UNIVERSITY OF CAPE TOWN

A FAST CORRELATION ATTACK IMPLEMENTATION

Honours Project 2011

Azhar Desai

SUPERVISORS
Dr Anne Kayem
Dr Christine Swart

Abstract

Stream ciphers are used to encrypt data on devices that have limited resources or where time-sensitive encryption is required. Certain stream ciphers use a shared encryption key to produce a pseudo-random sequence called a keystream, which is combined with the plaintext for encryption. This report studies Linear Feedback Shift Register (LFSR) sequences and how they are used in keystream generators called combination generators. Certain combination generators produce keystreams that are correlated to an LFSR sequence used in creating the keystream. A statistical model is developed that allows a fast correlation attack to exploit this vulnerability to recover the correlated LFSR sequence faster than an exhaustive search over all the possible LFSR sequences. An implementation is presented of an algorithm for a fast correlation attack that successfully recovers the correlated sequence when the number of LFSR taps is less than or equal to 6, and the LFSR length is below 16. The results indicate that a correlation of the keystream to an LFSR sequence should be avoided in the design of stream ciphers to prevent this type of correlation attack.

Category	Allocated Marks
1 Requirement Analysis and Design	0
2 Theoretical Analysis	25
3 Experiment Design and Execution	0
4 System Development and Implementation	5
5 Results, Findings and Conclusion	20
6 Aim Formulation and Background Work	10
7 Quality of Report Writing and Presentation	10
8 Adherence to Project Proposal and Quality of Deliverables	10
9 Overall General Project Evaluation	0
Total	80

Contents

List of Figures

List of Tables

Glossary

characteristic polynomial Polynomial associated with a linear recurrence; in this case of the LFSR recurrence.

ciphertext The result obtained from the encryption of the plaintext.

combination generator A keystream generator combines the output of several LFSR with a boolean function on each clocking to produce one keystream bit.

feedback polynomial The same as characteristic polynomial.

keystream A bit stream that is combined with plaintext to produce ciphertext.

LFSR Linear Feedback Shift Register.

linear recurrence A recursively defined equation that defines a sequence given appropriately many initial terms. The terms (of the sequence) depend on linear combinations of previous terms.

parity check An equation used to test whether a given bit was produced by a specified LFSR.

plaintext The original message before it is encrypted.

Chapter 1

Introduction

One aim of symmetric key cryptography is to provide a means to ensure confidential exchange of information between two parties, both of whom possess the same shared secret number or binary string called a key. Where both parties have limited computationally power at their disposal, a class of symmetric key cryptosystems called stream ciphers are used. Examples of stream ciphers include A5/1, which is used in Global System for Mobile Communications (GSM) for cellphone communications and E0, the stream cipher used in the Bluetooth protocol. Fast correlation attacks have been made on E0. See for example [4].

Additive stream ciphers encrypt the message, the plaintext, for confidentiality by generating a pseudo-random sequence, the keystream, from the key and combining it with the plaintext to produce the ciphertext. The intended recipient of the ciphertext knows the key and produces the same keystream, which is used to retrieve the plaintext from the ciphertext. Both extracting the plaintext and combining it with the cipher text, is done using bitwise addition modulo 2 (or equivalently XOR).

Fast correlation attacks work on certain keystream generators called combination generators. Combination generators use Linear Feedback Shift Registers (LFSRs), which produce a deterministic sequence of bits from an initial bit string. The combination generator combines the output of several LFSR. When used in a stream cipher, the key is the concatenation of the initial states of all the LFSR.

However, some combination generators produce keystream sequences that is correlated to the output sequence of one of its constituent LFSR (as opposed to being a good mix of all of them.) For a given keystream, if enough bits are the same as the LFSR sequence, we can infer of the LFSR sequence and find out information about the key used to initialise this combination generator.

Fast correlation attacks exploit this flaw by using the first N bits of the keystream, and the parameters of the correlated LFSR, the algorithm aims to find the sequence of the correlated LFSR. Since this sequence includes some of the data used to initialise the combination generator, partial knowledge of the key is gained.

This project presents an implementation of a fast correlation attack. The implemented algorithm produces the LFSR sequence, and the range of the parameters under which this is successful is documented.

The background section of this report first covers the preliminary material needed to discuss fast correlation attacks, followed by related work on generating parity check equations. The rest of the report is structured as follows. In Chapter 3 the statistical model for the

algorithm, the algorithm, and implementation is detailed. Finally the results are discussed followed by some concluding remarks. The appendix includes a theorem used in the report and describes another algorithm studied but not used in the implementation.

Chapter 2

Background

2.1 Preliminaries

This section introduces stream ciphers and explains the components necessary in order to precisely formulate the fast correlation attack problem.

2.1.1 Stream Ciphers

A stream cipher generates a pseudo-random sequence, the keystream, from a key and combines it with the plaintext to produce the cipher text. Most stream ciphers are additive stream ciphers. These combine the plaintext and the keystream using bitwise XOR. To decrypt, a similar procedure is followed by the receiver of the ciphertext. The keystream is produced from the shared key and used to extract the plaintext.

In the case of additive stream ciphers, this extraction can just be done by XOR of the ciphertext and keystream. This works because the sender encrypts by computing $C = P \oplus K$ where P is the plaintext and K is the keystream. The receiver then uses the shared key to compute K just as the sender does, and then computes $C \oplus K$ which is the plaintext because

$$C \oplus K = (P \oplus K) \oplus K = P \oplus (K \oplus K) = P \oplus 0 = P. \tag{2.1}$$

In a known-plaintext attack, the cryptanalyst is assumed to know a certain plaintext ciphertext pair, (C, P), but not the key. Note that now the cryptanalyst can find the keystream by an XOR of P and C since

$$C \oplus P = (P \oplus K) \oplus P = K \oplus (P \oplus P) = K. \tag{2.2}$$

This fast correlation attack applies to certain stream ciphers in the circumstance, such as a known-plaintext attack, where N bits of the keystream are known. This is used to gain knowledge of the key. A common keystream generator is called the combination generator. This combines the output of a few components, called LFSR, with a boolean function to produce the next bit of output. Correlation attacks work when the boolean function is poorly chosen.

2.1.2 Linear Feedback Shift Registers

Before describing a combination generator, we need to describe a common stream cipher component called a Linear Feedback Shift Register (LFSR). This is a register that is L bits

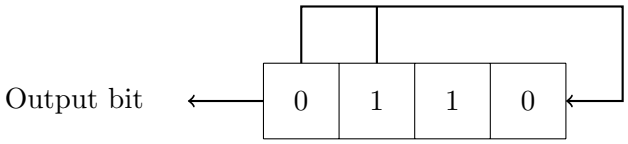

Figure 2.1: A LFSR of length 4

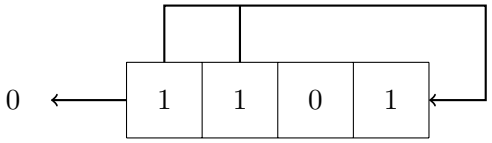

Figure 2.2: The LFSR in Figure 2.1 clocked once

long. The state of the register, $\underline{S_t} = (s_t, s_{t+1}, \ldots, s_{t+L-1})$, is the contents of at time t. At $t = 0$ the LFSR register is filled with an initial state, and then in each clock cycle it is clocked to produce one output bit. To clock the register, the contents of the register are shifted to left by one bit. The leftmost bit becomes the next output bit and the rightmost bit is filled with the XOR of a selection of the previous L bits.

