alpha4=[1,0,0,0,0]; %D^4+D^1+1

alpha5=[0,0,1,0,1];

alpha=cat(3,alpha0,alpha1,alpha2,alpha3,alpha4,alpha5); %

for al=6:N2

alpha(:,:,al)=xor(alpha(:,:,al-3),alpha(:,:,al-5)); % p(x) +1:-10

end

if d1-d0<0

d1=d1+N2;

end

k=xor(alpha(:,:,(d1-d0)+1),alpha(:,:,1)); % result of Zech

for m=1:N2

if k==alpha(:,:,m)

break;

end

end

zech=m-1+d0; % decrease the order

if zech>N2

zech=zech-N2;

end

y=zech;

Appendix B. MAPLE Source code

This is the source code for computing reciprocal polynomial p(x), which is important for computing Zech Logarithm. T is the degree of LFSR2 and P2(x) is the polynomial of LFSR2. T and P2(x) should be changed accordingly.

**> **

**> **

**> **

**> **

**> **

**> **

**> **

**> **

**> **