

ANALYSIS OF THE COMPRESSION TECHNIQUES AND MEMORY
BIT EFFECTS ON COMPRESSION FOR PSEUDO-RANDOM
GENERATION

by

Ümit Seren

B.S., Electrical-Electronic Engineering., Boğaziçi University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronic Engineering
Boğaziçi University
2007

ACKNOWLEDGEMENTS

Firstly, I would like to thank to my thesis supervisor Prof. Dr. Emin Anarım for his kind interest, technical advices and patience.

Additionally, I would like to thank to Kıvanç Mıhçak and İmran Ergüler for their kind interest to my questions and for discussions and also, many thanks to committee for their kindness of participation to my thesis.

The last but not least I would like to thank my parents. I always felt their support, courage, endless love, and belief on my success.

This work is supported in part by The State Planning Organization of Turkey, Project: Next-Generation Satellite Networks and Applications, DPT-03 K120250.

ABSTRACT

ANALYSIS OF THE COMPRESSION TECHNIQUES AND MEMORY BIT EFFECTS ON COMPRESSION FOR PSEUDO-RANDOM GENERATION

Compression of sequence is one of the most useful tools for strengthening the pseudo-random generators used in stream ciphers. Using compression components can prevent algebraic attacks against LFSR-based stream ciphers. Some of the proposed compression algorithms are bit-search generator (BSG), ABSG (modified version of BSG), Self-Shrinking Generator (SSG), Shrinking Generator (SG).

In this thesis, we analyze the compression of pseudo-random generation and determine the optimum compression algorithm among existing ones that has the optimal trade-off between output rate and resistance against general attacks.

We also aim to investigate the memory bit effect on compression for pseudo-random generation. We present the algorithm EBSG which is similar to ABSG. EBSG uses memory bit and after each output bit generation, it inserts the bit stored in memory to the input sequence. We show that EBSG increases output rate while providing good linear complexities and randomness.

ÖZET

SÖZDE-RASGELE ÜRETİMİN SIKIŞTIRILMASI TEKNİKLERİNİN İNCELENMESİ VE HAFIZA İKİLİNİN SÖZDE-RASGELE ÜRETİMİN SIKIŞTIRILMASI ÜZERİNE ETKİSİ

Dizilerin sıkıştırılması, dizitip şifreleyicilerde kullanılan sözde-rasgele üreteçlerin güçlendirilmesini sağlayan yararlı yöntemlerden biridir. Sıkıştırma elemanlarının kullanılması, Doğrusal Geri Beslemeli Kayan Saklaçların (LFSR) kullanıldığı dizitip şifreleyicilere karşı yapılacak cebirsel atakları önler. Bugüne kadar dizi sıkıştırılması konusunda sunulan bazı algoritmalar: İkili(bit) Arama Üreteci (BSG), Alternatif İkili Arama Üreteci (ABSG), Kendiliğinden Büzen Üreteç (SSG) ve Büzen Üreteçtir (SG).

Bu tezde, öncelikle sözde-rasgele dizi üretiminin sıkıştırılmasını çözümledik ve kullanılmakta olan sıkıştırma algoritmaları içinde en iyisini, çıktı oranı ve genel ataklara karşı direnç ödünleşimini en uygun seviyede sağlayacak şekilde belirledik.

Bir diğer amaçladığımız şey ise hafıza ikilinin sözde-rasgele dizi üretiminin sıkıştırılması üzerinde nasıl bir etkisi olduğunu incelemektir. Bu amaçla ABSG'ye benzeyen EBSG adlı yeni bir algoritma önerdik. EBSG algoritması her çıktı ikili üretiminden sonra girdi dizisine hafızada saklanan ikili eklemektedir. Göstereceğimiz üzere EBSG algoritması çıktı oranını artırırken, aynı zamanda uygun doğrusal karmaşayı ve rasgeleliği de sağlamıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	ix
LIST OF TABLES	x
LIST OF SYMBOLS / ABBREVIATIONS	xii
1. INTRODUCTION	1
1.1. Motivation	2
1.2. Cryptology	2
1.3. Symmetric Key Cipher	4
1.4. Cryptanalysis	6
1.5. Thesis Outline	8
2. INTRODUCTION TO STREAM CIPHERS	9
2.1. Stream ciphers	9
2.2. Linear Feedback Shift Registers	10
2.3. Boolean Functions	14
2.4. Some Basic Stream Cipher Designs	17
2.5. Cryptanalysis of Stream ciphers	19
3. COMPRESSION OF PSEUDO-RANDOM SEQUENCES	22
3.1. Introduction to Compression	22
3.2. Compression Model for Pseudo-random Generation	23
3.2.1. Prefix Codes and Binary Trees	24
3.2.2. General Framework	25

3.3. Requirements on C	26
3.4. Requirements on f	28
3.4.1. The Prefix code output case	29
3.4.2. The non-prefix output case	31
3.4.3. General Case	31
4. THE $\{0, 1\}$ - CASE	34
4.1. Parameters of the $\{0,1\}$ - Case	34
4.2. Security Analysis	35
4.2.1. Exhaustive reconstruction	35
4.2.2. Reconstruction based on the most probable case	36
4.2.3. Equations Retrieval	37
4.2.4. FBDD Attack	39
4.3. Optimal Choices	42
4.4. Some Popular Compression Algorithms	45
4.4.1. ABSG	45
4.4.2. BSG (Bit Search Generator)	48
4.4.3. SSG (Self-Shrinking Generator)	50
5. MEMORY BIT EFFECT ON COMPRESSION FOR PSEUDO - RANDOM GENERATION	52
5.1. Editing Bit Search Generator – EBSG	52
5.2. Security Analysis of EBSG	56
5.2.1. Exhaustive Reconstruction Attack Against EBSG	56
5.2.2. Most Probable Case Attack Against EBSG	60
5.2.3. Equations Retrieval Attack Against EBSG	64
5.2.4. FBDD Attack Against EBSG	66
5.3. EBSG algorithm inserting more than one bit	68

5.3.1. Exhaustive Reconstruction Attack Against EBSG(2) and EBSG(3)	71
5.3.2. Most Probable Case Attack Against EBSG(2) and EBSG(3)	71
5.3.3. Equations Retrieval Attack Against EBSG(2) and EBSG(3)	72
5.3.4. FBDD Attack against EBSG(2) and EBSG(3)	73
5.3.5. Results	74
5.4. Comparison with ABSG, BSG and SSG	75
6. CONCLUSION AND FUTURE WORK	79
APPENDIX A: PROOF OF PROPOSITION 3.7	80
APPENDIX B: PROOF OF PROPOSITION 4.4	82
REFERENCES	85

LIST OF FIGURES

Figure 1.1.	General structure of a block Cipher	4
Figure 1.2.	General structure of a stream cipher	5
Figure 2.1.	General form of an LFSR with length l	11
Figure 2.2.	Structure of a nonlinear combination generator	17
Figure 2.3.	Structure of a nonlinear filter generator	18
Figure 2.4.	Alternating step generator	19
Figure 3.1.	ABSG code tree	25
Figure 4.1.	LFSR based keystream generator	39
Figure 4.2.	BSG code tree	50
Figure 4.3.	SSG code tree	51
Figure 5.1.	How EBSG produce output	61
Figure 5.2.	The look up table	63

LIST OF TABLES

Table 4.1.	Characteristic of attack exponents against T_h^d trees	44
Table 4.2.	ABSG algorithm	45
Table 4.1.	BSG algorithm	49
Table 5.1.	EBSG algorithm	53
Table 5.2.	Statistical properties of some EBSG output sequences	59
Table 5.3.	Some keystreams and their most probable preimages	62
Table 5.4.	Some elements of the look up table for EBSG	63
Table 5.5.	Complexities of most probable case attacks for EBSG	64
Table 5.6.	Some $p_C(m)$ values for EBSG	67
Table 5.7.	EBSG(2) algorithm	68
Table 5.8.	EBSG(3) algorithm	69
Table 5.9.	Complexities of most probable case attacks for EBSG(2) and EBSG(3)	72
Table 5.10.	Some $p_C(m)$ values for EBSG(2) and EBSG(3)	74
Table 5.11.	General Attack complexities for EBSG, EBSG(2) and EBSG(3)	74
Table 5.12.	Complexities of Attacks for different compression algorithms	76

Table 5.13.	Ouput distribution for $p = 0.5$	76
Table 5.14.	Ouput distribution for $p = 0.1$	77
Table 5.15.	Ouput distribution for $p = 0.2$	77
Table 5.16.	Ouput distribution for $p = 0.3$	77
Table 5.17.	Ouput distribution for $p = 0.7$	77
Table 5.18.	Ouput distribution for $p = 0.8$	77
Table 5.19.	Ouput distribution for $p = 0.9$	78

LIST OF SYMBOLS / ABBREVIATIONS

C	Binary Code
C_y	The set of possible preimages of y
D_b^n	Number of preimages of b with length n
D_y^d	Number of preimages of sequence y with length d
E_b	The expected length on the preimage of b
$E(C)$	Mean length of words of C
E_Y	The expected length on the preimage of output sequences with length n
$f(C)$	Image set of C
H_b	The entropy on the preimage of b
H_b^{length}	The entropy on the length of the preimage of b
H_Y	The entropy on the preimage of output sequences with length n
H_Y^{length}	The entropy on the length of the preimage of output sequences with length n
L_n	Number of leaves at depth n
N_n	Number of nodes at depth n
$P(x y)$	Probability that x is preimage of output sequence y
$\text{Rate}(C, f)$	Output rate of (C, f)
s	Input sequence
x^d	Preimage of y of length d
$x^*(y)$	Most probable preimage of output sequence y
$\mathbb{F}_q$	A field defined on q elements
$\overline{\phi_b}$	Average number of retrieved linear equations for b
ϕ_b^n	Number of linear equations on the input bits, if the preimage of output bit b has length n
ϕ_y^d	Number of linear equations on the input bits, if the preimage of output sequence y has length d

$\overline{\phi_Y}$	Average number of retrieved linear equations for output output sequences with length n
BSG	Bit Search Generator
EBSG	Editing Bit Search Generator
SG	Shrinking Generator
SSG	Self-Shrinking Generator

1. INTRODUCTION

Up to now, we have seen many new proposals for stream ciphers. The main objective of these stream ciphers is to produce a random looking output sequences.

A Linear Feedback Shift Register (LFSR) with primitive feedback polynomial produce a sequences that have most of the properties which we expect from a random sequence, so we generally use LFSRs in design of stream ciphers. The problem may occur if we use only LFSR in stream cipher design, because any output bit of the LFSR can be written as a linear function in the initial state bits. We can overcome this problem with introducing some nonlinearity into the cipher. There are many ways to do this and generally boolean functions are preferred for introducing some nonlinearity into the cipher.

There is also another common way to introduce non-linearity is to compress the LFSR output sequence. There are some popular compression algorithms such that bit-search generator (BSG) [2,3], ABSG (updated version of BSG) [2], shrinking generator (SG) [4] and self-shrinking generator (SSG) [5].

In this thesis, we make analysis of compression techniques and investigate the effects of memory bit on compression for pseudo-random generation.

In the rest of this chapter, We make you an introduction of cryptology, symmetric-key systems and cryptanalysis methods. In the section 1.1 of this chapter, We briefly explain ideas which have motivated me to do this work. Section 1.2 makes a brief introduction to cryptology, section 1.3 gives symmetric-key ciphers and section 1.4 discusses cryptanalysis and attack methods. In section 1.5, you can see the outline of the thesis.

1.1. Motivation

Recently, there has been new proposals for stream ciphers that can be implemented easily and provide security for high data rate application. These new kind of ciphers uses the idea of compression.

Compression of pseudorandom sequences is interesting research area which focuses on strengthening the pseudo-random sequences.

Assume that input sequence s is the pseudo-random output of a public mechanism with secret parameters (e.g., the output of an LFSR initialized with a secret key and initialization vector). Our objective is to delete from s enough information to prevent such attacks that may apply to s , by hiding algebraic properties of the input sequence. Moreover, our output shouldn't be too short compared with its input, so we can use it for the same applications as s .

In this work, we aim to see the effect of memory bit on compression, so we propose a new compression algorithm Editing Bit Search Generator-EBSG which inserts a bit to the input sequence s after each output generation. We expect that this algorithm provide higher output rate, high linear complexity and also good security level.

1.2. Cryptology

Cryptology is the uniting name for a broad scientific field in which one studies the mathematical techniques of designing, analysing and attacking information security services. Cryptology consists of two subfields; cryptography and cryptanalysis. Cryptography is the field in which one studies techniques for providing security services and cryptanalysis is the field in which one studies techniques for defeating(attacking) the security services [12].

In [13], four objectives for cryptographic services are presented:

- **Confidentiality:** The objective is to ensure that information can be accessed by only authorized users.
- **Data integrity:** The objective is to ensure that only authorised users can change the information without it being noticed.
- **Authentication:** The objective is to ensure origin or destination of data.
- **Non-Repudiation:** The objective is to ensure that someone can not deny a previous commitment or action.

These issues are mentioned in cryptography using different cryptographical primitives.

We can identify primitive as a fundamental tool or algorithm designed to solve a specific information security problem.

The classic information security objective, to provide secrecy, is obtained by using a primitive called a cipher. Some examples of other primitives are signing primitives which should provide the same functionality as a handwritten signature, or Message Authentication Codes which provide means to ensure that a message has not been changed on the route to the receiver.

The cryptographical primitives can be classified into three categories; unkeyed primitives, symmetric-key primitives, asymmetric-key primitives, where the latter are also known as public-key primitive

The unkeyed primitives include, for example, hash functions which can be used in databases and search algorithms. Keyed primitives are the symmetric-key and asymmetric-key primitives.

As mentioned before, keyed primitives are the symmetric-key and asymmetric-key primitives, we emphasize on the cipher primitive, as we prefer to use stream ciphers in applications of telecommunication.

The word symmetric comes from the fact that the same key is used for both encryption and decryption. A general property of symmetric-key encryption is that it should be easy to compute and derive decryption key from the encryption key and vice versa.

1.3. Symmetric Key Cipher

We can classify symmetric-key ciphers into two categories: block ciphers and stream ciphers.

Block cipher is a device that takes as an input L_{key} bit key and an n bit plaintext string and then generates an n bit ciphertext string as output. Plaintext is the message to be encrypted, ciphertext is obtained by encryption of plaintext and the key is the secret information used by both side of communication to have secret data transfer or communication. We call parameter n as block size.

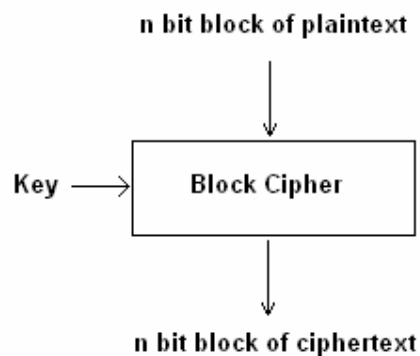


Figure 1.1. Block cipher

We should emphasize on the fact that block cipher have to define a permutation on the set of n bit binary strings. If the key is fixed and we encrypt all possible n bit messages, then we will get all possible n bit strings as output.

Famous block ciphers include the DES (Data Encryption Standard), and the newly standard AES (Advanced Encryption Standard) [13].

Stream cipher operates on each individual characters in the underlying alphabet, with a time varying function. We don't have the problem of patterns in the plaintext being encrypted to identical patterns in ciphertext, because encryption is time varying.

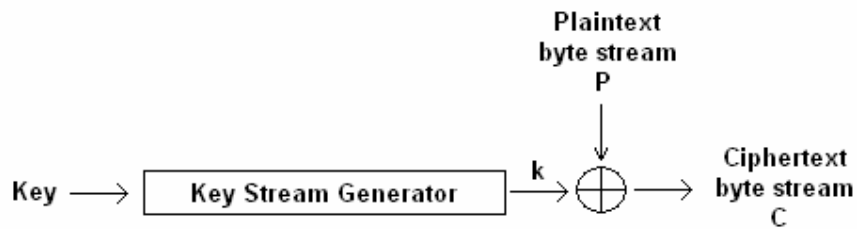


Figure 1.2. Stream cipher

Moreover, error propagation is limited because each character of plaintext is encrypted individually. Stream ciphers are easier to establish, also smaller and cheaper to build in hardware. We prefer stream ciphers due to properties mentioned above for telecommunication systems or high speed network devices to have maximum security.

1.4. Cryptanalysis

As mentioned before, cryptanalysis is the study of techniques for breaking a cryptographical primitive. When we evaluate the strength of cipher, we generally compare it to the generic attack of exhaustively searching over all possible keys to find the right one. We call this attack as exhaustive key search. In practice, no cipher can be more secure than the time it takes to test all possible keys, so this is accepted as the highest achievable strength of a cipher. On the other hand, there is one famous exception to this property, the ultimate secure cipher; the one-time pad (OTP). OTP is a vigenere cipher with a keylength equal to the length of the message. The important thing in OTP is that key bits must be completely random and used only once. It was shown that this cipher is unconditionally secure. This property tells us that attacker can never find out which plaintext was sent, no matter how big or fast computer the attacker has.

Disadvantage of OTP is that the key must be as long as the message that is to be encrypted. In fact, this condition is necessary for any cipher that claims unconditional security. We can say that as the key is secret, we should first send the key which is long as plaintext to the receiver through secure channel, before we send encrypted message. In practical sense, it is not efficient to use.