Example 1. *Figure 2.1 shows an example LFSR of length 4. The state shown is the initial state, and table 2.1 demonstrates the bits produced in the sequence. The two left most boxes are summed to produce the next bit in the register. These are called taps. Figure 2.2 shows the same LFSR after being clocked once, with* 0 *produced as the output bit.*

Clocks	Output bit	Register state
0	-	0 1 1 0
1	0	1 1 0 1
2	1	1 0 1 0
3	1	0 1 0 1
4	0	1 0 1 1

Table 2.1: Table of the first four output values of the LFSR in Figure 2.1 after successive clocks

Alternatively, we can describe the sequence of bits, s_n, produced by the LFSR by a linear recurrence relation

$$s_n = s_1 s_{n-1} + c_2 s_{n-2} + \ldots + c_L s_{n-L} \tag{2.3}$$

The non-zero c_i are the taps or the connection coefficients. All addition is modulo 2, which is equivalent to XOR.

Example 2. *The recurrence for the LFSR in Figures 2.1 and 2.2 is*

$$s_n = s_{n-3} + s_{n-4} \tag{2.4}$$

Equation 3.1 can be rewritten in the form

$$\sum_{i=0}^{L} c_i s_{t-i} = 0$$

where $c_0 = 1$. Associated with this recurrence is its characteristic or feedback polynomial

$$C(X) = \sum_{i=0}^{L} c_i X^i$$

The characteristic polynomial is an element of $\mathbb{Z}_2[X]$, which means that the coefficients come from the binary field so are either 1 or 0.

Example 3. *The characteristic polynomial of the recurrence in equation 2.4 is*

$$C(X) = 1 + X^3 + X^4.$$

Some polynomials $f(X) \in \mathbb{Z}_2[X]$ of degree L have the property that they are primitive. This means that the smallest positive integer e such that $f(X)$ divides $X^e - 1$ is $e = 2^L - 1$. If the characteristic polynomial of an LFSR sequence is primitive, then the LFSR sequence has a period of $2^L - 1$. (The LFSR sequence repeats itself after $2^L - 1$ bits.) The output bit depends on the the previous L bits of the sequence, in other words, the contents of the register just before the bit is first inserted into the register. This means that every LFSR cycles through every possible non-zero state. (If the register was filled with only zeroes then, every subsequent state would remain that way.) The LFSR sequence is said to be maximum-length if it has a period of $2^L - 1$. Maximal-length LFSR sequences have the property have two useful properties. The first is that in one period, the number of 1's in the sequence is one greater than the number of 0's. The second is that occurrences of patterns of fixed length less than L are almost uniform. For a detailed treatment of this see [7].

The LFSR sequences used for the development of the statistical model are required to be maximal-length. Unless otherwise stated, all of the LFSR sequences in this report are chosen to be maximal-length. This has the consequence that the number of taps, $t = |\{c_i | c_i = 1\}| - 1$, always has to be even. If it were odd, then since it is a maximal length sequence, the state consisting only of 1's would occur. The XOR of an odd number of 1's is always 1. Thus, every subsequent state would remain the same, contradicting the fact that the sequence is maximal. For the rest of the report, it is consequently assumed that the number of taps is even.

Before ending this subsection a final representation of the LFSR needs to be introduced. The transition from the LFSR state $\underline{S_t} = (s_t, \ldots, s_{t+(L-1)})$ to the state $\underline{S_{t+1}} = (s_{t+1}, \ldots, s_{t+L-1}, \sum_{i=1}^{L} c_i s_{t+L-i})$ can represented using the following matrix

$$A = \begin{pmatrix} 0 & 0 & \cdots & c_L \\ 1 & 0 & \cdots & c_{L-1} \\ 0 & 1 & \cdots & c_{L-2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & c_1 \end{pmatrix} \tag{2.5}$$

resulting in the equation

$$\underline{S_{t+1}} = \underline{S_t} A. \tag{2.6}$$

Example 4. *The transition between the states the LFSR shown in Figures 2.1 and 2.2 can be written as*

$$(1,1,0,1) = (0,1,1,0)\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{2.7}$$

2.1.3 Parity Check Equations

Every bit of the LFSR sequence satisfies the linear combination of the previous L bits. Any previous bit in the equation can be replaced by linear combination of earlier bits, resulting in a new linear equation. For a given LFSR sequence, there are many such linear equations that the sequence satisfies. The example below demonstrates this how one new linear equation can be obtained from the one defining the sequence.

Example 5. *Consider the LFSR sequence defined by the recurrence in equation 2.4. Replacing each term on the right by the linear combination that defines it yields an equation with only two terms. Note that the new recurrence is "wider" in the sense that the terms involved appear further apart in the sequence.*

$$\begin{aligned} s_n &= s_{n-3} + s_{n-4} \\ &= (s_{n-6} + s_{n-7}) + (s_{n-7} + s_{n-8}) \\ &= s_{n-6} + s_{n-8} \end{aligned} \tag{2.8}$$

The term $2s_{n-7} = 0$ since the equations are in characteristic 2. (The coefficients are used modulo two.)

These equations are called parity check equations. The name is used because the LFSR sequence will satisfy the equations so they can be used to see if a LFSR sequence is correct. More of these can be found by considering the recurrences associated with polynomial multiples of the characteristic polynomial. The algorithms in detailed in this report will consider only polynomial multiples obtained by repeatedly squaring the characteristic polynomial. This is detailed in section 3.2.1.

Example 6. *Continuing with the same LFSR sequence with characteristic polynomial $C(X) = 1 + X^3 + X^4$, the square of it is*

$$C(X)^2 = 1 + X^6 + X^8. \tag{2.9}$$

The recurrence associated with this squared polynomial is the recurrence found earlier in equation (2.8).

We define the weight of the parity check equation to be the number of terms involved in the equation. Equivalently, this is the number of terms in the associated characteristic polynomial including the constant term. The parity check equations used for the iterative decoding algorithm in this report all have constant weight. This turns out to be important in keeping the algorithm's running time efficient.

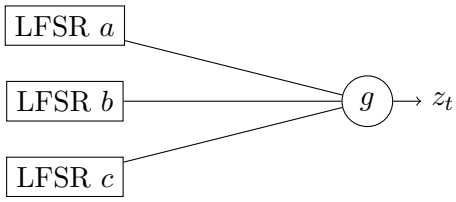

Figure 2.3: The Geffe combination generator: the output of three LFSR are combined, using the boolean function g to produce the keystream.

a_t	b_t	c_t	$g(a_t, b_t, c_t)$
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

Table 2.2: Correlation of the Geffe combining function to its first LFSR.

2.1.4 Combination Generator

Using these LFSR we are now able to define a keystream generator called the Geffe generator. This is an example of a combination generator, that this report's implementation is tested against. The attack, however, works for all combination generators that suffer from the correlation.

Example 7 (Geffe Keystream Generator)**.** *The Geffe Generator consists of any three LFSR and the boolean combining function g. The keystream bit, z_t, at time t is given by*

$$z_t = g(a_t, b_t, c_t) = a_t b_t + b_t c_t + c_t \tag{2.10}$$

where a_t, b_t and c_t are the output bits of the three LFSR after t clocks.

In general a combination generator may have any number of LFSR as the inputs to any nonlinear boolean function.

Fast correlation attacks require a flaw in the combining function of the combination generator. If the output of the generator is correlated to one of its constituent LFSR, the observed keystream could be used to deduce sequence produced the LFSR. The LFSR are chosen to be maximal so that the keystream sequence has the longest period possible for the lengths of the input LFSR. The inputs to g are thus balanced which allows the function to be treated as if the correlated input bit is passing through g having a constant probability of being complemented. Once the cryptanalyst has recovered the correlated LFSR sequence, she knows part of the key used to initialise the combination generator.