Moreover, for practical ciphers situation is much more different. We generally take into account *computationally secure ciphers*. This means that, among possibilities of today's computers and the increase predicted for tomorrow's computers, the attacker can not defeat the system.

Another concept is *computational security* of ciphers which is computationally effort required to capture initial state of cipher. One more definition to describe the security of the cipher is to try to prove that breaking the cipher is equivalent to solve a complex mathematical problem, like factoring integers or solving discrete log problem. This way of arguing can be defined as provable security. This is somewhat misleading since there is never proof of the complexity of the underlying problem. The notion is that these problems have been studied by the mathematicians for centuries and probably very difficult to solve.

The first and most important rule for the designer of a cryptographic primitive is called Kerckhoff's Principal:

The security of the encryption scheme must depend only on the secrecy of the key, and not on the secrecy of the algorithm.

Taking into consideration the Kerckhoff's Principal, we assume that attacker knows everything concerning the cryptographical system and the protocols used between sender and receiver. Let's categorize the methods of attacks according to amount of information which attacker has and the objective of attack.

- **Ciphertext-only attack:** All the information that attacker has is the ciphertext communicated between receiver and sender. Trying to decrypt the message given only ciphertext is the most difficult attack since the attacker has the least information.
- **Known plaintext attack:** In this type of attack, attacker knows both the plaintext and the corresponding ciphertext. It might seem slightly improbable that both the plaintext and the ciphertext are revealed, but there are many situations where this could happen.
- **Chosen plaintext attack:** Here, we assume that attacker has oracle which can encrypt any given plaintext under the correct key. This can be seen as more powerful attack than the known plaintext attack, because attacker can choose plaintext which are especially favourable for breaking the cipher.
- **Chosen ciphertext attack:** This attack is similar to chosen plaintext attack, but this time attacker has two oracle, one can encrypt any plaintext into cipher text and one that decrypts any given ciphertext except the one attacker trying to break. This attack is more powerful because attacker has more freedom.

The primary objective of the attacks mentioned above is to capture the plaintexts of set of given ciphertexts or to recover the initial keys. There is another attack type known as

distinguishing attack where attacker tries to detect a difference between the actual cipher and the ideal cipher, but this doesn't seem powerful.

1.5. Thesis Outline

The rest of the thesis is organized in the following fashion. Chapter 2 makes introduction to stream ciphers. In this part, the definition and properties of a stream cipher, linear feedback shift registers and Boolean functions are given. We also introduce some basic stream cipher designs. In Chapter 3, we give a comprehensive information about the compression for the pseudo-random generation. In Chapter 4, we focus on optimal case with $f(C) = \{0,1\}$. In this part, some popular compression algorithms and their resistances against general attacks are discussed. In Chapter 5, the new compression algorithm EBSG which uses memory bit is presented. We give general properties and security analysis of EBSG. Chapter 6 concludes the thesis.

2. INTRODUCTION TO STREAM CIPHERS

2.1. Stream ciphers

In cryptography, a stream cipher is defined as a symmetric cipher in which the plaintext digits are encrypted one at a time, and in which the transformation of successive digits varies during the encryption. An alternative name for this cipher is a state cipher, because the encryption of each digit is dependent on the current state. In practice, the digits are typically single bits or bytes.

Stream ciphers are divided into two categories: synchronous and self-synchronous.

A *synchronous stream cipher* is a finite state machine for which the keystream is generated from the key, but independently of the plaintext message and the ciphertext.

In a synchronous stream cipher a stream of pseudo-random digits is produced independently of the plaintext and ciphertext messages, and then processed with the plaintext (to encrypt) or the ciphertext (to decrypt). In the most general form, binary digits are used (bits), and the keystream is combined with the plaintext to produce ciphertext using the exclusive or operation (XOR). This is defined as a binary additive stream cipher.

Moreover, we can define the additive stream cipher as a pseudo-random number generator or a keystream generator whose output is xored to the plaintext. We use the key to initialize or seed the generator, which starts to produce pseudo-random bits. Here, we see the similarity with the OTP; instead of having the complete keystream as the secret key, we use smaller key for seeding the generator. Then generator starts to produce a keystream which is a random looking as possible.

Synchronous stream cipher is not particularly vulnerable to errors in the transmission, since each symbol is encrypted independently. Attacker can change a particular ciphertext symbol and cause the corresponding plaintext error, however all other symbols will remain correct. For a receiver, there is no direct possibility of validating the received message, so additional mechanisms for message authentication are needed to defend against these kinds of attacks.

However, if the attacker deletes or inserts a symbol into a cipher text -or if that happens by accident- the sender and receiver lose the synchronisation and all plaintext symbols following the deleted one will become erroneous during decryption. To prevent this problem, perfect synchronisation between sender and receiver needs to be ensured and techniques for detecting loss of synchronisation employed.

In a synchronous stream cipher, the sender and receiver must be exactly in step for decryption to be successful. If any condition causes the digits to be added or removed from the message during transmission, then synchronisation is lost. To restore synchronisation, various offsets can be tried systematically to obtain the correct decryption. Another approach is to tag the ciphertext with markers at regular points in the output.

A *self-synchronizing stream cipher* is a finite state machine for which the keystream is generated as a function of key and a fixed number of the previous ciphertext symbols.

These ciphers have the property that they are able resynchronise after a finite number of received ciphertext symbols. Thus the state of a such cipher is only dependent on the previous generated keystream bits or symbols. If some ciphertext symbols are deleted or inserted during transmission, the self-synchronizing cipher will recover after some correct ciphertext symbols, whereas the synchronous ciphers will never regain synchronization.

We introduce you the two main parts generally found in stream ciphers:

i. Linear Feedback Shift Registers

ii. Boolean Functions.

2.2. Linear Feedback Shift Registers

The main functionality of keystream generator is to produce random looking symbols. The main requirements for distribution of symbols are to be uniform and unpredictable. To use LFSR can be a good start for achieving the requirements mentioned above. In fact, direct output of LFSR is not a good keystream generator, because each

symbol produced is simply a linear combination of the previous symbols, and it is very easy to predict. Nevertheless, LFSRs are generally used inside stream ciphers.

An LFSR is a device which is made up of registers, able to hold one symbol at a time. The symbols are elements from a field F_q , over which we have chosen to define LFSR. Generally we have $q = 2$ (binary field) or some extension field of the binary field where $q = 2^W$, where W is a symbol of the stream cipher. Implementation of LFSR in hardware and software is really easy. In LFSR, there is a system clock which is responsible for the timing of the events. In the figure 2.1, you can see a general LFSR, where the circles denote multiplication with the constant c_i and $\oplus$ denotes field addition operation.

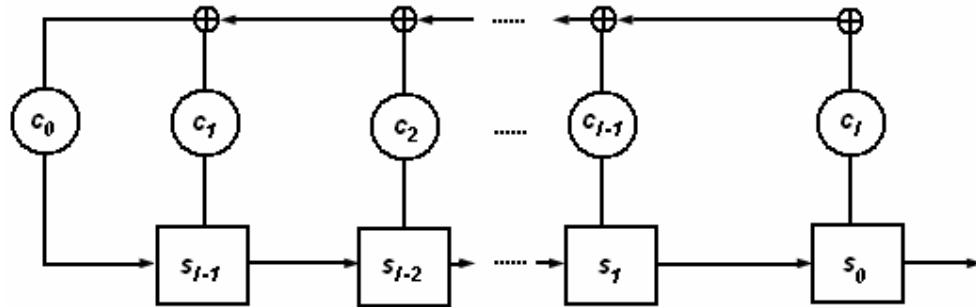


Figure 2.1. General form of a Linear Feedback Shift Register (LFSR)

At each clocking of LFSR, the registers read a new symbol from their respective input, and as the registers are coupled in series, the symbols move forward at each clocking. The first register receives a new symbol which is a linear combination of the symbols found in the registers after the previous clocking. The exact linear combination used for producing the feedback symbols is determined by the feedback coefficients $c_0, c_1, c_2, \dots, c_l$ shown in figure 2.1. Since we need the actual feedback connection c_0 to get any symbols into register, one normally assumes $c_0 = 1$. As we do not need more registers than necessary to make the feedback connection to work, we also assume $c_l \neq 0$ and define the length of the LFSR to be l . At each time $t \geq 0$ device is clocked, and we obtain a new

symbol $s_t \in \mathbb{F}_q$ at the output of the device. Due to linear feedback, the symbols s_t will always fulfill the linear recurrence equation:

$$c_0^{-1}s_{t+1} - c_1s_{t+l-1} - c_2s_{t+l-2} - \dots - c_ls_t = 0, \quad t \geq 0. \quad (2.1)$$

If the field characteristic is 2, and we assume $c_0 = 1$, we get the simpler form

$$\sum_{j=0}^l c_j s_{t+l-j} = 0, \quad t \geq 0. \quad (2.2)$$

Using the feedback coefficients, we can give a compact description of an LFSR through its *feedback polynomial*.

Definition 2.1: The feedback polynomial g is the polynomial $g(x) = c_0 - c_1x - c_2x^2 - \dots - c_lx^l$. The feedback polynomial is an element in the ring of polynomials $\mathbb{F}_q$, with x being the indeterminant.

The connection at c_0 is called the feedback position and each $c_j \neq 0$, $1 \leq j \leq l$ is called a tap position. An alternative description is the characteristic polynomial $f(x)$, which is the reciprocal polynomial of $g(x)$, $f(x) = c_l - c_{l-1}x - \dots - c_1x^{l-1} + c_0x^l$.

The content of the register at time t is $S_t = (s_{t+l-1}, s_{t+l-2}, \dots, s_t)$ and is called the *state* of the LFSR at time t .

The first l symbols to be produced, $S_0 = (s_{l-1}, s_{l-2}, \dots, s_0)$, are loaded into registers at the start, and we denote this state the initial state or the starting state of LFSR.

Since there are only a finite number of possible states q^l , the sequence produced by the LFSR must repeat itself after a finite period, i.e. for every starting state we can find a T such that $s_t = s_{t+T}$, $t \geq 0$. The period depends on properties of the feedback polynomial.

Definition 2.2: A polynomial $g(x) \in \mathbb{F}_q[x]$ is said to be irreducible over $\mathbb{F}_q$ if it can not be factored into polynomials of smaller positive degrees in the ring of polynomials $\mathbb{F}_q[x]$ [13].

Definition 2.3: An irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree l is said to be primitive if the root of $g(x)$ in the splitting $\mathbb{F}_{q^l}$ is a generator of the multiplicative group $\mathbb{F}_{q^l}^*$ [13].

Theorem 2.1: The period T of an LFSR with a primitive feedback polynomial $g(x) \in \mathbb{F}_q[x]$ of degree l and with a non-zero starting state is $T = q^l - 1$, [13].

Using the theorem and definitions mentioned above, we see that if we start with a non-zero state in the LFSR and use a primitive feedback polynomial, all possible states except the all-zero state will appear during a period. Maximum-length LFSR is LFSR with a primitive feedback polynomial. We call the sequence which is generated by maximum-length LFSR as a maximum-length sequence. We shouldn't forget that the initial state must be non-zero for the sequence to be maximum length, and hereafter it is assumed that the starting state is as such.

Definition 2.4: The linear complexity of a sequence $s = s_0, s_1, \dots, s_i \in \mathbb{F}_q$, denoted $L(s)$, is the length of the shortest LFSR, defined over $\mathbb{F}_q$, that generates the sequence. If the sequence is the all-zero sequence, then the linear complexity is 0, and if no LFSR can generate the sequence, then $L(s) = \infty$.

We can determine the linear complexity with the Berlekamp-Massey algorithm [14] that efficiently computes the feedback polynomial of the LFSR given at least $2L(s)$ of output symbols.

LFSRs generate sequences which have good statistical properties, desirable for keystream generator construction, but we need to increase non-linearity and decrease linearity, then sequence will be suitable to use. Classical method to obtain the objectives mentioned above can be accomplished by to use several LFSR and then using their output in a Boolean function as shown in the figure 2.2.

2.3. Boolean Functions

A Boolean function is an important element of the stream cipher that maps one or more binary input variables to one binary output variable. We can write this as a mapping from a vector of binaries $x = (x_1, x_2, \dots, x_n)$, where $x_i \in \mathbb{F}_2$, $1 \leq i \leq n$ to a single output bit

$$f: \mathbb{F}_2^n \rightarrow \mathbb{F}_2.$$

For n input variables there are 2^{2^n} distinct Boolean functions and we denote the set of Boolean functions in n variables by B_n . There are some ways to represent a Boolean function. If the number of input variable is small, we can construct truth table, where we list all possible input vectors with the corresponding output value. In table 2.1, you can see a truth table for particular Boolean function of three variables.

Table 2.1. The truth table of Boolean function $f(x_1, x_2, x_3) = x_1x_2 + x_2x_3 + x_3$

x_1	x_2	x_3	$f(x)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If the number of input variables are high, it is not possible to list all possible vectors, so we have to resort to a more compact description, for example *algebraic normal form* (ANF) .

$$f = \sum_{u \in F_2^n} \lambda_u \left(\prod_{i=1}^n x_i^{u_i} \right), \lambda_u \in F_2, \mathbf{u} = (u_1, u_2, \dots, u_n). \quad (2.3)$$

We call product of m distinct variables as an m th order product of the variables. To simplify Eq. (2.3), we can write every Boolean function as a (modulus 2) sum of distinct m th order product of the variables, $0 \leq m \leq n$.

The number of ones in the vector is defined as the Hamming weight of a binary vector. We call algebraic degree of function f , which is shown as $\deg(f)$, as the maximum value in Eq. (2.3) of the Hamming weight of $\mathbf{u}$ such that $\lambda_u \neq 0$, or simply the highest order of the terms in the ANF.

There are some other good and beneficial properties of Boolean functions for cryptographical applications. First, let's make a definition. A Boolean function is said to be balanced if it has same number of zeros and ones in the output column of the truth table. We can also say that if the Boolean function is balanced then $P(f(x) = 0) = P(f(x) = 1) = 1/2$ when you choose x uniformly from F_2 .

Definition 2.5: Functions of degree of at most one are called affine. The set of all affine functions in n variables is denoted as A_n . We can write

$$A_n = \{ a_0 + a_1x_1 + a_2x_2 + \dots + a_nx_n : a_i \in F_2, 0 \leq i \leq n \}. \quad (2.4)$$

We define linear function as an affine function with constant term $a_0 = 0$ and the set of linear functions in n variables is a subset of A_n .

Hamming distance between two function for example $f(x)$ and $g(x) \in B_n$ is the number of entries in the truth table output column where $f(x)$ and $g(x)$ differ.

Definition 2.6: The Non-Linearity of a Boolean function $f \in B_n$, denoted N_f , is the Hamming distance to the nearest affine function in A_n .

High nonlinearity for Boolean function is desired objective because it decreases the correlation between the output and the input variables or a linear combination of input variables. Many of the attacks against stream cipher succeed with the help of weakness of such a correlation between the combining Boolean function and some affine function.

There is one more important measure of Boolean functions named as correlation immunity. In [15], it was defined as in the following definition.

Definition 2.7: Let $X_1, X_2, \dots, X_n$ be independent binary random variables, each taking the values 0 and 1 with probability 1/2. A Boolean function $f(x_1, x_2, \dots, x_n)$ is said to be t -th order correlation immune, if for each subset of t variables $X_{i_1}, X_{i_2}, \dots, X_{i_t}$ with $1 \leq i_1 \leq i_2 \leq \dots \leq i_t \leq n$, the random variable $Z = f(X_1, X_2, \dots, X_n)$ is statistically independent of the random vector $(X_{i_1}, X_{i_2}, \dots, X_{i_t})$.

A Boolean function is t -resilient, if it is balanced and t -th order correlation immune.

It was shown in [15] that there is tradeoff between the algebraic degree and the order of correlation immunity.

Theorem 2.2: Let $f(x)$ be a balanced Boolean function in n variables of algebraic degree d which is t -th order correlation immune. Then the following upper bound must hold that

$$d + t \leq n - 1 \quad \text{if } 1 \leq t \leq n - 2 \quad (2.5)$$

$$d + t \leq n \quad \text{if } t = n - 1. \quad (2.6)$$

As a result, we have some observations about Boolean functions mentioned in [13]:

- High algebraic degree is desirable because it increases the linear complexity of the resulting keystream.
- High nonlinearity gives a weaker correlation between the input variable and the output variable. It also increases the resistance against the correlation attacks.
- High correlation immunity forces attacker to consider several input variables jointly and thus the vulnerability of divide-and-conquer attacks.

2.4. Some Basic Stream Cipher Designs

Stream ciphers based on LFSRs are generally divided into groups of three:

- Nonlinear combination generator
- Nonlinear filter generator
- Clock controlled generator

The nonlinear combination generator generally consists of n LFSRs and their outputs are combined in a Boolean function f . The output of the Boolean function becomes keystream output. These type of stream ciphers should have high algebraic degree, high nonlinearity and high order of correlation immunity. In order to have these properties large number of LFSRs should be used.

The nonlinear filter generator has different design principle. There is one LFSR and its different elements are used as an input of the Boolean function. In this stream cipher type, we call Boolean function as filtering function. We don't need to use all elements of LFSR as an input for Boolean function.

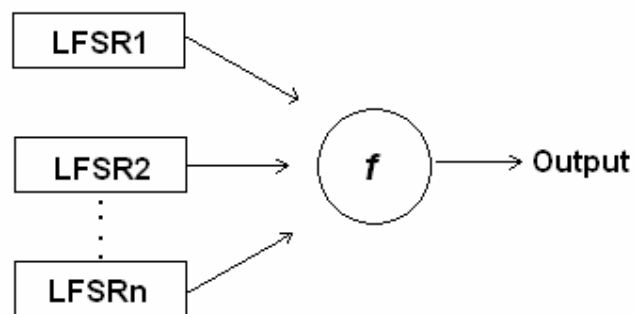


Figure 2.2. Nonlinear Combination Generator

In [16], it was shown that the upper bound on the linear complexity $L(s)$ for a nonlinear filter generator is $L(s) \leq \sum_{i=1}^d \binom{l}{i}$, where d is the algebraic degree of the function, l is length of LFSR and s is the desired keystream.

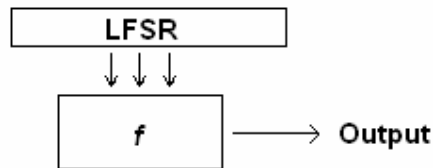


Figure 2.3. Nonlinear Filter Generator

As we told above, we have only considered pure Boolean function as the combiner/filter for non-linear combination generator and nonlinear filter generator, however we can use improved version of boolean function that is time varying . We call this type of generators as Finite State Machine (FSM).

The main property of FSM is that part of state space is updated via a nonlinear process. For FSM type generators the properties of pure Boolean functions are not directly applicable, especially the trade of between the algebraic degree and correlation immunity does not hold.

Another point can be to use key dependent S-boxes [17] in the combining/filtering function. Using S-boxes in Boolean function increases the complexity of cryptanalysis when compared with using pure Boolean function, also the cipher is not completely specified until the key is known.

The last category of the stream ciphers is the clock-controlled generators. These type of generators use LFSRs that are not clocked regularly. The idea in clock-controlled generator is that it uses some irregular signal to control the number of clockings of the LFSRs. This controlling signal can be output of any LFSR or some variables of cipher. The

linearity of the output of the irregularly controlled LFSR is then destroyed and attacks that uses weakness because of regular clocking is hard to apply.

Alternating step generator [17] is one of the classical clock-controlled generators. Its working procedure is :

- Register R1 is clocked by system clock.
- If the output of R1 is one then R2 is clocked, R3 is not clocked and its previous output bit is used
- If the output of R1 is zero then R3 is clocked, R2 is not clocked and its previous output bit is used.
- Sum of outputs from R2 and R3 is taken as the keystream output.

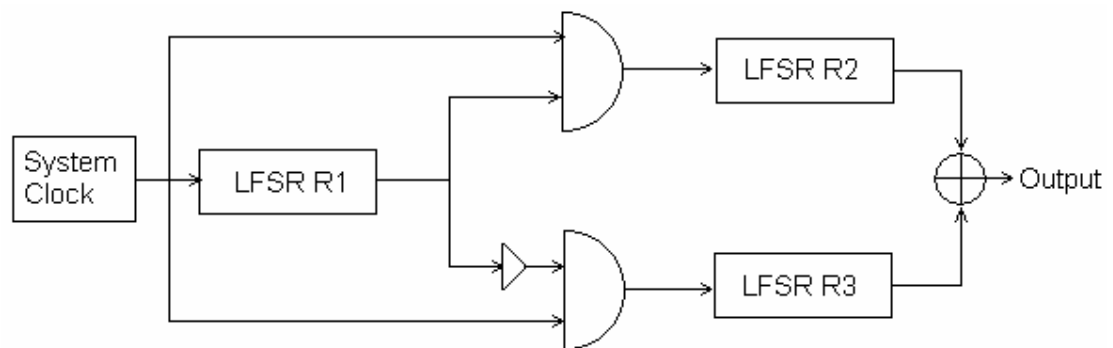


Figure 2.4. Alternating Step Generator

2.5. Cryptanalysis of Stream ciphers

When applying attacks against stream ciphers, there are some measurements which we have to take into account to compare the complexity of different attacks.

- **Time Complexity:** Can be defined as required number of operations needs to be processed to apply attack and to reach success. The operations we mentioned can be either low-level operations as “CPU instructions” or “cipher clockings” or

may be high level operations such as “table lookups” or “solving matrix equation”. Generally, time measure or complexity of attack is exponential in the length of the LFSR or in the key length.

- **Data complexity:** It can be defined as the required amount of keystream material which is needed to guarantee the success of attack.
- **Memory Complexity:** It can be defined as the required memory to apply attack. It is just like a combination of both the time and data complexity. Sometimes we need precomputational steps in attacks and we need to store these precomputed values. Moreover, we may need to build a search tree in memory based on the observed keystream. For some other attacks we need the complete observed keystream for random access, so memory complexity becomes equal to data complexity.

Some attacks against stream ciphers can be explained as below:

i. Tradeoff attacks: This attack is based on tradeoff for the time, memory, and data complexity. Attacker adjusts these complexities to obtain the lowest level of complexity.

ii. Guess and Determine Attacks : In Guess and Determine Attacks [18], you first guess some internal variables of the cipher, then you start to determine the other values of cipher depending on your guesses. If your guess is true, you can confirm it by generating some output sequence with your trial generator and then try to match these outputs with the observed sequence.

iii. Correlation attacks: In correlation attack [15], attacker aims to find a correlation between some linear combination of input variables to the combining function and the output from the combining function, and then use this correlation to obtain information about the correlated input variables. Simply, correlation means that one input variables has probability greater or less than $1/2$ to appear at the output. We can write correlation probabilities as $1/2+\epsilon$ for any input variable, where ϵ is defined as correlation between variables.

iv. Algebraic Attacks: The algebraic attacks on stream ciphers have been introduced by Courtois [19], Meier [20] and Armknecht [21]. Algebraic attacks on stream ciphers recover the key by solving an overdefined system of multivariate equations. Algebraic attack is newly found technique which aims to define a cipher system by a input–output relation of some degree d . If attacker can find a relation for some degree d , then attacker can theoretically recover the initial state of the cipher by solving multivariate equation by some or huge number of consecutive keystream bits. We also assume that the algorithm of cipher and also some keystream bits are known by attacker. Algebraic attacks are mostly applied against regularly clocked stream ciphers, non-linear filters and nonlinear combiners with or without memory. The steps generally followed in algebraic attacks are:

- We set up a system of equations which is obtained from the input-output relation of the stream cipher.
- We substitute the observed keystream bits into the equations derived in the first step.
- We recover the unknowns by solving the resulting system of equations using re-linearization [23], XL algorithm [22] or any other useful algorithm depending on attacker's choice.

3. COMPRESSION OF PSEUDO-RANDOM SEQUENCES

3.1. Introduction to Compression

Compression of sequences is one of the most promising tools for strengthening pseudo-random generators used in stream ciphers. Indeed, adding compression mechanism can provide a resistance against algebraic attacks aimed at LFSR-based stream ciphers.

We have some compression mechanisms which are still used for filtering LFSRs. The most popular ones are Bit Search Generator (BSG) [2], ABSG, MBSG (which are modified versions of BSG) [2], Shrinking Generator [4] and the Self-Shrinking Generator [5].

These compression mechanism are in fact weak against timing attacks [6]. As you know, speed of the output is strongly related with generator's state, so LFSR based stream ciphers using compression can easily be breakable, if attacker knows the number of times LFSRs are clocked for each output. This weakness can easily be overcome by buffering the output, but this work is not included in this thesis.

Generation of pseudo random generators using compression mechanism is based on the use of compression function. A *compression function* can be identified as a function that compresses any m -bit inputs to n output bits, be sure that these m and n are not fixed and can take different values, but they should always ensure the property that $m \geq n$.

Decimation based functions generally compress one source to obtain second source as an output that has better security quality than our first source (such as BSG, ABSG, MBSG, Self-Shrinking Generators), on the other hand Shrinking Generator uses two different sources to obtain the third source as an output.

Moreover, *output rate* is an important factor for compression. This term is generally used to compare the efficiency of compression systems.

3.2. Compression Model for Pseudo-random Generation

Generally people expect that data compression should compress input sequence into shortest length output sequence while keeping the ability to recover input sequence from the output sequence, which also means that we don't lose any information on input sequence.

Compression technique that we should follow in strengthening the sequence by pseudo-random generator is different. We use the compressed output as keystream in stream ciphers used to encrypt the message. Let's say that input sequence of the compression mechanism is s , which is generated by pseudo-random generators (for example output of LFSR). If we directly use this sequence s without any compression, this may cause weaknesses to attacks that uses correlation or algebraic properties of the sequences. We aim to delete some information from input sequence s to hide algebraic or correlation properties of the sequence, which prevents success of attacks on the sequence s . Another point that we should take into consideration is that output sequence of the compression should not be too small compared with input sequence s , because we should be able to use the output sequence for the same applications of input sequence.

All in all, we can say that our procedure for compression of pseudo-random sequences follows different way than used for usual data compression. Our procedure should guarantee that it should deliver the least information as possible, when converting input sequence to output sequence, it also should keep the output sequence as long as possible.

In this work, we call random input sequences are those sequences that are created by concatenation of binary words while providing uniform distribution. Probability of any binary word w is a prefix of any input sequence is $1/2^{|w|}$, while binary words are assumed to be independent.

3.2.1. Prefix Codes and Binary Trees

A binary code is a subset of words of $\{0,1\}^*$. The language C^* of a binary code C is the set of all binary words that are concatenation of words in C . A code C is a prefix code if no codeword has a strict prefix in C . In this case, the words of C^* parse into codewords in a unique manner.

A code is a maximal prefix, when no other prefix code contains it.

A code C is right complete if every word w can be completed into a word $v = ww'$ that belongs to C^* or if every word w with no prefix in C has a multiple $v = ww'$ in C .

Proposition 3.1: A code is maximal prefix if, and only if, it is prefix and right complete.

Proof: Assume that C is maximal prefix and w is a non-empty word which has no prefix in C . If C is a maximal prefix, $C \cup \{w\}$ is not a prefix code, then w has a right multiple in C . So C is right complete.

On the other hand, assume C is prefix and right complete, and C' is a prefix code that contains C . Let $w \in C'$. As C is right complete, w has a right multiple w' in C^* . Assume that m is the smallest prefix of w' in C . As C' is prefix, this implies that $m = w$, so we have $w \in C$ and so $C' = C$. Therefore C is maximal prefix. $\square$

In this thesis, we see that all suitable codes for our constructions are maximal prefix codes. There is a relation between binary prefix codes and binary trees called coding trees, in which a node is either leaf, or it has two children. We see that any link in binary tree ends with leaf can be seen as binary code word. An example of code tree for ABSG can be seen in figure 3.1.

It is clear that, not all binary codes are valid for this framework. For example, if you choose $C = \{00\}$, it does not enable us to process sequence that contains ones. There are also some requirements for the compression function f . In order to apply the framework to every possible input sequence, we have to define some key requirements for C and f .

These are:

- i. The choice of C should enable parsing of every random input sequence.
- ii. The choice of f should enable uniform distribution of output sequence for uniformly distributed input sequences.

3.3. Requirements on C

First of all, we only work with prefix codes. So, if any word w' has a prefix w in C , then w' should never appear in C . Therefore, we have to delete all codewords from C that has already prefix in C . This procedure transforms C into a binary prefix code.

We also want to make all possible random input sequences are processable, so it is clear that C have to be right complete. Moreover, we can use the following definition in order to define suitable binary prefix code for the framework.

Definition 3.2: A binary code C is suitable if it is prefix and if the expected length $E(C)$ of an element of C in the decomposition of a random input sequence is finite.

Proposition 3.2: For a suitable code C , the following equality holds:

$$\sum_{w \in C} \frac{1}{2^{|w|}} = 1.$$

Proof: Consider the binary tree that belongs to C . We define two terms L_n and N_n that represent the number of leaves and number of nodes of depth n . We have $L_0 = 0$ and $N_0 = 1$.

We have the relation that is $L_n + N_n = 2 N_{n-1}$ for $n \geq 1$.

$$A_n = \sum_{0 \leq k \leq n} \frac{L_n}{2^n}, \quad (3.1)$$

$$\frac{L_n}{2^n} = \frac{N_{n-1}}{2^{n-1}} - \frac{N_n}{2^n}, \quad (3.2)$$

$$\sum_{0 \leq k \leq n} \frac{L_n}{2^n} = \sum_{0 \leq k \leq n} \left(\frac{N_{n-1}}{2^{n-1}} - \frac{N_n}{2^n} \right), \quad (3.3)$$

using Eq. (3.2) and Eq. (3.3), we obtain Eq. (3.4)

$$A_n = N_0 - \frac{N_n}{2^n} = 1 - \frac{N_n}{2^n}, \quad (3.4)$$

so if we prove $N_n = O(2^n)$, it will be enough. N_n is the number of the nodes of depth n and any random input sequence begins with n bits corresponding to that node with

probability $\frac{N_n}{2^n}$. For each of that node, the first word of the random input sequence has

length at least n , so it contributes at least $n \frac{N_n}{2^n}$ to $E(C)$. We know that $E(C)$ have to be

finite, so if n goes to ∞ , $\frac{N_n}{2^n}$ should go to 0, so it is clear that the proposition 3.2 is true. $\square$

Moreover, for any suitable code, $E(C)$ is equal to the mean length of words of C , for the uniform distribution of the alphabet $\{0,1\}$, so we have proposition:

Proposition 3.3: Assume that C is a suitable code. Then, we have

$$E(C) = \sum_{w \in C} \frac{|w|}{2^{|w|}} \quad (3.5)$$

3. 4. Requirements on f

There are several requirements on $f(C)$ like C . First of all, $f(C)$ can be any set of words including empty word ε . Moreover, there should be at least two non-empty words starting with 1 and 0, to make random output looks uniformly distributed for any random input.

Moreover, it is clear that we should be able to construct any binary sequence with the elements of $f(C)$.

We have definition for $f(C)$ like definition 3.2.

Definition 3.3: Suppose that C is suitable code and we have a compression function $f: C \rightarrow \{0,1\}^*$, then we can say that (C, f) is valid encoder if the expected length $E(f(C))$ of the image of the C by the function f is finite and nonzero for any random input.

Proposition 3.4: For a proper encoder (C, f) , the expected length $E(f(C))$ of the image of C by the function f can be found by the following equation:

$$E(f(C)) = \sum_{w \in C} \frac{|f(w)|}{2^{|w|}}. \quad (3.6)$$

By using definition 3.2 and definition 3.3 we conclude that both $E(C)$ and $E(f(C))$ are finite and nonzero, then we obtain a new proposition that:

Proposition 3.5: The output rate of the proper encoder (C, f) is

$$\text{Rate}(C, f) = \frac{E(f(C))}{E(C)} \quad (3.7)$$

Now, we are going to choose the optimum set for the output $f(C)$ against the reconstruction of the input sequence easily. For each word in the output set, there are candidates which are defined as preimages of that word in C , so if we know the output, it

reduces the possible choices of preimages to one particular set. Now, we show that in order to minimize the information rate to reconstruct input sequence from the output sequence, we have to minimize the set $f(C)$.

Firstly, in order to satisfy the uniform distribution in the output sequence, each bit 0 and 1 have to appear in the bits of the output sequence with equal probability.

Therefore, for every $n \geq 1$:

$$\sum_{w \in C, |f(w)| \geq n, f(w)_n = 0} \frac{1}{2^{|w|}} = \sum_{w \in C, |f(w)| \geq n, f(w)_n = 1} \frac{1}{2^{|w|}} \quad (3.8)$$

where $f(w)_n$ is the n -th bit of the word $f(w)$.

3.4.1. The Prefix code output case

Firstly, consider the case where the set $f(C)$ is a prefix code. If it contains two elements, the optimum choice will be $f(C) = \{0,1\}$. In this case, 0 and 1 must have probability $\frac{1}{2}$ to appear in the output sequence for a random input sequence.

Now, let's consider the situation that $f(C)$ has more than two elements. We prove that for any random input sequence, knowing the output sequence, we can obtain more information on the first element of C than in the case $f(C) = \{0,1\}$.

Proposition 3.6: Let (C, f) be a proper encoder and for $x \in f(C)$, let $P(x) = \sum_{w \in f^{-1}(x)} \frac{1}{2^{|w|}}$. Then, for a random input sequence s , each word of decomposition of s over C has average length $E(C)$, and it is known with an average entropy :

$$E(C) + \sum_{x \in f(C)} P(x) \log P(x). \quad (3.9)$$

Proof : Assume that C_x is the preimage of x , where $x \in f(C)$. The probability that first element of C recognized in a random input sequence is mapped by f to x is $P(x)$
 $= \sum_{w \in C_x} \frac{|w|}{2^{|w|}}$ and expected length of element in the preimage of x is $E(C_x) = \frac{1}{P(x)} \sum_{w \in C_x} \frac{|w|}{2^{|w|}}$.