Example 8 (Correlation of the Geffe combining function)**.** *Table 2.2 shows the possible outputs of g, and the correlation: if the input is chosen at random we then with probability* $\frac{6}{8} = \frac{3}{4}$ *we have* $g(a, b, c) = a$*. A similar correlation holds for the third input c.*

2.1.5 Correlation problem

The keystream sequence, $\mathbf{z}$, is correlated to the LFSR sequence $\mathbf{a}$ if there is a constant probability p, such that

$$p = \text{Prob}(z_n = a_n) > 0.5 \tag{2.11}$$

This is not a major restriction since if $p < 0.5$ the keystream sequence can be complemented. The correlation probability is then $1 - p > 0.5$.

The problem, more precisely, is to use the keystream of length N of a correlated combination generator to recover the correlated LFSR sequence. The cryptanalyst is assumed to know the the length, L, the taps and the correlation probability of the correlated LFSR, $\mathbf{a}$. A trivial solution is do an exhaustive search over all $2^L - 1$ initial states of $\mathbf{a}$. For each initial state the correlation between the resulting LFSR keystream and the observed keystream is checked by counting the fraction of the bits for which $a_t = z_t$. However, fast correlation attacks aim to recover the sequence faster than an exhaustive search.

2.2 Related Work

The original correlation attack by Siegenthaler in [11] is faster than an exhaustive search of all the initial states of the constituent LFSR at once. Each LFSR can attacked in turn in time proportional to its exhaustive search.

Fast correlation attacks aim to recover the correlated LFSR sequence in faster time than the exhaustive search. The original fast correlation attacks were proposed by Meier and Staffelbach in [5, 6]. Their modified exhaustive search algorithm (Algorithm A) is described in the appendix, and their second algorithm, the iterative decoding algorithm (Algorithm B) is described in later sections and implemented.

Both methods rely on having parity check equations available for each bit. Having more allows a sharper distinction between keystream bits, z_t, where $z_t = a_t$ and where $z_t \neq a_t$. This section discusses alternate methods of finding parity check equations. Three techniques are discussed: using the feedback polynomial, using the matrix representation of the LFSR and using a coding theory approach.

2.2.1 Polynomial Multiples

We'll denote the characteristic polynomial of the correlated LFSR sequence as $C(X)$. In [5, 6], these polynomials are obtained by repeatedly squaring the feedback polynomial, $C(X)^j$ where $j = 2^i$, in order to keep the weight of parity checks constant. This is explained in Section 3.2.1.

Instead of only using repeated squares of the feedback polynomial we can consider all appropriate polynomials $Q(X)C(X)$ of degree less than N and whose constant term is 1. Canteaut and Trabbia in [1] present an algorithm for finding all the linear equations involving d bits of the sequence by considering all such polynomials of weight d.

2.2.2 Using the Matrix Representation of the LFSR

Mihaljevic and Golic propose using the matrix representation of the LFSR to generate more parity check equations in [8]. Recall from the preliminary section on LFSR, we can represent the between two consecutive states by a matrix, A. Using this we can write

$$\underline{S_t} = \underline{S_k} A^{t-k}. \tag{2.12}$$

where S_t denotes the row vector of the t^{th} state. The initial state of the LFSR is denoted S_0. If $S_t = (s_t, s_{t+1}, \ldots, s_{t+L})$ the values $k = 0, 1, 2, \ldots, t-1$ yield t more equations that S_t has to satisfy.

Example 9. *This example uses the LFSR sequence $s_t = s_{t-1} + s_{t-4}$ and its matrix representation, A. Consider the case where $k = t - 2$. Equation 2.12 in this case is*

$$S_t = S_{t-2} A^2 \tag{2.13}$$

and substituting the variables we have

$$(s_t, s_{t+1}, s_{t+2}, s_{t+3}) = (s_{t-2}, s_{t-1}, s_t, s_{t+1}) \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}. \tag{2.14}$$

For the above we get the following parity check equations

$$\begin{aligned} s_{t+2} &= s_{t-2} + s_{t+1} \\ s_{t-5} &= s_{t-5} + s_{t-4} + s_{t-2}. \end{aligned} \tag{2.15}$$

The first parity check is ignored since it is equivalent to the recurrence that defined the sequence, however the second one is new and can be used.

2.2.3 Correlation attacks modelled as a decoding problem

Finding the correlated LFSR from the keystream can be treated as a decoding problem. The code words are all possible sequences of a fixed length, N, that the original LFSR produces. This sequence is transmitted over a Binary Symmetric Channel (BSC) where each bit has a probability, p, of flipping to give the keystream sequence. For the LFSR a in equation 3.1, the output forms a $[N, L]$ linear code, $\mathcal{C}$. The output sequence of the LFSR $(a_1, a_2, \ldots a_N)$ is an element of $\mathcal{C}$. These codewords are said to have low weight if they have few 1 digits. Details of linear codes can be found in [10], and [3] gives a quick overview of how they apply to the correlation attack problem.

The above treatment allows a known problem in coding theory to be used to solve this fast correlation attack problem. Chepyzhov and Smeets in [2] present a method using a dual codes. Consider the possible output sequences of length N of an LFSR of length l as an $[l, N]$ linear code $\mathcal{C}$. The dual code of $\mathcal{C}$ is a code $\mathcal{H}$ consisting of codewords orthogonal to all the ones in $\mathcal{C}$. This means that for each $\mathbf{h} \in \mathcal{H}$

$$\mathbf{c} \cdot \mathbf{h}^{\mathrm{T}} = 0 \quad \forall \mathbf{c} \in \mathcal{C}. \tag{2.16}$$

Example 10 (Deriving a parity check equation from a codeword)**.** *Consider the LFSR sequence of length* 4 *as a codeword* $(a_0, a_1, a_2, a_3) \in \mathcal{C}$*. Assume we have a dual code for* C *containing the codeword* $(0, 1, 0, 1)$*. Equation 2.16 in this case is*

$$\begin{aligned} (a_0, a_1, a_2, a_3) \cdot (0, 1, 0, 1)^T &= 0 \\ (a_0, a_1, a_2, a_3) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} &= 0 \\ a_1 + a_3 &= 0 \end{aligned} \tag{2.17}$$

The above example shows parity check equations of low weight can be found, by searching for codewords of low weight in dual code.

Meier-Staffelbach's algorithm, using the squares of the feedback polynomial, only works for a small number of taps (less than ten). Milahjevic and Golic's algorithm, using the matrix representation of the LFSR, works with a much larger number of feedback taps. However, [3] points out that this method is requires much larger observed sequence for increased lengths of the LFSR.Chepyzhov and Smeets' algorithm, using the dual code method, works for an arbitrary number of taps. Also, it is shown in [2] that the complexity of the initial state recovery grows less than exponentially, even if the number of taps increases linearly with the length of the LFSR. Lastly, Canteaut and Trabbia's algorithm in [1] finds all appropriate polynomial multiples of the feedback polynomial that result in parity check equations of weight 4 and 5. This doesn't require that the feedback polynomial have low weight to begin with.

Chapter 3

Methodology

The approach taken to the problem here is that of Meier and Staffelbach. To find the LFSR sequence $\mathbf{a} = (a_1, \ldots, a_N)$ correlated to the N-bit output sequence $\mathbf{z} = (z_1, \ldots, z_N)$ from a combination generator, a statistical model of the correlation problem is used to develop two algorithms. The model provides a means of calculating the probability that a keystream bit is the same as the corresponding bit in the LFSR sequence. The two other parameters used in the model are p, the correlation probability, and L, the length of the LFSR. The first section of this chapter outlines the statistical model. The second section outlines the iterative decoding algorithm which relies on the calculations of the model.

3.1 Statistical Model

This section develops the model that allows us to calculate the probability, $p_i = \text{Prob}(z_i = a_i)$, that a given keystream bit is the same the corresponding bit in the LFSR sequence. The first subsection motivates the later development of the model. The second subsection provides a means of updating the probability p_i when as it is initially, the correlation probability is a constant p. The algorithm will iterate the updating of probabilities several times before re-setting the probabilities to the original constant correlation probability. On the first iteration the probability is constant for all the $p_i = p$. However for later iterations, the p_i will differ. The final subsection generalises the method for updating the p_i for later iterations when those probabilities are not all equal.

3.1.1 Motivation

The observed keystream sequence produced by the combination generator is $\mathbf{z} = (z_1, \ldots, z_N)$. Recall that the correlated LFSR sequence, $\mathbf{a}$, is generated by the linear recurrence

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_L a_{n-L}. \tag{3.1}$$

The number of taps, t, is the number of non-zero coefficients in the set $\{c_1, \ldots, c_L\}$. Since we are working with addition modulo 2, a_n can be added to both sides of the above equation to get the parity check equation

$$\sum_{0 \le j \le L, c_j \neq 0} a_{n-j} = 0. \tag{3.2}$$

Let's consider a fixed bit a_i. By using $t+1$ different values of n in equation (3.2) we get $t+1$ equations featuring a_i. The example belows demonstrates it for a recurrence.

Example 11. *Consider the recurrence $a_n = a_{n-2} + a_{n-5}$. We obtain the following three equations featuring a_i:*

$$\begin{aligned} \mathbf{a_i} + a_{i-2} + a_{i-5} &= 0 \\ a_{i+2} + \mathbf{a_i} + a_{i-3} &= 0 \\ a_{i+5} + a_{i+3} + \mathbf{a_i} &= 0 \end{aligned} \tag{3.3}$$

Section 3.2.1 will detail how to find more such equations but for now let's assume there are m such equations

$$\begin{aligned} L_1 &= a_i + b_1 = 0 \\ L_2 &= a_i + b_2 = 0 \\ &\vdots \\ L_m &= a_i + b_m = 0 \end{aligned} \tag{3.4}$$

where each b_k is the sum of exactly t other terms of the sequence. These equations will become our parity check equations when the the bits of the sequence $\mathbf{a}$ are replaced by the corresponding bits of the sequence $\mathbf{z}$, giving

$$\begin{aligned} L_1 &= z_i + y_1 \\ L_2 &= z_i + y_2 \\ &\vdots \\ L_m &= z_i + y_m \end{aligned} \tag{3.5}$$

where the L_k are not necessarily 0 and each y_k is the corresponding sum of t variables from $\mathbf{z}$. Whether L_k is 0 depends on whether $z = a$ and whether which do not necessarily satisfy all the parity check equations that bits of $\mathbf{a}$ would. The replacement of variables is the key idea that motivates the statistical model outlined in the next section.

3.1.2 The Model

This subsection outlines a statistical model by which an expression can be derived to compute the the updated $p_i = \text{Prob}(z_i = a_i)$, once learning how many parity check equations z_i satisfies. Importantly this model assumes initially $p_i = p$, the value of which is then updated depending on how many parity check equations z_i satisfies. We introduce the set of binary random variables $A = \{a, b_{11}, b_{12}, \ldots, b_{1t}, b_{21}, b_{22}, \ldots, b_{2t}, \ldots b_{m1}, \ldots b_{mt}\}$. These random variables satisfy the equations

$$\begin{aligned} a + b_{11} + b_{12} + \ldots + b_{1t} &= 0 \\ a + b_{21} + b_{22} + \ldots + b_{2t} &= 0 \\ &\vdots \\ a + b_{m1} + b_{m2} + \ldots + b_{mt} &= 0 \end{aligned} \tag{3.6}$$

Analogously we introduce the set of binary random variables for bits of $\mathbf{z}$, $Z = \{z, y_{11}, y_{12}, \ldots, y_{1t}, y_{21}, y_{22}, \ldots, y_{2t}, \ldots y_{m1}, \ldots y_{mt}\}$. The two sets of random variables are related via the correlation probability p by satisfying

$$\text{Prob}(a = z) = \text{Prob}(b_{ij} = y_{ij}) = p \tag{3.7}$$

Note that the expressions for z analogous to the equations in 3.6

$$z + y_{i1} + y_{i2} + \ldots + y_{it} \tag{3.8}$$

do not necessarily sum to zero. In order to find out when they do sum to zero several random variables derived from the above are introduced.

For each equation in (3.6), we define a new random variable, $b_i = b_{i1}+b_{i2}+\ldots+b_{it}$, to be the sum of all the bits except a in the equation. Similarly for z we introduce $y_i = y_{i1}+y_{i2}+\ldots+y_{it}$. Also, as in equation (3.5), each parity check is written as $L_k = z_i + y_k$ for $k = 1, 2, \ldots, m$. These are the parity check equations involving a fixed bit z_i. The k^{th} parity check equation is satisfied if $L_k = 0$.

3.1.3 Updating Probabilities

Initially, since there is a constant correlation between corresponding bits b_{ij} and y_{ij},[1]

$$s = \text{Prob}(y_k = b_k) \tag{3.9}$$

can be computed independently of i.

$$\begin{aligned} s(p, 1) &= p \\ s(p, t) &= ps(p, t-1) + (1-p)(1-s(p, t-1)). \end{aligned} \tag{3.10}$$

(The event $b_k = y_k$ occurs when either $b_{kt} = y_{kt}$ and the sum of the remaining variables are the same, or $b_{kt} \neq y_{kt}$ and the sum of the remaining variables are different.)

An expression can now be derived for the probability that $z_i = a_i$ if exactly h of the m parity check equations are satisfied. Assume without loss of generality that the first h relations are satisfied and the rest are not. We denote this event as H_i, meaning that $L_1 = L_2 = \ldots = L_h = 0$ and $L_{h+1} = L_{h+2} = \ldots = L_m = 1$ for a given bit z_i.

Using the law of total probability, the probability that H_i occurs can be expressed as

$$\text{Prob}(H) = \text{Prob}(z_i = a_i \text{ and } H_i) + \text{Prob}(z_i \neq a_i \text{ and } H_i) \tag{3.11}$$

Recalling equation (**??**), if $z_i = a_i$ for $L_1 = L_2 = \ldots = L_h = 0$, then $b_k = y_k$ for $k \leq h$. For the other unsatisfied relations, $k > h$, it needs to be that $b_k \neq y_k$. The case of $z_i \neq a_i$ works analogously. It follows that

$$\text{Prob}(H_i) = ps^h(1-s)^{m-h} + (1-p)(1-s)^h s^{m-h}. \tag{3.12}$$

[1]Since $\text{Prob}(b_{ij} = y_{ij}) = \text{Prob}(z_k = a_k) = p$ for some k.