Then we can calculate the entropy on the elements in C_x as shown below:

$$\begin{aligned} H(C_x) &= - \sum_{w \in C_x} \frac{1}{P(x)2^{|w|}} \log \frac{1}{P(x)2^{|w|}} = \sum_{w \in C_x} \frac{|w|}{P(x)2^{|w|}} (\log P(x) + |w|) \\ &= \frac{1}{P(x)} \sum_{w \in C_x} \frac{|w|}{2^{|w|}} + \frac{\log P(x)}{P(x)} \sum_{w \in C_x} \frac{1}{2^{|w|}} = E(C_x) + \log(P(x)) . \end{aligned} \quad (3.10)$$

The average number of retrieved bits is

$$\sum_{x \in f(C)} P(x) E(C_x) = E(C), \quad (3.11)$$

and we obtain average entropy as:

$$\sum_{x \in f(C)} P(x) (E(C_x) + \log P(x)) = E(C) + \sum_{x \in f(C)} P(x) \log P(x) , \quad (3.12)$$

with $\sum_{x \in f(C)} P(x) = 1$. □

For any given suitable code C , we can divide it into two equiprobable subsets.

(Where the probabilities of leaves in the tree is in the form $\frac{1}{2^n}$ with $n \geq 1$ and their sum is 1.)

1.) Thus, for any given suitable code, there is a relation $f: C \rightarrow \{0,1\}$, such that 1 and 0 have probability $1/2$ to appear in the output.

Therefore, in order to maximize the entropy for any given suitable code C , we should maximize the value $\sum_{x \in f(C)} P(x) \log P(x)$. We have to choose $\#(f(C)) = 2$ to maximize it, so

the optimal choice is $f(C) = \{0,1\}$ with 0 and 1 have probability 1/2 to be output for any random input sequence, because $\{11,10\}$ or $\{00, 01\}$ do not provide uniform distribution in the output, and also $\{11,00\}$ or $\{10,01\}$ only increase the size of data without making any improvement in the security.

3.4.2. The non-prefix output case

Let's consider the situation where $f(C)$ is not a prefix code and also does not contain empty word ε . We can see that, it is not possible to determine preimage of each word (w_i) of the output sequence with certainty. For statistical reason, $f(C)$ should have two words with starting 1 and 0, also to provide the non-prefix case there should be at least three elements in $f(C)$.

Proposition 3.7: Let (C,f) be a proper encoder such that $f(C)$ is a non-prefix code which does not contain empty word ε . Then, the average expected length of the first word of the decomposition of the input sequence over C is $E(C)$ if the input sequence is chosen uniformly distributed. This word is known with average entropy $E(C) + \Delta(C)$, with $\Delta(C) < -1$.

Proof: See the Appendix A. □

We see that optimal choice of the output set even if non-prefix output sets are taken in to consideration, is still $f(C) = \{0,1\}$ with 0 and 1 having equal probability to be output for a random input sequence.

3.4.3. General Case

We now suppose that ε can belong to the output set $f(C)$.

Proposition 3.8: Let's think the situation that ε is in the set of $f(C)$. Then, there exists a proper encoder (C', f') such that $f'(C')$ does not contain ε and that, for every binary sequence s , we have

$$Enc_{Cf}(s) = Enc_{C'f'}(s).$$

Moreover, define $P_\varepsilon = \sum_{w \in f^{-1}(\varepsilon)} \frac{1}{2^{|w|}}$, then we have

$$E(C') = \frac{1}{1 - P_\varepsilon} E(C) \text{ and } E(f'(C')) = \frac{1}{1 - P_\varepsilon} E(f(C)).$$

Proof : Let's define C_ε the set of preimages of ε , and $C_{\bar{\varepsilon}}$ the complement of C_ε in C . Let C' be the binary code defined by $C' = C_\varepsilon^* C_{\bar{\varepsilon}}$, the set of binary words that can parse into a sequence of words C_ε , followed by words of $C_{\bar{\varepsilon}}$. Let's consider that f' maps each element ww' of C , with $w \in C_\varepsilon^*$, and $w' \in C_{\bar{\varepsilon}}$, to $f(w')$. f' is well defined as the decomposition is unique.

For every input sequence s , the equality $Enc_{Cf}(s) = Enc_{C'f'}(s)$ is satisfied. We also have $f(C') = f(C) \setminus \{\varepsilon\}$, so image of f' does not contain ε .

Now, let's show (C', f') is a proper encoder. Firstly, C' should be prefix code because of unicity of decomposition over C . Then, as the length of ε is 0,

$$E(f'(C')) = \sum_{v \in C_\varepsilon, w \in C_{\bar{\varepsilon}}} \frac{|f(w)|}{2^{|w|+|v|}} = \sum_{n \geq 0} \left(\sum_{v \in C_\varepsilon} \frac{1}{2^{|v|}} \right)^n \times \sum_{w \in C_{\bar{\varepsilon}}} \frac{|f(w)|}{2^{|w|}} = \frac{E(f(C))}{1 - P_\varepsilon}.$$

As the two encoders (C, f) and (C', f') are equivalent, they have same output rate, which gives us the same relation between $E(C)$ and $E(C')$. Hence (C', f') is proper encoder. $\square$

Proposition 3.8 shows us that, we can assume that $f(C)$ does not contain ε , without loss of generality.

We have concluded that the optimal choice for $f(C)$ is $\{0,1\}$.

4. THE $\{0, 1\}$ - CASE

In section 3, we have concluded that optimum choice for the set of $f(C)$ is $\{0,1\}$. In this section, we focus on the optimal choice of the proper encoder with $f(C)=\{0,1\}$ where 0 and 1 are equal probable relatively to the uniform distribution over the input sequence.

Firstly, we give the results that arise from section 3 and we analyze the security of the framework against general attacks: exhaustive reconstruction, most probable case reconstruction, equation retrieval and FBDD attack.

4.1. Parameters of the $\{0,1\}$ - Case

First of all, let's define some parameters belongs to this part. C_0 and C_1 are the preimages of 0 and 1, means that the set of word that produce output bit 0 or 1 when they are given to encoder as an input. We also define $C_b^n = \{w \in C_b, |w| = n\}$ and $D_b^n = \#(C_b^n)$.

Proposition 4.1: Let (C, f) be a proper encoder with $f(C) = \{0,1\}$. Then for a random input sequence s , the average length and entropy of each word of the decomposition of s over C are respectively $E(C)$ and $E(C) - 1$

This result is obtained from proposition 3.6, when applied to the case $f(C) = \{0,1\}$ with 0 and 1 have probability 1/2 to appear in the output.

Proposition 4.2: Given a bit b of the output sequence, a word $w \in C_b$ is the preimage of b with probability $\frac{1}{2^{|w|-1}}$.

Proof: Each word w of C appears in the input sequence with probability $\frac{1}{2^{|w|}}$, and the probability w belongs to C_b is 1/2, which gives the result. $\square$

4.2. Security Analysis

In this section, we analyze the security levels provided by compression components. We also assume that input sequence is the output of maximal length LFSR. We also assume that input sequence is evenly distributed. (Independent identically distributed Bernoulli process with probability 1/2.)

4.2.1. Exhaustive reconstruction

Exhaustive reconstruction is simply constructing consecutive bits of the input sequence from the output sequence starting from a fixed point in the output sequence.

When a bit appears in the output sequence, the expected length E_b and entropy H_b on the preimage of b in the input sequence are respectively equal to

$$E_b = \sum_{w \in C_b} \frac{|w|}{2^{|w|-1}} \quad (4.1)$$

and

$$H_b = - \sum_{w \in C_b} \frac{1}{2^{|w|-1}} \log_2 \left(\frac{1}{2^{|w|-1}} \right). \quad (4.2)$$

Using the Eq. (4.1) and Eq. (4.2) we have concluded that

$$H_b = \sum_{w \in C_b} \frac{|w|-1}{2^{|w|-1}} = E_b - 1. \quad (4.3)$$

Therefore, an attacker can deduce E_b bits in the input, with the entropy of H_b .

Information theory tells us that to represent any of the sequence from pre-defined set, we need in average $H(x)$ bits where $P(x)$ denote the probability of particular sequence occurring from pre-defined set and $H(x) = \sum_x -P(x) \log P(x)$.

Suppose that the input sequence to the compression mechanism is the output of LFSR of length L with a public feedback polynomial and secret key as its initial state. If

$$E = \frac{E_0 + E_1}{2}, \text{ then it is possible to retrieve the complete state of LFSR with an attack}$$

of average complexity $O(2^{\frac{H}{E}L})$, requiring $O(\frac{L}{E})$ consecutive bits.

Moreover, if $E_0 \neq E_1$, then attack complexity decreases by seeking for a sequence where bits b mostly appear, with b such that $E_b < E_{\bar{b}}$. This let us to make an attack with less complexity but requiring more output bits. The objective of this attack is taking a window of consecutive bits in the keystream sequence where most bits are b . The difficulty when applying this attack is to determine the better trade-off between the lengths of the window and required number of bits b in this window in order to retrieve L equations involving consecutive bits of input sequence. Such an attack is described in [2] for BSG algorithm.

4.2.2. Reconstruction based on the most probable case

Another reconstruction attack is based on betting each time that preimage of output b is (one of) the most probable. For each output b , we set $l_b = \min\{|w|, w \in C_b\}$, and $C_b^{short} = \{w \in C_b, |w| = l_b\}$. In this attack, we can not choose the point which consecutive input bits will be effectively reconstructed.

For a bit b in the output, the preimage of b is $w \in C_b^{short}$, with probability $1 / 2^{l_b-1}$.

Suppose that input sequence is given by LFSR of length L and $(l_0 + l_1)/2 = l$. It is then possible to retrieve the complete state of LFSR with an attack of average complexity $O(2^{\frac{l-1}{l}L})$ and requires $O(\frac{L}{E} 2^{\frac{l-1}{l}L})$ output bits (enough for the bet to be successful). This attack has lower complexity in the case where $l_b < E_b$ than exhaustive reconstruction.

Like in exhaustive reconstruction, when $l_0 \neq l_l$ holds, the attack complexity can be reduced by seeking sequences where most bits are b , such that $l_b < l_{\bar{b}}$.

4.2.3. Equations Retrieval

Sometimes, when input sequence is given by a maximum-length LFSR, it is enough to obtain linear equations on bits that are not consecutive in the input sequence.

On the other hand, obtaining equations on bits that are apart in the input sequence is not necessarily easier, because of entropy created by compression mechanism on the length of the preimages of words in the output sequence. Retrieving bits that are apart means that we are able to control the length of gaps between the bits retrieved in the input sequence.

For a bit b in the output, the preimage of b has length n with probability $\frac{D_b^n}{2^{n-1}}$, where D_b^n is the number of preimages of b with length n . If preimage of b has length n , then we can derive a number ϕ_b^n of linear equations on the input bits satisfying

$$\max(0, n - (D_b^n - 1)) \leq \phi_b^n \leq n - \lceil \log D_b^n \rceil. \quad (4.4)$$

We can see that at least $n+1 - D_b^n$ equations with probability $\frac{D_b^n}{2^{n-1}}$ can be obtained.

Then it is clear that, for a bit b in the output, the average number of retrieved linear equations is

$$\overline{\phi_b} = \sum_{n \geq 1} \frac{D_b^n \phi_b^n}{2^{n-1}}, \quad (4.5)$$

and the entropy on the length of the preimage of b being

$$H_b^{length} = - \sum_{n \geq 1} \frac{D_b^n}{2^{n-1}} \log_2 \left(\frac{D_b^n}{2^{n-1}} \right). \quad (4.6)$$

In the best case (which can be satisfied by choosing C and f properly), where ϕ_b^n is the least possible, we can derive the proposition:

Proposition 4.3: Consider a proper encoder (C, f) such that $f(C) = \{0, 1\}$ with 0 and 1 having the same probability for random input sequences. Let $\overline{\phi_b}$ and H_b^{length} be the average number of retrieved equations for a bit b and the associated entropy on the length of the preimage of b . Then, we have

$$\overline{\phi_b} = E_b - \delta_b^\phi, \text{ with } \delta_b^\phi = \sum_{n \geq 1} \frac{D_b^n}{2^{n-1}} \min(n, D_b^n - 1), \quad (4.7)$$

and

$$H_b^{length} = E_b - 1 - \delta_b^H, \text{ with } \delta_b^H = \sum_{n \geq 1} \frac{D_b^n}{2^{n-1}} \log D_b^n. \quad (4.8)$$

Moreover, δ_b^ϕ and δ_b^H are both positive, and they satisfy $\delta_b^\phi \geq \delta_b^H$.

Proof: D_b^n can be at most 2^n , so $\log D_b^n$ is at most n . Then for any $a > 1$,

$a-1 \geq \log a$, so we obtain

$$\min(n, D_b^n - 1) \geq \log D_b^n \quad (4.9)$$

for every n such that $D_b^n \neq 0$. Using the relation (4.9), we obtain

$$\begin{aligned} \overline{\phi_b} &= \sum_{n \geq 1} \frac{D_b^n \phi_b^n}{2^{n-1}} = \sum_{n \geq 1} \frac{D_b^n (n - \log D_b^n)}{2^{n-1}} = \sum_{n \geq 1} \frac{D_b^n n}{2^{n-1}} - \sum_{n \geq 1} \frac{D_b^n \log D_b^n}{2^{n-1}} \\ &= \sum_{n \geq 1} \frac{D_b^n n}{2^{n-1}} - \delta_b^\phi \quad \text{with} \quad \delta_b^\phi = \sum_{n \geq 1} \frac{D_b^n \log D_b^n}{2^{n-1}} = \sum_{n \geq 1} \frac{D_b^n}{2^{n-1}} \min(n, D_b^n - 1). \\ H_b^{length} &= \sum_{n \geq 1} \frac{D_b^n n}{2^{n-1}} - \sum_{n \geq 1} \frac{n}{2^{n-1}} - \sum_{n \geq 1} \frac{D_b^n \log D_b^n}{2^{n-1}} \end{aligned}$$

$$= E_b - 1 - \sum_{n \geq 1} \frac{D_b^n \log D_b^n}{2^{n-1}} = E_b - 1 - \delta_b^H, \quad \delta_b^H = \sum_{n \geq 1} \frac{D_b^n \log D_b^n}{2^{n-1}}. \quad \square$$

As a result of proposition 4.3, when 0 and 1 have the same number of preimages of each given length, to retrieve L equations has complexity at least $O(2^{\frac{E-1-\delta^\phi}{E-\delta^\phi}L})$.

Suppose that input sequence to compression mechanism is given by LFSR of length L . It is therefore possible to retrieve L linear equations on the input bits of LFSR with an attack of average complexity $O(2^{(\frac{H_0^{length}}{\bar{\phi}_0} + \frac{H_1^{length}}{\bar{\phi}_1})\frac{L}{2}})$ requiring $O(L)$ output bits.

Like in the previous attacks, when $\frac{\bar{\phi}_0}{H_0^{length}} \neq \frac{\bar{\phi}_1}{H_1^{length}}$ holds, the complexity of attack can be reduced by seeking for a sequence where mostly bits 0 or 1 appear, we can decrease attack complexity, but we need more output bits.

Equation retrieval is not more effective than exhaustive reconstruction and their complexities are same, if the number of the possible preimages of length n for each output bit is one.

4.2.4. FBDD Attack

Krause [7] introduced a new type of attack against keystream generators, which is known as Free Binary Decision Attack (FBDD- attack), which is used in cryptanalysing of LFSR-based keystream generators.

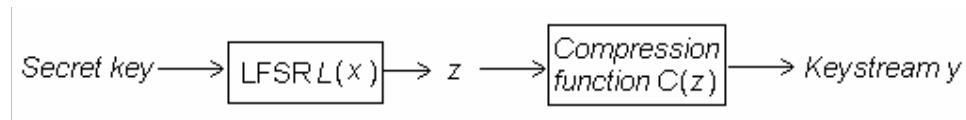


Figure 4.1. LFSR-Based keystream generator

We assume that our LFSR-based keystream generator E has a key-length n which is known as secret key. It also has linear bit-stream generator L and a compression function $C: \{0,1\}^* \rightarrow \{0,1\}^*$. Let $x \in \{0,1\}^n$ denote an initial state for L (Figure 4.1).

This cryptanalysis applied if the following property is fulfilled :

Independence Assumption: For all $m \geq 1$ the probability that $C(z)$ is a prefix of y for a randomly chosen and uniformly distributed $z \in \{0,1\}^m$ is the same for all keystreams y . We define this probability as $p_C(m)$.

Second parameter is known as α , which depends on the probability $p_C(m)$.

$$\alpha = \frac{1}{m} (H(Z^{(m)}) - H(Z^{(m)} | Y)) = \frac{1}{m} (m - \log(p_C(m)2^m)) = -\frac{1}{m} \log(p_C(m)) \quad (4.10)$$

where $Z^{(m)}$ denotes random $z \in \{0,1\}^m$ and Y a random keystream.

As you know, compression algorithm generates the keystream in online manner, time average between two succeeding key should be small and not vary too much which implies the following rule:

Each internal bitstream z can be divided into consecutive elementary blocks $z = z^0 z^1 z^2 \dots z^{s-1}$ such that $C(z) = y_0 y_1 \dots y_{s-1}$ with $y_j = C(z^j)$ for all $j = 0, 1, \dots, s-1$ and the average length of elementary blocks is small constant. This rule implies that $p_C(m)$ supposed to behave as $p_C(m) = 2^{-\alpha m}$, for constant $\alpha \in (0, 1]$.