The conditional probability that $z_i = a_i$ given that exactly h parity check equations are satisfied can now be computed as

$$\begin{aligned}\text{Prob}(z_i = a_i | H_i) &= \frac{\text{Prob}(z = a \text{ and } H_i)}{\text{Prob}(H_i)} \\ &= \frac{ps^h(1-s)^{m-h}}{ps^h(1-s)^{m-h} + (1-p)(1-s)^h s^{m-h}}.\end{aligned} \tag{3.13}$$

To end this subsection the basic idea of the method can be laid out in terms of the statistical model. After fixing a bit z_i, by observing the keystream, we observe the outcome of the z_i and the variables y_{ij}. The derived variables y_i and ultimately L_i can be computed, and the number of L_i that equal to 0 can be counted. We cannot observe the corresponding bits of the LFSR sequence a_i and b_{ij} which we are trying to find, but using equation (3.13) we can compute an updated probability, $p_i^\star$ that $z_i = a_i$. If this probability is very small, then it is likely that $z_i \neq a_i$ which means that z_i is the complement of a_i.

3.1.4 Updating Probabilities (General Case)

In the previous subsection updating the probability relies on the prior probabilities all being constant (i.e. $p_i = p$ for all i). The iterative decoding algorithm will often repeat the process of updating the p_i. In the later iterations the prior probabilities are no longer constant. The above analysis needs to be generalised.

A generalisation of $s(p,t)$ in equation (3.10) is required. This value can no longer be computed independently of i. For a bit z_i we define the function s_i that takes as input the probabilities of the other t bits of parity check equation L_k for z_i. Assuming the t other bits involved in L_k have probabilities $p_1, p_2, \ldots, p_t$ the new expression is

$$\begin{aligned}s_i(p_1, \ldots, p_t, t) &= p_t s_i(p_1, \ldots, p_{t-1}, t-1) + (1-p_t)(1 - s_i(p_1, \ldots, p_{t-1}, t-1)) \\ s_i(p_1, 1) &= p_1.\end{aligned} \tag{3.14}$$

This generalisation needs to carry over to other formulas, leading us to a generalisation of equation (3.13) for updating $\text{Prob}(z_i = a_i)$. Assuming z_i satisfies h of m parity check equations. Letting $s_{i,k}$ be the corresponding s_i value given the probabilities of the other t bits in the parity check equation, L_k, for z_i we have

$$\begin{aligned}\text{Prob}(z_i = a_i | H_i) &= \frac{\text{Prob}(z_i = a_i \text{ and } H_i)}{\text{Prob}(H_i)} \\ &= \frac{\text{Prob}(z_i = a_i \text{ and } H_i)}{\text{Prob}(z_i = a_i \text{ and } H_i) + \text{Prob}(z_i \neq a_i \text{ and } H_i)}\end{aligned} \tag{3.15}$$

where

$$\text{Prob}(z_i = a_i \text{ and } H_i) = p_i \prod_{k=0}^{h} s_{i,k} \prod_{k=h+1}^{m} (1 - s_{i,k}) \tag{3.16}$$

and

$$\text{Prob}(z_i \neq a_i \text{ and } H_i) = (1-p_i) \prod_{k=0}^{h} (1 - s_{i,k}) \prod_{k=h+1}^{m} (s_{i,k}). \tag{3.17}$$

The results of this entire section allow us to use the statistical model to update each p_i depending on how many parity check equations z_i satisfies when the prior values are each

$p_i = p$. Additionally, when repeating this process of updating the probabilities, the method of updating has been generalised to work where the p_i are not all the same.

3.2 Algorithm

Both the algorithm discussed in this section and the Modified Exhaustive Search algorithm in the appendix were proposed by Meier and Staffelbach in [5, 6]. Both algorithms use the same method for generating parity check equations that is sketched out in the next subsection. The Modified Exhaustive Search Algorithm uses the digits with high probability of being correct, to speed up the search for the correct LFSR sequence. The Iterative Decoding Algorithm repeatedly complements digits with low probability of being correct aiming to reproduce the correlated LFSR sequence.

3.2.1 Finding Parity Check Equations

Section 2.1.3 discusses the fact that an LFSR sequence, **a**, satisfies any recurrence associated with a polynomial multiple of the its characteristic polynomial.

The linear recurrences we use as parity checks are the recurrences associated with the polynomials obtained by repeatedly squaring the feedback polynomial. These are of the form $C(X)^{2^i}$.

The reason for using polynomial multiples obtained by squaring is that for polynomials with binary coefficients

$$C(X)^{2^i} = C(X^{2^i}). \tag{3.18}$$

A generalisation of this result is proved in Appendix A.2. Using p = 2 in equation (A.5), and applying the theorem i times (once for each squaring) yields the desired result. The consequence is that the weight of the parity checks, in other words the number of terms in the polynomials remain constant. This is important to keeping the running time of the algorithm down.

Since parity check equations obtained by squaring will be applied to keystream sequences of finite length, only finitely many of these squared polynomials can be used. The higher the degree of the polynomial the further apart the digits used in the equation appear in the sequence.

Example 12. *Consider a case where* 8 *bits of the keystream sequence is observed and the LFSR has a feedback polynomial*

$$C(X) = 1 + X + X^6$$

This corresponds to the equation

$$a_n = a_{n-1} + a_{n-6}.$$

Assuming the sequence observed is $z_0, \ldots z_7$*, the digits that can be used correspond to* n *being* 6 *or* 7*. However the square of this polynomial,* $C(X) = 1 + X^2 + X^{12}$ *has degree* 12*, corresponding to the recurrence*

$$a_n = a_{n-2} + a_{n-12}$$

so cannot be used at all.

To conclude this section we need an estimate of the average number of relations available for each bit, m, depending on the parameters d (the degree of the characteristic polynomial) and N (the length of the observed keystream).

First we need an estimate of the total number, T, of relations available involving all the bits of the keystream. This depends on several parameters: let the degree of the feedback polynomial be d and the number of observed bits be N. The degree of the polynomial obtained by i squarings of the feedback polynomial is $2^i d$. The total number of recurrences obtained from this polynomial is $N - 2^i d$. These are obtained by "shifting" the recurrence up by one as many times as possible. Since this number must be positive i is at most $\lfloor \log_2(N/d) \rfloor$, which we'll abbreviate as $\log(N/d)$. The total number is thus

$$\begin{aligned} T &= \sum_{i=0}^{\log(N/d)} \left(N - 2^i d\right) \\ &= N\left(\log\left(N/d\right) + 1\right) - d \sum_{i=0}^{\log(N/d)} 2^i \\ &= N\left(\log\left(N/d\right) + 1\right) - d\left(2^{\log(N/d)-1}\right) \\ &= N\log\left(N/d\right) + N - d\left(N/d - 1\right) \\ &= N\log\left(N/d\right) - log\left(2\right) + d \\ &= N\log\left(N/2d\right) + d. \end{aligned}$$

Each of these T relations involves $t+1$ bits of the sequence, where t is the number of taps in the correlated LFSR. The average number of relations involving a particular bit in the sequence is thus

$$\begin{aligned} m &= \left(\frac{T(t+1)}{N}\right) \\ &= \left(\log\left(\frac{N}{2d}\right) + \frac{d}{N}\right)(t+1) \\ &\approx \log\left(\frac{N}{2d}\right)(t+1). \end{aligned} \tag{3.19}$$

The last approximation holds because we are concerned with cases where the observed keystream is much longer than the LFSR length and where t is small, so $(d/N)(t+1) \ll 1$.

The above method of generating parity check equations is used in both algorithms. Having covered all the relevant background, we are now able to discuss the Iterative Decoding Algorithm.