Another parameter of C is the maximal number of output bits which C produces on internal bitstreams of length m . Due to partition rule this value can be supposed to behave as $m\gamma$, where γ is defined as best compression ratio of C .

Observe that if C always reads the same number k of internal bits for producing one output bit, then $\alpha = \gamma = 1/k$. If this number is not constant then α can be obtained by the formulae :

$$2^{-\alpha m} = p_C(m) = \sum_{i=0}^m 2^{-i} \Pr ob_z [|C(z)| = i], \quad (4.11)$$

where z denotes a random, uniformly distributed element from $\{0,1\}^m$. This equation yields $\gamma \geq \alpha$ and $\gamma \alpha^{-1} \leq 1$.

FBDD Assumption: The Graph ordering G_m^C has polynomial size in m . Moreover, for arbitrary keystreams y , the minimal G_m^C -driven FBDDs which decide for $z \in \{0,1\}^m$ whether $C(z)$ is a prefix of y have polynomial size in m , where G_m^C denote the graph ordering for all $m \geq 1$, which assigns to each internal bitstream z the order in which C reads the first m bits of z .

Pseudorandomness Assumption: For all keystreams y and all $m \leq [\alpha^{-1}n]$ it holds that

$$\Pr ob_z [C(z) \text{ is a prefix of } y] \approx \Pr ob_x [C(L_{\leq m}(x)) \text{ is prefix of } y]$$

where z and x denote uniformly distributed random elements from $\{0,1\}^m$ and $\{0,1\}^n$, respectively.

In [7], it is shown that FBDD assumption and Pseudorandomness Assumption are fulfilled by LFSR-based keystream generators.

Theorem 4.1: Let E be an LFSR-based keystream generator of key-length L with linear bitstream generator LG and a compression function C of information rate α and best case compression ratio γ . Let C fulfill the Independence assumption, the FBDD assumption and the Pseudorandomness assumption. Then, there is an $L^{o(1)} 2^{(1+\alpha)(1-\alpha)/L}$ -time bounded algorithm which computes the secret initial state x from the first consecutive bits $[\gamma \alpha^{-1}L]$ of $C(L(x))$.

4.3. Optimal Choices

Based on results obtained in previous sections, we estimate the properties of optimal proper encoder.

Firstly, based on security analysis, in order to prevent attacks based on asymmetry between the preimage of 0 and 1, each output bit must have same number of preimages of given length.

Moreover, in order to maximize the most probable case attacks complexity, we need to arrange that the length of the shortest word in C should be close as possible to the average length of the words in C .

Generally, equation retrieval attacks can have better complexity than exhaustive reconstruction attack. In order to lessen their efficiency, each bit should have at most one preimage of each length.

Let's think about construction of an optimum framework. For an output rate at least $1/2$, the number of choices for the proper encoder are finite and their output rate is 1 or $1/2$, because of symmetry requirements. Taking the output rate as 1 does not make any sense for security of compression mechanism and is very easy to break. Moreover, if we take the output rate as $1/2$, we have six proper compression algorithm. Suitable set is $C=\{00, 01, 10, 11\}$; and the function f with two preimages for output 1 and 0. For each choice, equations retrieval attack is easy to apply because length of the preimages are constant, and complexity is $O(L)$.

Let's take h the minimal depth in our code tree, to produce output. Due to symmetry requirements, h should be same for both 1 and 0 and also the number of preimages for length h , to ensure that each output bit must have the same number of preimages of each length.

The complexity of most probable case attack is $O(2^{\frac{h-1}{h}L})$. In order to maximize output rate we need to choose h as 2, and no level in the tree should have only internal nodes. We can comprehend that for each depth more than two, tree must have exactly two leaves, until the last level where there are four leaves.

If T_2^d is the set of trees with maximum depth d and minimum depth 2, and exactly 2 leaves for depths 2,3,4,5....., $d-1$ and 4 leaves for $d < \infty$. In order to obtain proper encoders using these codes, person should use a function f such that the number of preimages of 0 and 1 for each depth is same.

This optimal code can be done by also choosing the $h > 2$, which makes most probable case attacks more complex. The tree considered then has 2^{h-1} leaves for depths $h, \dots, d-1$ and maximal depth d where there is 2^h leaves. On the other hand equations retrieval attacks are more efficient for $h > 2$.

Proposition 4.4: Consider a proper encoder (C, f) such that the code tree of C is a T_h^d tree and 0 and 1 have the same number of preimages of each given length. Suppose that C is such that the number of equations linking the preimages of b of length n is the least possible, namely $\phi_b^n = \max(0, n - (D_b^n - 1))$. Then we have :

1. For every $h \geq 2$, the entropy on the length of the preimage of a given output bit

$$b \text{ is } H^{\text{length}} = 2 - 2^{h-d} \left(\frac{3+d-h}{2} \right).$$

2. We have $\overline{\phi_b} = h + 2 - 2^{h-2} - 2^{h-d} + 2^{h-1-d} (2^{h-2} - 1 - d)$, which is equal to 3 for $h = 3$ and $d = \infty$.

Proof: See the Appendix B. □

As a consequence of the results obtained, we can say that with the entropy remaining less than 2, the complexity of equations retrieval attacks does not grow fast when the output rate decreases. Therefore optimal framework against these attacks is reached when the code tree belongs to T_2^∞ . Attack complexity remains at least $O(2^{L/2})$ for trees in T_2^d .

The optimum choice for compression mechanism will be the one that provides the best output rate and security. As you see in the table 4.1, encoders with the associated code tree belongs to T_3^∞ seems more resistant against attacks, but on the other hand T_2^∞ has the best output rate.

Definition 4.1: We say that a proper encoder (C, f) is an optimal encoder if the associated code tree belongs to T_2^∞ , and if 0 and 1 have exactly one preimage by f of length l , for $l \geq 2$.

Complexities of attacks against proper encoders constructed on the basis of general trees T_h^d can be seen in table 4.1. We should also remember that most probable case attack requires much known keystream, on the other hand the other attacks require only a number of bits linear in L , where L is the number of the bits we want to obtain.

We can also see that equations retrieval attacks are more powerful against T_2^d trees than exhaustive reconstruction, which is why we didn't consider them as an optimal.

Table 4.1. Characteristics and attack exponent against T_h^d trees filtering LFSRs.

	Output Rate	Exhaustive reconstruction	Most probable case	Equations Retrieval	FBDD Attacks
T_h^d	$\frac{1}{h+1-2^{h-d}}$	$\frac{h+2^{h-d}}{h+1-2^{h-d}} L$	$\frac{h-1}{h} L$	See Prop.4.4	N /A
T_h^∞	$\frac{1}{h+1}$	$\frac{h}{h+1} L$	$\frac{h-1}{h} L$	See Prop.4.4	N /A
T_2^d	$\frac{1}{3-2^{2-d}}$	$\frac{2-2^{2-d}}{3-2^{2-d}} L$	$\frac{1}{2} L$	$\frac{2-2^{2-d}(\frac{d+1}{2})}{3-2^{2-d}-d2^{1-d}}$	N /A
T_2^∞ (ABSG)	$\frac{1}{3}$	$\frac{2}{3} L$	$\frac{1}{2} L$	$\frac{2}{3} L$	$\cong 0.54L$
T_3^∞	$\frac{1}{4}$	$\frac{3}{4} L$	$\frac{2}{3} L$	$\frac{2}{3} L$	$\cong 0.62L$
SSG	$\frac{1}{4}$	$\frac{3}{4} L$	$\frac{1}{2} L$	$\frac{2}{3} L$	$\cong 0.65L$

4.4. Some Popular Compression Algorithms

4.4.1. ABSG

ABSG is proposed algorithm in [2], which is modified version of BSG (Bit Search Generator). The idea behind ABSG is very simple; the compression algorithm first split up input sequence into codewords, then ABSG maps codewords to their second bit.

Let s denote a binary input string and z represent the output of ABSG algorithm. Each output bit is produced splitting up input sequence s into subsequences of the form with $(b\bar{b}^ib)$ and $i \geq 0$. The output bit is b for $i = 0$ and $\bar{b}$ otherwise.

Summary of ABSG algorithm can be seen in table 4.2:

Table 4.2. ABSG Algorithm; s is the input string, z is the output

<i>ABSG</i>
$i=0; j=0;$ while(1) output $z_j = s_{i+1}, b = s_i$ $i=i+1;$ while($s_i = \bar{b}$) $i=i+1;$ $i=i+1;$ $j=j+1;$

The output rate of ABSG is found 1/3 using the formula:

$$\frac{E(f(C))}{E(C)}.$$

E_b is the expected length of the preimage b in the input sequence when a bit b appears in the output. E_b is found 3 for ABSG by using Eq. (4.1).

H_b , the entropy on the preimage of b in the input sequence is 2 for ABSG using the equation

$$\sum_{w \in C_b} \frac{1}{2^{|w|-1}} \log_2 \left(\frac{1}{2^{|w|-1}} \right)$$

which was mentioned in section 4.2.1.

As mentioned above, $E(C)$ is 3 for ABSG, it means that any output bit is generated by three input bits in average.

As you can see in the section 4.2.1, the exhaustive reconstruction attack has complexity $2^{\frac{H_L}{E}}$, then exhaustive reconstruction attack has complexity $2^{\frac{2}{3}L}$ for ABSG where the input sequence is given by a LFSR of length L with a public feedback polynomial and with secret key as its initial state.

We have also simulated this attack on computer. As mentioned before, output bit b is generated by three input bits in average using ABSG algorithm. We have used all output sequences y with $|y| = n$, then reconstruct all possible input sequences x for each y with $|x| \leq 6n$ and $\text{ABSG}(x) = y$. In average, ABSG output sequence y with $|y| = n$ is produced by the preimage x with $|x| = 3n$, so we guarantee spanning most of the set that contains all possible preimages of $|y| = n$ by reconstructing possible preimages x with $|x| \leq 6n$. The probability that any ABSG output sequence y with $|y| = n$ is produced by the preimage that belongs to the set of reconstructed ones is found approximately 0.999956 with simulation, so neglecting possible preimage sequences x where $|x| > 60$ is acceptable in mathematical manner.

E_b and H_b are found 3 and 2 as expected using the simulation mentioned above for $n = 4, 6, 8, 10, 11, 12$; so complexity of this attack is still $2^{\frac{2}{3}L}$ using simulation.

The other attack we need to take into consideration is most probable case attack. Using the results in the section 4.2.2, we have found that the complexity of most probable case attack against ABSG is $2^{L/2}$, because the shortest codeword length which produces one bit output is two. We have also applied this attack empirically and while applying this kind of attack we assume that ABSG output sequence y where $|y| = n$, is generated by its most probable preimage sequence $x^*(y)$ where $|x^*(y)| = \min(|x|) = 2n$.

Suppose that input sequence to ABSG is the output of maximum length LFSR of length 10. To capture the initial state of LSFR, we need to obtain 10 consecutive output bits of LFSR. For each ABSG output sequence y where $|y| = 5$, we bet that this output sequence is produced by its most probable preimage $x^*(y)$ where $|x^*(y)| = 10$. Our bet is succesful with probability 2^{-5} , so we need to make 2^5 bets to reach success. We have reconstructed all possible preimage sequences x which produce output sequence y where $|y| = 5$ and $x \leq 30$, We neglect possible preimage sequences x where $x > 30$ because they have very low probability to appear in the input sequence. The probability that most probable case preimage $x^*(y)$ appear in the input sequence out of all preimages where $x \leq 30$ is also 2^{-5} . To guarantee one of our bet is succesful we need to repeat this process 2^5 times.

The next attack is equations retrieval attack. Let's consider the ABSG code tree. For every length $n \geq 2$, there is exactly one preimage of 0 and one preimage of 1 of length n . We find that

$$\overline{\phi_b} = \sum_{n \geq 2} \frac{n}{2^{n-1}} = 3 \quad \text{and} \quad H_b^{length} = \sum_{n \geq 2} \frac{n-1}{2^{n-1}} = 2.$$

Using the results shown above we see that equations retrieval attack has the same complexity of exhaustive reconstruction attack which is $2^{\frac{2}{3}L}$.

The next type of attack is FBDD attack. As it is clear, one bit output is generated by at least two bits of input, so γ is $\frac{1}{2}$.

We also need to find α where $\alpha = -\frac{1}{m} \log(p_C(m))$ and $p_C(m)$ is the probability that for all $m \geq 1$, $C(z)$ is a prefix of y for a randomly chosen and uniformly distributed $z \in \{0,1\}^m$ is the same for all keystreams. I have found this α using simulation. For $m = 5, 10, 15, 20$ We have found $p_C(m)$ values. Then using the relation $\alpha = -\frac{1}{m} \log(p_C(m))$, I have found the α values. In average α is found as 0.30.

As a result, we see that complexity of FBDD attack for ABSG is approximately $L^{O(1)} 2^{0.54L}$ while requiring $O(1.66L)$ output bits.

4.4.2. BSG(Bit Search Generator)

BSG algorithm was presented in [3], and then was described again with ABSG in [2]. BSG is a member of T_2^∞ type code tree. The corresponding code for BSG is $C = \{01^k 0, 10^k 1, k \geq 0\}$ where output rate for BSG is $\frac{1}{3}$ like ABSG. In figure 4.2, BSG code tree is shown.

Exhaustive reconstruction attack complexity is $O(2^{\frac{L}{2}})$ while requiring $O(\frac{L}{2})$ output bits.

BSG converts codewords of length two to 0, and other codewords to 1, so it is not an optimal encoder. This asymmetry results several attacks which were shown in [1,2,8]. For

instance the equations retrieval attack uses advantage of this asymmetry and can be applied to BSG with the complexity of $O(2^{\frac{L}{3}})$.

Most probable case attack complexity for BSG is $O(L^3 2^{\frac{L}{4}})$ while requiring $O(2^{\frac{L}{4}})$ output bits [2].

Table 4.3. BSG Algorithm; s is the input string, z is the output

<i>BSG</i>
$i=0; j=0;$ <i>While</i> (1) $i=i+1;$ $j=j+1;$ $b = s_i$ $i=i+1;$ if($b = s_i$) Output $z_j = 0;$ else $z_j = 1;$ $i=i+1;$ while ($s_i = \bar{b}$) $i=i+1;$

FBDD attack complexity for BSG is $L^{O(1)} 2^{0.33L}$ to $L^{O(1)} 2^{0.62L}$, however it is not clear whether FBDD attack will work because BSG does not fulfill independence assumption.

We see that BSG is weaker against attacks than ABSG while providing the same output rate, so we don't prefer BSG to use in our applications.

5. MEMORY BIT EFFECT ON COMPRESSION FOR PSEUDO-RANDOM GENERATION

In this section, we investigate the memory bit effect on compression for pseudo-random generation. As mentioned earlier, there are two important factor we need to take into account for compression:

- Output rate
- Security.

We take ABSG as a reference for our work. As discussed previously, ABSG is compression algorithm which belongs to code tree T_2^∞ . This code tree is determined as the optimal code tree for compression. ABSG seems most optimum compression algorithm among existing algorithms due to its output rate and security analysis.

We focus on increasing output rate while providing enough resistance against attacks. We expect that using memory bit increase the output rate.

Firstly, We present you the new compression algorithm Editing Bit Search Generator-EBSG which uses one memory bit. We insert the bit stored in memory to the input sequence after each output generation.

5.1. Editing Bit Search Generator – EBSG

In this section, we propose a new algorithm named as Editing Bit Search Generator EBSG which is modified version of ABSG. It increases the output rate from $\frac{1}{3}$ to $\frac{1}{2}$ by a insertion of a bit which is stored in memory bit to input sequence after generation of each output bit. Firstly, we give the general construction of the proposed algorithm EBSG and some of its properties.

The EBSG is identical to the ABSG except that after each output generation one bit is inserted to the input string s . The value of inserted bit for the first iteration is 0, it changes to 1 or does not change due to each input sequence or codeword that produce output.

Let's denote the inserted or added bit as a and initially we set it to 0. For this algorithm each output is produced from the pattern $\bar{b}b^k\bar{b}$ for $k \geq 0$ and inserted bit is a . If k is odd, inserted bit changes to $\bar{a}$, otherwise nothing changes. Its implementation can be seen in the table 5.1:

Table 5.1. EBSG algorithm; s is the input string, z is the output and a is parameter determining the value of inserted bit after each output generation

<i>EBSG</i>
$i=0; j=0; a=0;$ while(1) output $z_j = s_{i+1}, b = s_i$ $i=i+1;$ $k=0;$ while ($s_i = \bar{b}$) $i=i+1; k=k+1;$ if (mod($k,2$)=1) $a = \bar{a};$ $s_i = a;$ $j=j+1;$

Let $s = 1000101010101110110$ be the input sequence. Then the action of the EBSG on s can be shown as

$$\underbrace{10001}_0 \wedge \underbrace{01}_0 \wedge \underbrace{00}_0 \wedge \underbrace{10}_1 \wedge \underbrace{11}_1 \wedge \underbrace{01}_0 \wedge \underbrace{110}_1 \wedge \underbrace{110}_1 \Rightarrow 00011011$$

Proposition 5.1: Let s be a pseudorandom bit sequence. Assume that the output sequence $z = EBSG(s)$ is evenly distributed. Then the output rate of EBSG is asymptotically $\frac{1}{2}$.