3.2.2 Iterative Decoding Algorithm

This algorithm exploits the fact that if z_i has a low updated probability, $p_i^\star$, of being correct (or equivalently satisfies few parity check equations), then it is likely that the complement $1 - z_i = a_i$. In order to reconstruct the LFSR sequence, this algorithms finds the bits with small $p_i^\star$, and complements them. By repeating this process, the sequence can be recovered. Under a suitable range of the parameters, the number of bits such that $z_i = a_i$ increases with each iteration. Meier and Staffelbach also introduce an effective variation to this method: instead of updating the probabilities for each bit once, the process is iterated several times for given sequence, before complementing the bits.

Correction Effect of Complementing Bits

In order to choose to choose an appropriate threshold probability some analysis is needed of the correction effect of complementing bits below a threshold. The analysis in this section applies at the start of the algorithm before any probabilities have been updated. We denote the event that z_i, satisfies at most h of m of its parity check equations as $H_i < h$. The probability of this event can be computed using equation (3.12) as

$$\text{Prob}(H_i < h) = \sum_{k=0}^{h} \binom{m}{k} (ps^k(1-s)^{m-k} + (1-p)(1-s)^k s^{m-k}) \tag{3.20}$$

Note that this probability depends only on the parameters m, h and p. Similarly the event that z_i satisfies at most h relations coinciding with $z = a$ and $z \neq a$ can be computed as

$$\begin{aligned} \text{Prob}(H_i < h \text{ and } z_i = a_i) &= \sum_{k=0}^{h} \binom{m}{k} ps^k(1-s)^{m-k} \\ \text{Prob}(H_i < h \text{ and } z_i \neq a_i) &= \sum_{k=0}^{h} \binom{m}{k} (1-p)(1-s)^k s^{m-k} \end{aligned} \tag{3.21}$$

These latter two equations also depend solely on the parameters m, h and p. For a given h chosen as the threshold value, the expected number of bits that satisfy at most h relations is $\text{Prob}(H < h) \cdot N$. If all these bits were complemented, then the the expected number of bits correctly changed is $\text{Prob}(H_i < h \text{ and } z_i \neq a_i) \cdot N$ and erroneously changed is $\text{Prob}(H^{<h} \text{ and } z = a) \cdot N$. The increase in the number of correct digits is

$$\text{Prob}(H_i < h \text{ and } z_i \neq a_i) \cdot N - \text{Prob}(H_i < h \text{ and } z_i = a_i) \cdot N. \tag{3.22}$$

The probability values depend only on m, h and p , so the relative increase can be computed as

$$\begin{aligned} I(p, m, h) &= \text{Prob}(H_i < h \text{ and } z_i = a_i) - \text{Prob}(H < h \text{ and } z_i \neq a_i) \\ &= \sum_{k=0}^{h} \binom{m}{k} (ps^k(1-s)^{m-i} - (1-p)(1-s)^k s^{m-k}) \end{aligned} \tag{3.23}$$

Setting the threshold to be $h = h_{\max}$ that maximises $I(p, m, h)$ will (on average) achieve the greatest correction.

Probability Threshold

For the modification to the method that does several iterations of updating the individual bits probability of being correct we need an equivalent threshold that can be compared the the update probabilities $p^\star$. Recall equation (3.13) that for a bit z_i and the updated probability $p_i^\star$ is given by

$$\begin{aligned} p_i^\star(p, m, h) &= \text{Prob}(z_i = a_i | H = h_i) \\ &= \frac{ps^h(1-s)^{m-h}}{ps^h(1-s)^{m-h} + (1-p)(1-s)^h s^{m-h}}. \end{aligned} \tag{3.24}$$

This allows to find the corresponding threshold on $p_i^\star$ which is chosen (again for maximum correction effect) to be[2]

$$p_{\text{thr}} = 1/2 \left(p^\star(p, m, h_{\max}) + p^\star(p, m, h_{\max} + 1) \right). \tag{3.25}$$

After the initial assignment of probabilities the expected number of bits with $p_i^\star < p_{\text{thr}}$ is

$$N_{\text{thr}} = \text{Prob}(H < h_{\max}) \cdot N. \tag{3.26}$$

The iteration of assignment of probabilities referred to above occurs when the observed number of digits with $p_i^\star \leq p_{\text{thr}}$ is less than the expected value. The probabilities are then updated again, only this time, the previous updated values are used as the prior correlation probability instead of the original values. As will be discussed in the results section, only a limited number of these iterations are necessary. (Meier and Staffelfbach found 5 to an appropriate number).

The algorithm is formulated in detail in Algorithm Listing 1. For later analysis the outermost loop is called a round, and the inner while loop that updates the probabilities is called an iteration. At the start of each round all the $\text{Prob}(z_i = a_i)$ are reset to the original correlation probability. Several iterations of updating the probabilities are done until either the expected number of bits below the threshold is reached, or the maximum number of iterations have occurred. Then each bit that has a probability of being correct below the threshold is complemented. If the resulting sequence is an LFSR sequence, the algorithm terminates, otherwise it goes through another round.

Algorithm 1 Iterative Decoding Algorithm for the Fast Correlation Attack

```
Determine m (3.19)
Find h = h_max that maximises I(p, m, h) (3.23)
while z is not a LFSR sequence do
  Reset all p_i ← p
  iter ← 0
  N_w ← 0
  while N_w < N_thr or iter < α do
    Update all probabilities p_i ← p_i^⋆
    iter ← iter + 1
  end while
  Complement all bits of z with p < p_thr
end while
return z
```

[2]The threshold value is chosen to be the average for continuity correction. The distribution of h is discrete and $p^\star$ is continuous, so it cannot be evaluated at only the single point corresponding to $h_{\max}$.

3.3 Implementation

The Iterative Decoding Algorithm was implemented in C. The need to use large efficient bit arrays motivated this choice. The program was tested on Ubuntu Linux i386 running on a laptop with an Intel Core Duo CPU 1.83GHz processor. While the code conforms to the C99 standard and is reasonably portable, it may not run correctly on systems on which the type `unsigned char` is not 8 bits. This is because the bit array implementation represents the array as an array of `unsigned char`. When modifying an individual bit, the given bit position modulo 8 is computed to see which `unsigned char` it is in the array. On systems where the size of `unsigned char` is not 8 bits, this may result in unexpected behaviour.

The code is built using GNU Autotools suite and requires GNU Libtool to build the shared libraries. Unit testing is provided by Check, which makes use of the built shared libraries. The unit tests cover almost all the code, including all the probability computations, and the parity check code. Credit needs to be given to Michael Dipperstein's BitArray implementation[3] which is used throughout the code.

Additionally the code depends on the following the libraries:

- GNU Scientific Library
- getopt (for parsing command line options)
- Check Library (for running unit tests)

The Geffe generator is implemented in a python script. This script generates test data for the after prompting for the initial states and taps of the three input LFSR. Usage details are contained in the README files in the `./tests/` directory of the source code. Joerg Arndt's lists of primitive polynomials[4] in $\mathbb{Z}_2[X]$ is used for generating the test data. The use of these primitive polynomials ensures that the corresponding inputs LFSR sequences to Geffe generator are maximal-length.

[3] `http://michael.dipperstein.com/bitlibs/`

[4] `http://www.jjj.de/mathdata/all-lowblock-primpoly.txt`
`http://www.jjj.de/pari/primpoly.gpi`

Chapter 4

Results

The keystreams on which this program is tested on are all the output of the Geffe keystream generator defined in Section 2.1 with different LFSR. Since the Geffe generator has a correlation probability $p = 0.75$ the testing does not consider the effect of varying correlation probability on the algorithm's success. The parameters that do vary are N, the length of the observed keystream, t the number of taps and L the LFSR length. This section first discusses the running of the algorithm on a typical successful case. Next it considered a certain case where the algorithm converges but to an incorrect LFSR sequence. Lastly, the range of parameters the algorithm is successful under is established, and suggestion is made for an improvement of it in some failed cases. The test data used in this section is available in the `./tests/data` directory of the source code archive.