Proof: For any output bit y_i is produced after $i+1$ input bits with probability 2^{-i} for $i \geq 1$. So the average number of input bits needed to produce one output bit is $\sum_{i=1}^{\infty} (i+1)2^{-i} = 3$. Since at each output generation one bit is inserted, the net average number of used input bits is $3-1 = 2$. This shows that the rate of the EBSG is asymptotically $1/2$. $\square$

Proposition 5.2: Let s be an evenly distributed binary sequence. After one output bit generation, the state of algorithm changes with probability $1/3$ and remains same with probability $2/3$.

Proof: Assume $S_t \in \{1,0\}$ denotes the state of algorithm for the time t or it can be defined as the inserted bit for time t . For each output bit y_i ; if y_t is produced from the image block of $b^k \bar{b}$ for a positive odd number k , then the state of the algorithm is changed $S_{t+1} \neq S_t$. k is 1 with probability $1/4$, k is 3 with probability $1/16$..., so $k = i$ with probability of $1/2^{i+1}$. Thus, $P(k = \text{odd}) = 1/3$, which is equivalent to say that $P(S_{t+1} \neq S_t) = 1/3$. On the other hand, the state of algorithm remains same with a probability $P(S_{t+1} = S_t) = 2/3$. $\square$

Proposition 5.3: Let s be an pseudorandom bit sequence. Assume that the output sequence $y = EBSG(s)$ is evenly distributed. Then for every output bit b , the expected number of known bits of s is 2.

Proof: Let $S_t \in \{0,1\}$ denote the state of algorithm (inserted bit at time t) and $y_t \in \{0,1\}$ represents the observed output at time t . Then we can write the following conditional state transition probabilities:

$$P(S_{t+1} = 0 | S_t = 0, y_t = 0) = 1$$

$$P(S_{t+1} = 1 | S_t = 0, y_t = 1) = 0$$

$$\begin{aligned}
P(S_{t+1} = 0 | S_t = 0, y_t = 1) &= 1/3 & P(S_{t+1} = 1 | S_t = 0, y_t = 1) &= 2/3 \\
P(S_{t+1} = 1 | S_t = 1, y_t = 1) &= 1 & P(S_{t+1} = 0 | S_t = 1, y_t = 1) &= 0 \\
P(S_{t+1} = 1 | S_t = 1, y_t = 0) &= 1/3 & P(S_{t+1} = 0 | S_t = 1, y_t = 0) &= 2/3
\end{aligned}$$

We can express $P(S_{t+1} | y_t)$ as

$$P(S_{t+1} | y_t) = \frac{\sum_{\forall S_t \in \{1,0\}} P(S_{t+1} | S_t, y_t) P(y_t | S_t) P(S_t)}{P(y_t)} \quad (5.1)$$

Since assume that y is evenly distributed and $P(S_t) = 1/2$ for $S_t \in \{0,1\}$, then we get the following probabilities

$$\begin{aligned}
P(S_{t+1} = 0 | y_t = 0) &= 5/6, & P(S_{t+1} = 1 | y_t = 0) &= 1/6, \\
P(S_{t+1} = 0 | y_t = 1) &= 1/6, & P(S_{t+1} = 1 | y_t = 1) &= 5/6.
\end{aligned}$$

$$P(S_{t+1} = 0) = 5/6 \quad \text{for } y_t = 0 \quad \text{and} \quad P(S_{t+1} = 1) = 5/6 \quad \text{for } y_t = 1.$$

In case of algorithm $S_t = 0$; every zero in output sequence y corresponds to a bit 0 in the input sequence s . Consequently, if an output bit is a zero, then one input of s is known. On the other hand, every one in output sequence y corresponds to the pattern $1^i 0$ with $i \geq 1$ in the sequence s and the following possibilities may exist: two bits are known with probability $1/2$, three bits are known with probability $1/4$, $\dots$, so $i+1$ bits are known with probability of 2^{-i} for $i \geq 1$.

In case of algorithm $S_t = 1$; we just swap the probabilities 0 and 1, so every one in the output sequence y corresponds to a bit 1 in the sequence s and every zero in the output sequence y corresponds to the pattern $0^i 1$ with $i \geq 1$.

Suppose that $y_{t+1} = 0$; according to this scenario, for an output bit b the expected number of input bits E_b is :

$$\begin{aligned}
E_b &= P(y_t = 0) \left[P(S_{t+1} = 0 | y_t = 0) + \sum_{i=1}^{\infty} P(S_{t+1} = 1 | y_t = 0)(i+1)2^{-i} \right] \\
&\quad + P(y_t = 1) \left[P(S_{t+1} = 0 | y_t = 1) + \sum_{i=1}^{\infty} P(S_{t+1} = 1 | y_t = 1)(i+1)2^{-i} \right] \\
&= \frac{1}{2} + \left[\frac{5}{6} + \sum_{i=1}^{\infty} \frac{1}{6}(i+1)2^{-i} \right] + \frac{1}{2} \left[\frac{1}{6} + \sum_{i=1}^{\infty} \frac{5}{6}(i+1)2^{-i} \right] = 2.
\end{aligned} \tag{5.2}$$

Notice that we can compute the same value for $y_{t+1} = 1$ with assuming output sequence y is evenly distributed. $\square$

5.2. Security Analysis of EBSG

In this section, we give general analysis of the security provided by the EBSG compression algorithm. We focus on the case when the input sequence is the output of a maximal length LFSR. We also assume that input sequence is evenly distributed. (independent identically distributed Bernoulli process with probability 1/2.)

5.2.1. Exhaustive Reconstruction Attack Against EBSG

As mentioned in section 4.2.1, this attack type aims to reconstruct possible input sequences from the output sequences.

First, we need to find what is the probability of inserted bit to be 0 or 1 at any random instance, because when we reconstruct input sequence from the output sequence, firstly we guess the first added bit. It is clear that to calculate the entropy on preimage and probability of any possible preimage to appear in the input sequence, we need this probability.

We have taken all input sequences s where $|s| = 40$ and process them using EBSG algorithm. This means that we have processed this application 2^{40} times, and for each process we have stored the added bits which have been used for each application in vectors.

This process lasted approximately 168 hours on 1.83 Ghz Core Duo computer . At the end in vectors we have seen that $72.0790956 \times 10^{11}$ inserted bits are 1 and $73,9116150 \times 10^{11}$ inserted bits are 0. As a result at any random instance, probability of inserted bit to be 1 is 0.4937 and to be 0 is 0.5063, so we can accept that 1 and 0 has almost equal probability to be inserted at any random instance.

E_b is the expected length and H_b is the entropy on the preimage of output bit b . We have found E_b in previous section. In this section, We find E_b and H_b empirically on computer.

When we reconstruct EBSG output sequences, we focus on all output sequences y with $|y| = n$. There are 2^n possible elements of the set $Y = \{y \in Y, |y| = n\}$. EBSG output sequence y with $|y| = n$ is produced by preimage x and in average $|x| = 2n$. With respect to this observation, we have reconstructed all possible preimage sequences x of $|y| = n$ where $x \leq 4n$ and $|EBSG(x)| = n$. We found the probability that preimage of any y with $|y| = n$ belongs to set of reconstructed possible preimages is 0.999669, so we almost guarantee that we have spanned most of the set which contains all possible preimages x where $|EBSG(x)| = n$.

Firstly, we find E_b empirically.

$$E_y = \sum_{x \in C_y} P(x | y) |x| \quad (5.3)$$

$$E_Y = \sum_{\forall y \in \{|y|=n\}} E_y P(y) \quad (5.4)$$

$$E_b = \frac{E_Y}{n} \quad (5.5)$$

The sequence x is the preimage of the output sequence y with probability $P(x | y)$. C_y is the set that contains possible preimages of y , $x \in C_y$ and $|y| = n$.

The sequence y appears in the output with probability $P(y)$ where $\sum_{y \in \{|y|=n\}} P(y) = 1$.

E_y is the expected length on the preimage of output sequence y .

E_Y is average expected length on the preimage of output sequences with length n . We also divide E_Y by n , to find E_b in average.

E_b is found 2 as expected for $n = 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14$.

The next thing we need to find is H_b for EBSG algorithm.

$$H_y = - \sum_{x \in C_y} P(x | y) \log P(x | y) \quad (5.6)$$

$$H_Y = \sum_{y \in \{|y|=n\}} P(y) H_y \quad (5.7)$$

$$H_b = \frac{H_Y}{n}. \quad (5.8)$$

H_y is entropy on preimage of output sequence y . H_Y is the average entropy on the preimage of output sequences with length n .

We have used the following steps to find H_b empirically for EBSG:

- We find entropy one by one on the preimage of all output sequences y where $|y| = n$.

- We calculate average entropy H_Y on the preimage of output sequences with length n using the entropies calculated in previous step.
- We divide the H_Y by n to find the average entropy on the preimage of one bit. H_b is found empirically 1.1 with $n = 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14$.

Suppose that input sequence is given by a LFSR of length L with a public feedback polynomial and with the secret key as its initial state. It is therefore possible to recover the complete state of LFSR with an attack of average complexity $O(2^{0.55L})$, requiring $O(\frac{L}{2})$ consecutive output bits.

As you can see in the table 5.2, each output sequence y where $|y|=10$ has almost same possibility to appear in the output, however number of possible preimages x where $|x| \leq 40$ for each y are not same.

In EBSG, the current output bit is not independent from the inserted bit and also previous output bit, so the number of possible preimages changes for each output sequence. For example, if the first added bit for output sequence 0000000000 is guessed as zero, then it is impossible to have preimage that contain one for that sequence.

Table 5.2. Statistical properties of some EBSG output sequences

Output sequence	Probability to appear in the output	Number of possible preimages
0000000000	0.00097655	33830
0001110111	0.000976418	840974
0011000111	0.000976369	1123830
0100110101	0.000975819	3950930
0110011001	0.000976054	2730810
1000010001	0.000976337	1072970
1001100001	0.000976276	1402450
1100111101	0.000976296	1232580
1111111111	0.00097655	33830

5.2.2. Most Probable Case Attack Against EBSG

In this type of attack, we bet that any output bit b is generated by the most probable preimage of b as explained in section 4.2.2.

In EBSG for any output bit b , there are two most probable preimage depending on the current inserted bit.

When we reconstruct EBSG output sequences using most probable case preimages, if added bit is guessed as b for the current output and output is b , then our most probable preimage becomes bb where the b is inserted bit. On the other hand if inserted bit is guessed as $\bar{b}$, then our most probable image for output b becomes $\bar{b}b\bar{b}$, where the first $\bar{b}$ is inserted bit.

We assume that in average $\frac{L}{3}$ output bits are guessed as coming from bb and $\frac{L}{3}$ output bits are guessed as coming from $\bar{b}b\bar{b}$.

- If output bit is b and inserted bit is b , then probability that output bit b is generated by bb is 1.
- If output bit is b and inserted bit is $\bar{b}$, then probability that output bit b is generated by $\bar{b}b\bar{b}$ is $\frac{1}{2}$.
- When we are reconstructing the output sequence for the most probable case we need to guess the first inserted bit with an average probability of $\frac{1}{2}$.

Suppose that input sequence is given by LFSR of length L . It is then possible to retrieve complete state of LFSR with an average attack complexity of $O(2^{\frac{L}{3}+1})$, requiring $O(\frac{4L}{3} 2^{\frac{L}{3}+1})$ output bits (namely enough for the bet to succeed).

For example take the output sequence 10110011, if we reconstruct the this output sequence for the most probable case:

$$\begin{array}{ccccccc} \overset{1}{\wedge} & \overset{1}{\wedge} & \overset{0}{\wedge} & \overset{1}{\wedge} & \overset{1}{\wedge} & \overset{0}{\wedge} & \overset{0}{\wedge} & \overset{1}{\wedge} \\ \underbrace{1} & \underbrace{01} & \underbrace{10} & \underbrace{1} & \underbrace{01} & \underbrace{0} & \underbrace{10} & \underbrace{1} \\ \underset{1}{} & \underset{0}{} & \underset{1}{} & \underset{1}{} & \underset{0}{} & \underset{0}{} & \underset{1}{} & \underset{1}{} \end{array} \Rightarrow \text{its most probable preimage is } 101101010101.$$

Suppose that input sequence is given by LFSR of length 12. The preimage of keystream 10110011 is 101101010101 with probability of $2^{-1}2^{-4} = 2^{-5}$, where 2^{-1} is the probability that first added bit is guessed as 1 and 2^{-4} comes from the fact that four times we bet that output bit is generated by $\bar{b}b\bar{b}$ type preimage with probability of 2^{-1} for each.

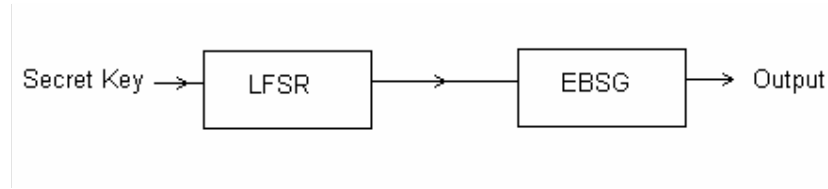


Figure 5.1. How EBSG produce output

Moreover, we also found the complexity of this attack empirically. Firstly, We explain you the procedure which we have followed to find the complexity of attack empirically.

As you know, to recover initial state of LFSR with length L , we need L consecutive output bits of that LFSR. When we reconstruct the most probable preimage $x^*(y)$ of output sequence y , our most probable preimage $x^*(y)$ should be at least L -bit long to recover the initial state of LFSR with length L . Using this observation:

$$x^*(y_n) = \operatorname{argmax} P(x | y_n), \quad x \in X(y_n), \quad (5.9)$$

where y_n is output sequence and $|y_n| = L$, $x^*(y_n)$ is the most probable preimage of y_n and $X(y_n)$ is the set that contains all possible preimages of y_n .

Observe that

$$q(y_n) \equiv P(x = x^*(y_n) | y_n) \quad (5.10)$$

$$\cong \frac{P(x = x^*(y_n))}{P(X(y_n))}$$

$$\cong \frac{2^{-(|x^*(y_n)|+1)}}{\sum_{x \in X(y_n)} 2^{-(|x|+1)}} .$$

For the i th query:

- Observe some y_n in the output sequence with probability $P(y_n)$,
- Bet that preimage of y_n is $x^*(y_n)$ with probability $q(y_n)$,
- At the i th query $P(\text{Success}) = \sum_{y_n \in \{|x^*(y_n)|=L\}} q(y_n)P(y_n) = \alpha$,

then complexity of this attack will be $\frac{1}{\alpha}$, because to guarantee one of our bet is

successful, we need to repeat this process $\frac{1}{\alpha}$ times.

In table 5.3, you can see some keystreams and their most probable preimages regarding EBSG algorithm. For any keystream y , where $|y| = n$, value of $|x^*(y)|$ may belongs to interval $[n, 2n)$, which means that length of the most probable preimage doesn't have to be same for each y .

Table 5.3: Some keystreams and their most probable preimages

Keystream	Most Probable Preimage
0000000010	000000001001
0000000000	0000000000
001010	0010011001
0000011010	00000101011001
111110	1111101
0110101	010101100110
000000000000001	0000000000000010
0000101010	0000100110011001
01101	01010110
0000001	00000010

Let's suppose that we are going to apply most probable case attack to recover initial state of LFSR of length L . We can prepare a look up table (see figure 5.2), where we store the all keystreams y that have most probable preimage $x^*(y)$ and $|x^*(y)| = L$. In the look up table, the right column represents keystreams y and left column represents their most probable preimages $x^*(y)$ where $|x^*(y)| = L$.

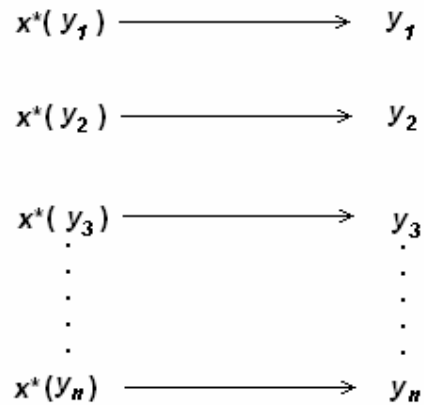


Figure 5.2: The look up table

Table 5.4 : Some elements of look up table which is used in most probable case attack
for $L = 10$

Most Probable Preimage	Keystream
0100101001	010010
1011001101	101011
0000100110	0000101
0010010101	0010011
0010101000	00110000
1010001011	10000111
0000000010	000000001
1111111101	111111110
0000000000	0000000000
1111111111	1111111111

In the table 5.5, you can see simulation results for the most probable case attack using look up table where input sequence is given by LFSR of length L where $L = 4, 6, 8, 10, 12, 14, 16, 18, 20$.

Table 5.5: Complexities of most probable case attacks using look up table where input sequence is given by LFSR of length L

Length of LFSR	Attack complexity (If EBSG is used)
4	2^2
6	$2^{2.6074}$
8	$2^{3.2189}$
10	$2^{3.8614}$
12	$2^{4.4422}$
14	$2^{5.0541}$
16	$2^{5.6657}$
18	$2^{6.2765}$
20	$2^{6.8954}$

Using the simulation results we have estimated the complexity of most probable case attack. Suppose that input sequence to EBSG algorithm is given by LFSR of length L . It is then possible to retrieve the complete state of LFSR with an attack of average complexity $O(2^{0.35L})$, requiring $O(\frac{4}{3}L 2^{0.35L})$ output bits.