4.1 A Typical Successful Run of the Algorithm

The successful running of an instance of the correlation problem is shown in Table 4.1. The symbol `x` indicates an incorrect bit, and `-` a correct one. The highlighted bits indicate that the current probability, $\text{Prob}(z_i = a_i)$, is below the threshold. At the beginning of each round, iteration 0 is the current modified sequence before any probabilities have been updated. For round 1, only one iteration of updating the probabilities was necessary to obtain sufficient bits with their probability below the threshold. So the bits z_0, z_2, z_3 and z_4 are flipped. Note that three of those bits are erroneously complemented, so there is a decrease in the number of correct digits. This point will be explained later when discussing the correction factor. In round 2 the first iteration of the probabilities yields no probabilities below the threshold. In general it is observed that after the first round, a few iterations are necessary to find bits below the threshold; the probabilities of each bit tend to converge with every iteration to either a high value close to 1, or a small value close to 0. In this case after 1 iteration during round 2 we have $\text{Prob}(z_0 = a_0) = 0.64$, $\text{Prob}(z_2 = a_2) = 0.75$ and $\text{Prob}(z_3 = a_3) = 0.62$. These are lower than the the probability of the correct bits, for example $\text{Prob}(z_6 = a_6) = 0.92$, but still above the threshold of 0.52. After the second iteration a sharper distinction between the correct digits and incorrect digits is now possible. Now $\text{Prob}(z_0 = a_0) = 0.49$, $\text{Prob}(z_2 = a_2) = 0.37$, $\text{Prob}(z_3 = a_3) = 0.50$, and the $\text{Prob}(z_6 = a_z) = 0.99$. This process didn't work for all the bits in this round. A given bit may still satisfy enough relations, involving enough other bits of high probability, and making it as though the bit is correct. An example is the last bit where $\text{Prob}(z_9 = a_9)$, is relatively low after the first iteration of the round, 0.64, and has a similar

Round	Iteration	z
1	0	- x - - x - - - - x
	1	- x - - x - - - - x
2	0	x x x x - - - - - x
	1	x x x x - - - - - x
	2	x x x x - - - - - x
3	0	- x - - - - - - - x
	1	- x - - - - - - - x
	2	- x - - - - - - - x
	0	- - - - - - - - - -

Table 4.1: A successful run of the algorithm on an instance with $N = 10$, $L = 2$, and correlated LFSR $a_n = a_{n-1} + a_{n-2}$.

probability after the second iteration.

In the last round, two iterations of the probability assignments are necessary to sharply polarise the bits believed to be correct or not. At the end, the Prob($z_0 = a_0$) and Prob($z_9 = a_9$) are both approximately 0.2 and most of the other bits in the sequence have high probabilities greater than 0.90 of being correct. The bits are complemented, and the sequence is tested to see if it satisfies the LFSR recurrence, which indeed it is in this case. This polarisation that occurs justifies ending the round after a finite, α, number of iterations since the probabilities of the bits would have by then converged to very high or low probabilities.

4.2 Converging on an Incorrect Phase of the LFSR Sequence

It can happen in certain cases where $d = {}^{N}/_{L}$ is small that the algorithm converges to a sequence that satisfies the LFSR recurrence but it is incorrect. In this case, since the input LFSR are maximal, the algorithm's sequence would be a shift of the correct sequence. This happens because both the algorithm's sequence and the correct sequence, have a similar number of digits differing from the keystream. The example in table 4.2 is the same as table 4.1 except with 4 less keystream bits.

Round	Iteration	z
1	0	- x - x - -
	1	- x - x - -
	2	- x - x - -
	0	x x - x x -

Table 4.2: Algorithm converging on the wrong sequence with $N = 6$ and $L = 2$.

In this case it happens that both the algorithm's and the correct LFSR sequence only differ in two positions from the keystream. Table 4.3 highlights the positions where the sequences

different. It is easy to see that both are valid LFSR sequences, since both satisfy the equation $a_n = a_{n-1} + a_{n-2}$.

	Sequence
Keystream	001111
Correlated LFSR	011011
Algorithm's LFSR	101101

Table 4.3: Comparison of keystream and the two valid LFSR sequences.

It is sufficient to add 4 keystream bits and this problem then becomes the one in table 4.1, and the algorithm finds the correct answer. Experiments with the Geffe generator have found that choosing $d \geq 5$ is sufficient to avoid this problem of two valid LFSR sequences being too close to the keystream. However, this is not always sufficient to ensure the success of the algorithm.

4.3 Parameters for a Successful Attack

This section seeks to establish the range of parameters under which the algorithm is successful. Table 4.4 summarises the running of twenty instances of the problem under various values of t and $d = {}^{N}/_{L}$. It appears that more taps require higher value of d which in this is case is achieved by increasing N.

	$d = {}^{N}/_{L}$				
t	10	10^2	10^3	10^4	10^5
2	2	2	2	2	2
4	-	6	4	3	3
6	-	-	-	63	59
8	-	-	-	-	-

Table 4.4: Number of rounds required by the algorithm to produce the correct sequence. Failure is indicated by -.

In order to analyse the effectiveness of the algorithm, we consider the expected number of digits corrected at each round. [1] During the initial phase of the algorithm, $h_{\max}$ is chosen that maximises the function $I(p, m, h)$, the relative increase in the number of correct digits. We can define $I_{\max}$ to be the maximised value of that function $I_{\max} = I(p, m, h_{\max})$. Recall that m only depends on the parameters t and $d = {}^{N}/_{L}$ (see equation 3.19), and $h_{\max}$ depends only on p and m. This means that $I_{\max}$ can be written as a function of $I_{\max} = I_m(p, t, d)$. The expected increase in the number of correct digits is thus

$$N_c = I_m(p, t, d) \cdot N.$$

[1] This estimate is most accurate in the early rounds where roughly the expected number of digits have their probabilities below the threshold. In later rounds, less bits are incorrect, so less have their probabilities below the threshold.

For the analysis it will be useful to write this as a product independently of N

$$N_c = F(p, t, d) \cdot L$$

where $F(p, t, d) = I_m(p, t, d) \cdot {}^N/_L = I_m(p, t, d) \cdot d$ is the correction factor. The correction factor value for the test data is laid out in table 4.5. The highlighted values indicate where the algorithm failed. The table makes it clear that those cases with correction factor values

	$d = {}^N/_L$				
t	10	10^2	10^3	10^4	10^5
2	0.456	12.424	169.079	1965.961	21418.698
4	0.001	0.553	14.735	254.003	3679.062
6	< 0.00	0.000	0.000	0.018	1.190
8	< 0.00	< 0.00	< 0.00	< 0.00	< 0.00

Table 4.5: Correction factor values for test data. Highlighted values indicate the unsuccessful attempts and < 0.00 indicates a small negative value.

that are 0 or less will definitely fail. Large values are very likely to succeed. The small positive values (< 0.5) only sometimes succeed as is the case when $t = 6$ and $d = 10^4$. In the case illustrated by 4.1, it is observed that in the first round the increase in the number of correct digits is -2. With small correction factor values, this is always a possibility that the sequence will stray further from the correct sequence, and eventually not converge. (The correction factor in that case is 0.11.) However it is fortunate that in that case the algorithm is able to correct those mistakes in later rounds. When increasing the value of N and hence d, the correction factor increases and the possibility of significantly decreasing the number of correct digits is much less.