5.2.3. Equations Retrieval Attack Against EBSG

EBSG has exactly one preimage of 0 and 1 of length m as shown in the following for $m \geq 1$:

- If inserted bit is 1 and output bit is 1, then possible preimage is 1;
- If inserted bit is 1 and output bit is 0, then possible preimage is $0^i 1$, where $i \geq 1$;
- If inserted bit is 0 and output bit is 1, then possible preimage is $1^i 0$ where $i \geq 1$;
- If inserted bit is 0 and output bit is 0, then possible preimage is 0 .

As it is shown above, output bit b has at most one preimage of length m , so we expect that $\overline{\phi_b} = E_b$ and $H_b^{length} = H_b$ as mentioned in section 4.2.3.

$\overline{\phi_b}$ and H_b^{length} are average number of retrieved equations for a bit b and the associated entropy on the length of the preimage of b in average can be found using the equations:

$$\overline{\phi_y} = \sum_n D_y^d \phi_y^d P(x^d | y) \quad (5.11)$$

$$\overline{\phi_Y} = \sum_{y \in \{|y|=n\}} P(y) \overline{\phi_y} \quad (5.12)$$

$$\overline{\phi_b} = \frac{\overline{\phi_Y}}{n} \quad (5.13)$$

$$H_y^{length} = - \sum_n P(x^d | y) \log P(x^d | y) \quad (5.14)$$

$$H_Y^{length} = \sum_{|y|=n} P(y) H_y^{length} \quad (5.15)$$

$$H_b^{length} = \frac{H_Y^{length}}{n} \quad (5.16)$$

D_y^d is the number of preimages of sequence y with length d , ϕ_y^d is number of linear equations on the input sequence, if x^d is preimage of y .

The sequence x^d is preimage of y with probability $P(x^d | y)$, y is output sequence with $|y| = n$.

$x^d \in C_y$ where C_y is the set that contains possible preimages of y and $|x^d| = d$.

$\overline{\phi_y}$ is average number of retrieved equation for output sequence y and H_y^{length} is the associated entropy on the length of the preimage of output sequence y .

$\overline{\phi_Y}$ is average number of retrieved equation for output sequences with length m and H_Y^{length} is the associated entropy on the length of the preimage of output sequences with length n .

$\overline{\phi_b}$ is found 2 and H_b^{length} is found approximately 1.1 using the Eq. (5.11), Eq. (5.12), Eq. (5.13), Eq. (5.14), Eq. (5.15) and Eq. (5.16) for $n = 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14$. Suppose that input sequence is given by LFSR of length L , it is therefore possible to retrieve L linear equations on the input bits of the LFSR with an attack of average complexity $2^{0.55L}$ requiring $O(\frac{L}{2})$ consecutive output bits.

5.2.4. FBDD Attack Against EBSG

We have explained in section 4.2.4 that Krause [7] introduced a new type of attack against keystream generators, which is known as Free Binary Decision Attack(FBDD-attack), which is used in cryptanalysing of LFSR-based keystream generators.

We assume that our LFSR-based keystream generator E has a key-length n which is known as secret key. It also has linear bit-stream generator L and a compression function C .

This cryptanalysis method relies on two assumption which are *FBDD Assumption* and *Pseudorandomness Assumption*. The cost of the cryptanalysis depends on two parameters of compression algorithm C . The first parameter is γ where it is the best case compression ratio of C .

There is another assumption which have to be fulfilled to apply FBDD cryptanalysis as mentioned in previous sections.

Independence Assumption: For all $m > 1$, the probability that $C(z)$ is a prefix of y for a randomly chosen and distributed $z \in \{0,1\}^m$ is the same for all keystreams y . Let denote this probability as $p_C(m)$. There is another parameter defined as α which depends on the probability $p_C(m)$.

The probability $p_C(m)$ is supposed to behave as $p_C(m) = 2^{-\alpha m}$ with a α constant where $0 < \alpha < 1$ and we have $\alpha \approx -\frac{1}{m} \log(p_C(m))$.

It is clear that parameter γ is 1 for EBSG algorithm.

The probability $p_C(m)$ is not same for all keystreams y where m is constant, so we see that EBSG algorithm does not fulfill the independence assumption, so it is not possible to use this attack against EBSG algorithm.

It is not clear that whether the FBDD attack is still relevant, now let's suppose it is and try to estimate its complexity against EBSG output sequence. We should at least take into account that the probability we defined as $p_C(m)$ does depend on y .

From an attacker's point of the view worst case occurs for attacker when output sequence is composed of $(10)^m$ or $(01)^m$ such that output sequence is 01010101.... 01 or 10101010...10.

Best case occurs when output sequence is composed of only 1s or 0s such that 111...1 or 000...0.

We have found $p_C(m)$ values for the best case and worst case using $m = 6, 8, 10, 15, 20$. You can see them in the table 5.6.

Table 5.6 : Some $p_C(m)$ values for the best case and worst case

m	$p_C(m)$ for best case	$p_C(m)$ for worst case
6	0.1563	0.1719
8	0.0762	0.1094
10	0.0352	0.0674
15	0.0047	0.0186
20	0.00060844	0.0055

Using the results found, in general case, α belongs to the interval $[0.36, 0.52]$. If the attack can be extended, its complexity will range from $L^{O(1)}2^{0.32L}$ to $L^{O(1)}2^{0.47L}$. Then, the attacker should start at the most interesting place in the output sequence, but in practice he has no hope to achieve best case complexity.

To obtain the best case attacker needs to find an all-zero or all-one subsequence with length $2L$, and this can hardly be achieved.

5.3. EBSG algorithm inserting more than one bit

In section 5.2, we have seen that output rate has increased from $1/3$ to $1/2$ when we insert one bit to input sequence which is stored in memory bit after each output generation.

Now, we focus on how behaviour of EBSG algorithm changes, if we insert more than one bit.

Table 5.7. EBSG (2) algorithm; s is the input string, z is the output and a is parameter determining the value of inserted bits after each output generation

<i>EBSG (2)</i>
$i=0; j=0; a=0;$ while(1) output $z_j = s_{i+1}, b = s_i$ $i=i+1;$ $k=0;$ while ($s_i = \bar{b}$) $i=i+1; k=k+1;$ if ($\text{mod}(k,2)=1$) $a = \bar{a};$ $i=i-1;$ $s_i = a;$ $s_{i+1} = \bar{a};$ $j=j+1;$

Firstly, we give you a brief explanation about procedures of EBSG inserting two bits and three bits.

The EBSG inserting two bits (EBSG(2)) is identical to the EBSG except that after each output generation two bits are inserted to the input string s . For the first iteration inserted bits are 01, they change to 10 or do not change due to each input sequence or codeword that produce output.

Let's denote the inserted or added bits as $a\bar{a}$ and initially we set it to 01. For this algorithm each output is produced from the pattern $\bar{b}b^kb$ for $k \geq 0$ and inserted bits are $a\bar{a}$. If k is odd, inserted bits change to $\bar{a}a$, otherwise nothing changes. Its implementation can be seen in the table 5.7.

Table 5.8 EBSG(3) algorithm; s is the input string, z is the output and a is parameter determining the value of inserted bit after each output generation

<i>EBSG (3)</i>
$i=0; j=0; a=0;$ while(1) output $z_j = s_{i+1}, b = s_i$ $i=i+1;$ $k=0;$ while ($s_i = \bar{b}$) $i=i+1; k=k+1;$ if (mod($k,2$)=1) $a = \bar{a};$ $i=i-2;$ $s_i = a;$ $s_{i+1} = \bar{a};$ $s_{i+2} = \bar{a};$ $j=j+1;$

Let $s = 1000101110001001110$ be the input sequence. Then the action of the EBSG(2) on s can be shown as

$$\underbrace{10001}_0 \wedge \underbrace{01}_0 \wedge \underbrace{1}_{10} \wedge \underbrace{10}_{01} \wedge \underbrace{0}_{10} \wedge \underbrace{01}_{10} \wedge \underbrace{001}_0 \wedge \underbrace{110}_1 \Rightarrow 00111001$$

The EBSG inserting three bits (EBSG(3)) is also identical to the EBSG except that after each output generation three bits are inserted to the input string s . The value of inserted bits for the first iteration is 011, it changes to 100 or do not change due to each input sequence or codeword that produce output.

Let's denote the inserted or added bit as $\bar{a}\bar{a}\bar{a}$ and initially we set it to 011. For this algorithm each output is produced from the pattern $\bar{b}b^k\bar{b}$ for $k \geq 0$ and inserted bits are $\bar{a}\bar{a}\bar{a}$. If k is odd, inserted bits change to $\bar{a}aa$, otherwise nothing changes. Its implementation can be seen in the table 5.8:

Let $s = 1000101010101110110$ be the input sequence. Then the action of the EBSG(3) on s can be shown as:

$$\underbrace{10001}_0 \wedge \underbrace{01}_{100} \wedge \underbrace{0}_{011} \wedge \underbrace{10}_{011} \wedge \underbrace{1}_{100} \wedge \underbrace{01}_{100} \wedge \underbrace{110}_{011} \wedge \underbrace{110}_{011} \Rightarrow 00110011$$

For both of algorithms, at least one bit is used to produce output bit like in EBSG, so we expect that output rate does not change and still be $\frac{1}{2}$.

E_b is the expected length and H_b is the entropy on the preimage of output bit b in average can be found empirically using the set up given in section 4.2.1 for inserting 2 bits and 3 bits condition.

We find E_b and H_b empirically for EBSG(2) and EBSG(3) for $n = 4, 6, 8, 10, 11, 12, 14$. The E_b and H_b are same for two types of algorithm and they are 2 and 1.15 respectively. We have seen that E_b is 2 like in EBSG, however there is a small increase for the entropy on preimage of output bit b . We have concluded that using 2 memory bits or more does not increase the output rate.

Now, we analyze the security properties of EBSG algorithm inserting more than one bit. We also assume that input sequence is evenly distributed. (independent identically distributed Bernoulli process with probability 1/2.)

5.3.1. Exhaustive Reconstruction Attack Against EBSG(2) and EBSG(3)

Using the set up given in section 5.2.1, average complexity of exhaustive reconstruction attack to retrieve the complete state of LFSR of length L , if LFSR is filtered by EBSG(3) or EBSG(3) is $O(2^{0.58L})$ requiring $O(\frac{L}{2})$ output bits.

5.3.2. Most Probable Case Attack Against EBSG(2) and EBSG(3)

We have found the most probable case attack complexities where input is given by LFSR of length L and $L = 4, 6, 8, 10, 12, 14, 16, 18, 20$ for EBSG(2) and EBSG(3). Results can be seen in table 5.9.

Using the results found empirically (see section 5.2.2 for details), we have concluded that most probable case attack average complexity is $O(\frac{6}{5} 2^{0.35L})$ for EBSG(2) and EBSG(3).

Table 5.9: Complexities of most probable case attacks using look up table where input sequence is given by LFSR of length L for EBSG(2) and EBSG(3)

Lenght of LFSR	Attack complexity
4	$2^{2.4150}$
6	2^3
8	$2^{3.6082}$
10	$2^{4.2189}$
12	$2^{4.8283}$
14	$2^{5.4422}$
16	$2^{6.0493}$
18	$2^{6.65584}$
20	$2^{7.2653}$

5.3.3. Equations Retrieval Attack Against EBSG(2) and EBSG(3)

EBSG(2) and EBSG(3) have one preimage of 0 and 1 of length m , for $m \geq 1$ and each inserted bits sequence:

For EBSG(2) algorithm:

- If inserted bits are 10 and output bit is 0, then possible preimage is 0^i1 where $i \geq 0$.
- If inserted bits are 01 and output bit is 1, then possible preimage is 1^i0 where $i \geq 0$.

For EBSG(3) algorithm:

- If inserted bits are 100 and output bit is 0, then possible preimage is 0^i1 where $i \geq 0$.
- If inserted bits are 011 and output bit is 1, then possible preimage is 1^i0 where $i \geq 0$.

$\overline{\phi_b}$ is found 2 and H_b^{length} is found approximately 1.15 for both algorithms inserting two bits and three bits (see section 5.2.3 for details), so if input sequence is given by LFSR of length L , it is therefore possible to retrieve L linear equations on the input bits of the LFSR with an attack of average complexity $2^{0.58L}$ requiring $O(\frac{L}{2})$ consecutive output bits.

5.3.4. FBDD Attack against EBSG(2) and EBSG(3)

The probability $p_C(m)$ is not same for all keystreams y where m is constant, so we see that EBSG (2) and EBSG(3) do not fulfill the independence assumption, so it is not possible to use this attack against EBSG(2) and EBSG(3) algorithm.

It is not clear that whether the FBDD attack is still relevant, now let's suppose it is and try to estimate its complexity against EBSG(2) and EBSG(3) output sequence. We should at least take into account that the probability we defined as $p_C(m)$ does depend on y .

From an attacker's point of the view worst case occurs for attacker when output sequence is composed of $(10)^m$ or $(01)^m$ such that output sequence is 01010101.... 01 or 10101010...10 for EBSG(2) and worst case occurs for EBSG(3) when output sequence is composed of only 1s or 0s such that 111...1 or 000...0.

Best case occurs when output sequence is composed of only 1s or 0s such that 111...1 or 000...0. for EBSG(2) and best case occurs for EBSG(3) when output sequence is composed of $(10)^m$ or $(01)^m$ such that output sequence is 01010101.... 01 or 10101010...10.

We found $p_C(m)$ values for the best case and worst case using $m = 6, 8, 10, 15, 20$. You can see them in the table 5.10.

Using the results found, in general case, α belongs to the interval $[0.3095, 0.5008]$. If the attack can be extended, its complexity will range from $L^{O(1)}2^{0.33L}$ to $L^{O(1)}2^{0.53L}$. Then,

the attacker should start at the most interesting place in the output sequence, but in practice he has no hope the achieve best case complexity.

Table 5.10. Some $p_C(m)$ values for the best case and worst case

m	$p_C(m)$ for best case	$p_C(m)$ for worst case
6	0.1094	0.25
8	0.0586	0.3196
10	0.0303	0.1128
15	0.0058	0.0394
20	0.00096607	0.0137

To obtain the best case attacker needs to find an appropriate subsequence with length $2L$, and this can hardly be achieved.

5.3.5. Results

Firstly, when we look at the attack complexities, inserting more than one bit has small advantage compared to inserting one bit related to attack complexities.

Moreover, when we have looked at the output sequences generated by these algorithms we see that there is not uniform distribution in the output sequences generated by EBSG(2) and EBSG(3), however if we insert only one bit, we provide uniform distribution in the output.

Table 5.11 : Attack complexities for EBSG, EBSG(2) and EBSG(3)

	EBSG	EBSG(2) or EBSG(3)
Exhaustive Reconstruction	$2^{0.55L}$	$2^{0.58L}$
Most Probable Case	$2^{0.35L}$	$6/5 \cdot 2^{0.35L}$
Equations Retrieval	$2^{0.55L}$	$2^{0.58L}$
FBDD	$L^{O(1)}2^{0.32L}$ to $L^{O(1)}2^{0.47L}$	$L^{O(1)}2^{0.33L}$ to $L^{O(1)}2^{0.53L}$

For EBSG algorithm:

- If inserted bit is 1, then output bit is 1 or 0 with probability $1/2$,
- If inserted bit is 0, then output bit is 1 or 0 with probability $1/2$.

For EBSG(2) or EBSG(3) algorithm:

- If inserted bits are 10 or 100, then output bit is 0 with probability one and output bit is 1 with probability zero.
- If inserted bits are 01 or 011, then output bit is 1 with probability one and output bit is 0 with probability zero.

We can see that if we insert more than one bit, it is not possible to produce output bit 1 with inserted bits 10 or 100 and output bit 0 with inserted bits 01 and 011, so it is not possible to have uniform distribution in output sequence.

We have concluded that inserting more than one bit does not provide us any important advantage, so we don't prefer it. Inserting one bit is the most efficient condition.

5.4. Comparison with ABSG, BSG and SSG

In the table 5.9, you can see the attack complexities to recover initial state of LFSR with length L , where it is filtered by different compression algorithms.

When we look at the complexity of attacks, EBSG seems better than BSG, but weaker than SSG and ABSG against general attacks. On the other hand, EBSG provides the best output rate. We can say that there is a trade-off between output rate and security. If output rate, also speed of the algorithm are more important for us, we can prefer EBSG for applications.

The other thing we should take into account that if input sequence given to compression algorithms is not uniformly distributed, then what is its resultant effect on output distribution and output rate ?

Let's suppose that s_i is the i th bit of the input sequence s , i th bit of s is 0 with probability $P(s_i = 0) = p$ and i th bit of s is 1 with probability $P(s_i = 1) = 1 - p$.

We have tested that how the distribution changes in output sequence for different values of p . We have generated input sequences with length 60000 and $p = 0.1, 0.2, 0.3, 0.5, 0.7, 0.8, 0.9$ and these sequences are given to the four compression algorithms as an input. We have repeated the simulation 10 times for each p . Average results can be seen in table 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19.

Table 5.12. Complexities of attacks if LFSR with length L is filtered by different compression algorithms

	EBSG	ABSG	BSG	SSG
Exhaustive Reconstruction	$2^{0.55L}$	$2^{0.66L}$	$2^{0.5L}$	$2^{0.75L}$
Most Probable Case	$2^{0.35L}$	$2^{0.5L}$	$2^{0.25L}$	$2^{0.5L}$
Equations Retrieval	$2^{0.55L}$	$2^{0.66L}$	$2^{0.33L}$	$2^{0.66L}$
Free Binary Decision Diagram	$2^{0.32L}$ to $2^{0.47L}$	$2^{0.54L}$	$2^{0.33L}$ to $2^{0.62L}$	$2^{0.65L}$
Output Rate	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{4}$

Table 5.13: Output distribution with $p = 0.5$

$p = 0.5$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.5	0.5	0.5	0.5
$P(y_i = 1)$	0.5	0.5	0.5	0.5
Output rate	0.5	0.33	0.33	0.25

Table 5.14: Output distribution with $p = 0.1$

$p = 0.1$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.1	0.1	0.82	0.1
$P(y_i = 1)$	0.9	0.9	0.18	0.9
Output rate	0.4	0.33	0.33	0.45

Table 5.15: Output distribution with $p = 0.2$

$p = 0.2$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.2	0.2	0.7	0.2
$P(y_i = 1)$	0.8	0.8	0.3	0.8
Output rate	0.45	0.33	0.33	0.4

Table 5.16: Output distribution with $p = 0.3$

$p = 0.3$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.3	0.3	0.58	0.3
$P(y_i = 1)$	0.7	0.7	0.42	0.7
Output rate	0.475	0.33	0.33	0.35

Table 5.17: Output distribution with $p = 0.7$

$p = 0.7$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.7	0.7	0.58	0.7
$P(y_i = 1)$	0.3	0.3	0.42	0.3
Output rate	0.475	0.33	0.33	0.15

Table 5.18: Output distribution with $p = 0.8$

$p = 0.8$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.8	0.8	0.7	0.8
$P(y_i = 1)$	0.2	0.2	0.3	0.2
Output rate	0.45	0.33	0.33	0.1

Table 5.19: Output distribution with $p = 0.9$

$p = 0.9$	EBSG	ABSG	BSG	SSG
$P(y_i = 0)$	0.9	0.9	0.82	0.9
$P(y_i = 1)$	0.1	0.1	0.18	0.1
Output rate	0.4	0.33	0.33	0.05

When we analyze the simulation results, it is clear that distribution of input sequence is transferred to output sequence by ABSG, SSG and EBSG algorithms, however BSG algorithm does not transfer the distribution of the input sequence to output sequence except uniform distribution of input sequence.

For SSG algorithm, we see that when p decreases output rate increases and when p increases output rate decreases. Output rate of EBSG algorithm is also affected by distribution of input, but in worst case its output rate is still 0.4 and higher than ABSG and BSG.

Regarding the result obtained above, we see that EBSG shows good behaviour for different input distributions, its output rate belongs to interval $[0.4, 0.5]$ for different p values.

Moreover, we should not forget that EBSG is the strongest compression algorithm that provides $1/2$ output rate if input is uniformly distributed, because the other possible compression algorithm that provides the same output rate has the preimage set $C = \{00, 01, 10, 11\}$ with two preimages for output 1 and 0. Equations retrieval attack is easy to apply because length of the preimages are constant, and complexity is only $O(L)$, so we can say that we have increased the resistance against attacks while providing the same output rate

$$\frac{1}{2}.$$

6. CONCLUSION AND FUTURE WORK

In this thesis, we firstly analyzed compression for pseudo-random generation. We see that the optimum code tree for compression is T_2^d with output rate $1/3$. We determine that ABSG is the optimal compression algorithm among existing ones.

We also aim to see the memory bit effect on compression. We present new modified bit search generator named as EBSG which is related to ABSG, using memory bit to increase the output rate from $1/3$ to $1/2$. Experiments and analysis suggest that EBSG produces output sequences with good linear complexity for the input sequences generated by a maximum length LFSR.

We also conclude that using memory bit in compression algorithm does not bring improvement to security, but increases the output rate. Moreover, inserting more than one bit does not provide us any important advantage compared to inserting one bit, so we prefer to insert only one bit.

In future, research in compression for pseudo-random generation can follow way about input-output distribution. New algorithm can be designed to erase the bias of any pseudo-random generator that does not produce 0 and 1 with equal probability.

APPENDIX A: PROOF OF PROPOSITION 3.7

Proposition 3.7. Let (C, f) be a proper encoder such that $f(C)$ is a non-prefix code which does not contain empty word ε . Then, the average expected length of the first word of the decomposition of the input sequence over C is $E(C)$ if the input sequence is chosen uniformly distributed. This word is known with average entropy $E(C) + \Delta(C)$, with $\Delta(C) < -1$.

Proof: For $x \in f(C)$, let define C_x the set of preimages of x in C , and we define

$$P(x) = \sum_{w \in C_x} \frac{|w|}{2^{|w|}}. \text{ Let } \mathbf{y} \text{ be the output sequence corresponding to a randomly chosen}$$

input sequence $\mathbf{s}$. Assume $C(\mathbf{y})$ be the set of words of C such that, for every $w \in C(\mathbf{y})$, the sequence $\mathbf{y}$ begins with w .

Let $P_y(x)$ is the probability that image by the f of first element of C recognized in $\mathbf{s}$ is x , given $\mathbf{y}$. Then we have $\sum_{x \in C(\mathbf{y})} P_y(x) = 1$. Each element of C_x , has probability $\frac{1}{P(x)2^{|w|}}$ to be preimage of x . Thus, the average length of the first element of C recognized in $\mathbf{s}$, given $\mathbf{y}$, is

$$E_y(C) = \sum_{x \in C(\mathbf{y})} P_y(x) \sum_{w \in C_x} \frac{|w|}{P(x)2^{|w|}}.$$

We have chosen output sequences following the uniform distribution on the input sequences, so the average length of the first element of C recognized in a random input sequence knowing the output is $E(C) = \sum_{w \in C} \frac{|w|}{2^{|w|}}$. The average value of E_y for random input sequences is $E(C)$ which is found before.

The entropy on the first element of C recognized in $\mathbf{s}$, given $\mathbf{y}$, is

$$H_y(C) = - \sum_{x \in C(\mathbf{y})} \sum_{w \in C_x} P_y(x) \frac{1}{P(x)2^{|w|}} \log \frac{P_y(x)}{P(x)2^{|w|}}$$

$$\begin{aligned}
&= \sum_{x \in C(y)} \sum_{w \in C_x} \frac{P_y(x)}{P(x) 2^{|w|}} (|w| + \log P(x) - \log P_y(x)) \\
&= E_y(C) + \sum_{x \in C_y} P_y(x) (\log P(x) - \log P_y(x)), \tag{A.1}
\end{aligned}$$

so H_y is equal to the sum of $E(C)$ and value of $\Delta_y(C) = \sum_{x \in C(y)} P_y(x) (\log P(x) - \log P_y(x))$ for

random input sequences. Take a bit b , which is the first bit of y . Then $C(y)$ is included in the subset $C(b^*)$ of those words that are mapped by f to a word whose first bit is b . For

statistical reasons, we have $\sum_{x \in C(b^*)} \frac{1}{2^{|w|}} = 1/2$, which shows that

$$\sum_{x \in C(y)} \frac{1}{2^{|x|}} \leq 1/2. \tag{A.2}$$

We have at least three elements in $f(C)$, so there are some output sequences y such that the inequality (A.2) is strict. Using the equality $\sum_{x \in C_y} P_y(x) = 1$ and the inequality (A.2), we conclude that $\Delta_y \leq -1$ for every output y , the inequality is being strict when that of (A.2) is.

Therefore, the average value of Δ_y for all random input sequences is strictly less than -1 [1]. □

APPENDIX B: PROOF OF PROPOSITION 4.4

Proposition 4.4. Consider a proper encoder (C, f) such that the code tree of C is a T_h^d tree and 0 and 1 have the same number of preimages of each given length. Suppose that C is such that the number of equations linking the preimages of b of length n is the least possible, namely $\phi_b^n = \max(0, n - (D_b^n - 1))$. Then we have :

1. For every $h \geq 2$, the entropy on the length of the preimage of a given output bit

$$b \text{ is } H^{\text{length}} = 2 - 2^{h-d} \left(\frac{3+d-h}{2} \right).$$

2. We have $\overline{\phi_b} = h + 2 - 2^{h-2} - 2^{h-d} + 2^{h-1-d} (2^{h-2} - 1 - d)$, which is equal to 3 for $h = 3$ and $d = \infty$.

Proof:

$$\begin{aligned}
 H_b^{\text{length}} &= - \sum_{n \geq 1} \frac{D_b^n}{2^{n-1}} \log_2 \left(\frac{D_b^n}{2^{n-1}} \right) \\
 &= - \sum_{n=h}^d \frac{2^{h-2}}{2^{n-1}} \log_2 \left(\frac{2^{h-2}}{2^{n-1}} \right) \\
 &= - \sum_{n=h}^d \frac{2^{h-1}}{2^n} (h - n - 1) \\
 &= 2^{h-1} \sum_{n=h}^d \frac{1-h}{2^n} + 2^{h-1} \sum_{n=h}^d \frac{n}{2^n} \\
 &= 2^{h-1} (1-h) \sum_{n=h}^d \frac{1}{2^n} + 2^{h-1} \sum_{n=h}^d \frac{n}{2^n}
 \end{aligned}$$

Lemma A.1:

$$\sum_{i=m}^n \frac{i}{2^i} = m 2^{-m+1} - n 2^{-n} + 2^{-m+1} - 2^{-n+1}$$

Proof:

$$\sum_{i=m}^n \frac{i}{2^i} = m 2^{-m} + (m+1) 2^{-(m+1)} + (m+2) 2^{-(m+2)} + \dots + (n-1) 2^{-(n-1)} + n 2^{-n} = a$$

$$2a = m 2^{-m+1} + (m+1) 2^{-(m)} + (m+2) 2^{-(m+1)} + \dots + (n-1) 2^{-(n-2)} + n 2^{-(n-1)}$$

$$2a - a = m 2^{-m+1} + 2^{-(m)} + 2^{-(m+1)} + \dots + 2^{-(n-1)} - n 2^{-n}$$

$$a = m 2^{-m+1} - n 2^{-n} + \sum_{i=m}^{n-1} 2^{-i} \quad \text{where} \quad \sum_{i=m}^n x^i = \frac{x^{n+1} - x^m}{x-1}$$

$$a = m 2^{-m+1} - n 2^{-n} + 2^{-m+1} - 2^{-n+1}$$

$$\sum_{i=m}^n \frac{i}{2^i} = m 2^{-m+1} - n 2^{-n} + 2^{-m+1} - 2^{-n+1} . \quad \blacksquare$$

$$= (2^{h-1} - 2^{h-1}h) (-2^{-d} + 2^{-h+1}) + 2^{h-1} (h 2^{-h+1} - d 2^{-d} + 2^{-h+1} - 2^{-d+1})$$

$$= h 2^{h-d-1} - 2^{h-d-1} + 1 - h + h - d 2^{h-1-d} + 1 - 2^{h-d}$$

$$H_b^{length} = 2 - 2^{h-d} \left(\frac{3+d-h}{2} \right)$$

$$\overline{\phi}_b = \sum_{n \geq 1} \frac{D_b^n \phi_b^n}{2^{n-1}}$$

$$= \sum_{n \geq 1} \frac{2^{h-2} (n - D_b^n + 1)}{2^{n-1}}$$

$$= 2^{h-2} \sum_{n \geq 1} \frac{n - 2^{h-2} + 1}{2^{n-1}} = 2^{h-1} \sum_{n=h}^d \frac{n}{2^n} + 2^{h-2} \sum_{n=h}^d \frac{1 - 2^{h-2}}{2^{n-1}}$$

Using Lemma A.1

$$\begin{aligned}
&= 2^{h-1} \sum_{n=h}^d \frac{n}{2^n} + (2^{h-1} - 2^{2h-3}) \sum_{n=h}^d \frac{1}{2^n} \\
&= 2^{h-1} \sum_{n=h}^d \frac{n}{2^n} + (2^{h-1} - 2^{2h-3}) (2^{-h+1} - 2^{-d}) \\
&= 2^{h-1} (h 2^{-h+1} - d 2^{-d} + 2^{-h+1} - 2^{-d+1}) + 2^0 - 2^{h-1-d} - 2^{h-2} + 2^{2h-3-d} \\
&= h + 2 - 2^{h-2} - 2^{h-d} + 2^{h-1-d} (2^{h-2} - 1 - d).
\end{aligned}$$

□

REFERENCES

1. Gouget, A. and H. Sibert, *How to Strengthen Pseudo-random Generators by Using Compression*, EUROCRYPT 2006, LNCS 4004, pp.129-146.
2. Gouget, A., H. Sibert, C. Berbain, N. Courtois, N. Debraize and C. Mitchell, *Analysis of the Bit-Search Generator and sequence compression techniques*, Proceedings of FSE'05, LNCS 3557, Springer-Verlag, (2005).
3. Gouget, A. , H. Sibert,, *The Bit-Search Generator, In the State of the Art of Stream Ciphers: Workshop Record, Brugge, Belgium*, October 2004, pages 60-68, (2004).
4. Coppersmith, D. , H. Krawczyk, Y. Mansour, *The Shrinking Generator*, Advances in Cryptology – CRYPTO'93 Proceedings, LNCS 773, Springer-Verlag, (1993), 22-39.
5. Meier, W. , O.Staffelbach, *The Self-Shrinking Generator*, Advances in Cryptology- EUROCRYPT'94 Proceedings, LNCS 950, Springer-Verlag, (1994), 205-214.
6. Kocher, P. , *Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems*, Proceedings of Crypto1996, LNCS 1109, Springer-Verlag, (1996).
7. Krause, M. , *BDD-based Cryptanalysis of Keystream Generators*, In EURO-CRYPT 2002, pp.222-237, LNCS 2332, Springer, (2002).
8. Hell, M. , T.Johansson, *Some attacks on the Bit-Search Generator* Proceedings of FSE'05, LNCS 3557, Springer-Verlag, (2005).
9. Kessler, I. , H. Krawczyk, *Minimum Buffer Length and Clock Rate for the Shrinking Generator Cryptosystem*, IBM Research Report, RC 19938 (88322), (1995)

10. *An security examination of the self-shrinking generator*, Circencester, UK, December 1995.
11. Berstel, J. , D. Perrin, *Theory of Codes*, Academic Press, (1985).
12. Babbage, S. , C. De Canniere, J.Lano, B. Preneel, and J.Vanderwalle. *Cryptanalysis of Sober-t32*. In T.Johansson, editor, Fast Software Encryption 2003.
13. Barkan, E. , E. Biham, and N. Keller. *Instant ciphertext only cryptanalysis of GSM encrypted communication*. In D. Boneh, editor, Advances in Cryptology-CRYPTO 2003, volume 2729 of Lecture Notes in Computer Science, pages 600-616. Springer Verlag, 2003.
14. Massey, J.L. , *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory,15:122-127,1969.
15. Siegenthaler, T. , *Correlation-immunity of non-linear combining functions for cryptographic applications*. IEEE Transactions on Information Theory, 30:776-780,1984.
16. Key, E. , *An analysis of the structure and complexity of nonlinear binary sequence generators*. IEEE Transactions on Information Theory, 22:273-736,1976.
17. Ekdahl, P. , *On LFSR based Stream Ciphers Analysis and Design*, Ph.D Thesis, Lund University, 2003.
18. Günther, C.G. , *Alternating step generators controlled by de Bruijn sequences*. In D. Chaum and W.L. Price, editors, Advances in Cryptology-EUROCRYPT'87, volume 304 of Lecture Notes in Computer Science, pages 91-103. Springer-Verlag,1988.
19. Courtois, N. ,: *Higher Order Correlation Attacks, XL Algorithm and Cryptanalysis of Toyocrypt*, ICISC 2002, November 2002, Seoul, Korea, LNCS 2587, Springer, 2002.

20. Courtois, N. , Willi Meier: *Algebraic Attacks on Stream Ciphers with Linear Feedback*, Eurocrypt 2003. Warsaw, Poland, LNCS 2656, pp. 345-359, Springer.
An extended version is available at <http://www.minrank.org/toyolili.pdf>

21. Armknecht, F. .: *A Linearization Attack on the Bluetooth Key Stream Generator*, Available on <http://eprint.iacr.org/2002/191/>. 13 December 2002.

22. Shamir, A. , Jacques Patarin, Nicolas Courtois, Alexander Klimov : *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations*, Eurocrypt'2000, LNCS 1807, Springer, pp. 392-407.

23. Shamir, A. , Aviad Kipnis: *Cryptanalysis of the HFE Public Key Cryptosystem*, Proceedings of Crypto '99, Springer LNCS 1666, 1999, pp. 19-30.

24. Rueppel, R.A. , *Analysis and Design of Stream Ciphers*, Springer-Verlag,(1986).

25. Armknecht, F. , M.Krause, *Algebraic Attacks on Combiners with Memory*, Advances in Cryptology – CRYPTO'03 Proceedings, LNCS 2799, Springer-Verlag, (2003), 162-176.

26. Touretzky, D.S. , *Basics of Information Theory*, November 2004.