The table illustrates that larger values of t require larger values for d to succeed. A natural question to ask is, if this implementation were to succeed on an LFSR of 8 taps how large would $N_{\text{req}} = N$ have to be to make d large enough? Since N is the number of keystream bits, another way of asking this, is to ask how much keystream would need to be observed. Meier and Staffelbach, in [6], calculate values of d and p required to make $F(p, t, d) = 0.5$. Since this test data is generated by the Geffe generator we require $p \approx 0.75$, which requires $d > 10^{10}$. In this case, L is 10, which means that $N_{\text{req}} > 10^{11}$. However the program cannot store bit arrays this long, because it uses arrays of 8 bit `unsigned char`, indexed by 32 bit `unsigned int`. The maximum N_m is thus 2^{35}. Taking logs (base 2) of both these values shows that

$$\log_2 N_m = 35 \leq 36 \leq 11 \log_2 10 \leq \log_2 N_{\text{req}}. \tag{4.1}$$

4.4 A Discussion of Failed Cases

A last case to discuss is the behaviour of the algorithm on certain data that doesn't fit the assumptions. The LFSR used as input to the keystream generator are assumed to maximal, as they are most likely are in practice. This can cause the algorithm to fail if an LFSR is used which isn't maximal. These fail in a similar way to previous cases. At the last round, the

algorithm iterates the maximum α number of times, and the probabilities of each bit being correct is very high. Since none are below the threshold, the algorithm cannot complement any bits. There is no point continuing at this stage, since exactly the same behaviour would happen on the next round. The algorithm then checks if this sequence satisfies the LFSR recurrence and fails if it doesn't. On some test data it has been observed that, after the first iteration of the last round, some of the incorrect bits have a high probability, but lower than others with very high probability of being correct. However, the probabilities of the incorrect values are still to high to be below the threshold. After the next iteration, the probability values are all indistinguishably very high. A possible improvement based on this observation is that, after recognising that it is in this state, the algorithm could backtrack to just after the first iteration of the last round. A higher threshold could be calculated and used instead. This may require analysis of the inter-dependence of the bits introduced from the changes made in earlier rounds.

In summary, for the test data generated by the Geffe generator, the algorithm is most successful for number of taps $t < 8$, and having N chosen so that $d = {}^{N}/_{L}$ is large enough. Large enough requires that $F(p, t, d) > 0.5$, however, for this implementation to be successful on the input, N should be smaller than the maximum length it can handle.

Chapter 5

Conclusion

Some stream ciphers use a combination generator, which consists of a boolean combining function taking inputs from several LFSR. The resulting keystream produced is intended to be pseudo-random. A poorly chosen combining function can leak information about its constituent LFSR. One way of leaking information is if the keystream and an input LFSR sequence is correlated. Correlation Attacks exploit this, and the linearity of LFSR sequences, to recover the LFSR sequence. Knowing the sequence implies knowing the initial state of the LFSR, and since combination generators are often initialised with the key for the stream cipher, partial knowledge of the key too.

This project provided an implementation of a Fast Correlation Attack and tested it under a range of inputs. The implementation allows the recovery of the correlated LFSR sequence for taps less than or equal to 6 and length up to 16 bits, provided that the number of keystream bits, N, is made suitably long for the correction factor to be greater than 0.5. Larger N is better, though the program is limited in the number of bits it can handle.

The work here can be extended in several ways. The algorithm for finding parity check equations could be improved using methods from [1, 3, 9], to allow for a sharper distinction between the correct and incorrect bits. Analysis could be done to explain the effectiveness of the method in later rounds, while considering the inter-dependency of bits introduced by the earlier rounds. The implementation could also be improved to handle longer sequences. Lengths of 10^{12} may allow for the successful recovery of a correlated LFSR sequence produced with 8 taps. The running time could be improved with a faster method evaluating parity check equations.

A further extension could be to consider this attack in the context of known-ciphertext. Redundancy in the plaintext would be used to deduce bits of the keystream. (For example the lower case letters in the ASCII encoding all have the same two bit prefix 01). The algorithm could be extended to operate on keystream sequences where there are semi-regular gaps of unknown bits.

Appendix A

Appendix

A.1 Modified Exhaustive Search Algorithm

The Modified Exhaustive Search Algorithm relies on the fact that the LFSR sequence being searched for can be constructed out of L bits of the sequences by solving linear equations for the initial state. (If the equations are linearly dependent, more bits will have to be found.) This algorithm chooses L bits that satisfy more than a threshold number of parity check equations, and uses these to find an initial guess for $\mathbf{a}$. This is tested to see if it is the correlated sequence, otherwise small modifications are made to the initial guess and the process is repeated.

Some analysis is required to chose the appropriate threshold value for the number of parity checks to be satisfied. The probability that a given bit z_i satisfies at least h relations, denoted as the event $H_i > h$ is

$$\text{Prob}(H_i > h) = \sum_{k=h}^{m} \binom{m}{k} (ps^k(1-s)^{m-k} + (1-p)(1-s)^k s^{m-k}) \tag{A.1}$$

This follows from equation (3.12) and depends only on p, h and m. The probability that both $z_i = a_i$ and z_i satisfies at least h parity check equations is given by

$$\text{Prob}(z_i = a_i \text{ and } H^{>h}) = \sum_{k=h}^{m} \binom{m}{k} ps^k(1-s)^{m-k}, \tag{A.2}$$

which likewise depends only on the same three variables. This allows the conditional probability that $z = a$ given that at least h parity check equations are satisfied to be expressed as

$$\text{Prob}(z_i = a_i | H_i > h) = \frac{\text{Prob}(z_i = a_i \text{ and } H_i > h)}{\text{Prob}(H_i > h)}. \tag{A.3}$$

The expected number of bits satisfying at least h relations is $\text{Prob}(H_i > h) \cdot N$. The threshold, $h = h_{\max}$ needs to be chosen so that $\text{Prob}(H_i > h) \cdot N \geq L$. The bits that satisfy more than $h_{\max}$ relations will give the initial guess I_0 for the LFSR sequence $\mathbf{a}$. Of these bits the expected number of incorrect bits is

$$(1 - \text{Prob}(z_i = a_i | H_i > h)) \cdot \text{Prob}(H_i > h) \cdot N. \tag{A.4}$$

If this number is small, then the correct sequence can be found by trying small modifications to I_0. After each modification, the corresponding LFSR sequence can be found using linear algebra to find the initial state, and then tested to see if it is sufficiently correlated to the keystream sequence (see [11]).

The algorithm is stated Algorithm Listing 2. The Hamming distance between two binary strings is the number of corresponding positions at which they differ.

Algorithm 2 Modified Exhaustive Search for the Fast Correlation Attack

```
Determine m (3.19)
Find maximum h = h_max such that Prob(H^{>h}) > l (A.1)
Set I_0 as the sequence bits of z that satisfy at least h_max relations
for i = 0, 1, 2 ... → l do
  for all I' at Hamming distance i away from I_0 do
    Get LFSR sequence, a', from I'
    if a' is sufficiently correlated with z then
      return a'
    end if
  end for
end for
```

A variation of this algorithm is, instead of selecting digits that satisfy a certain threshold of parity checks, to compute the updated probability, $p_i^\star$ for each bit. Then select L bits with the highest probability of being correct to use as the initial guess I_0.

A.2 The Freshman Theorem

For a polynomial $C(X) = c_0 + c_1X + c_2X^2 + \ldots + c_nX^n$ in $\mathbb{Z}_p[X]$ (i.e. with integer coefficients taken modulo p) where p is a prime number,[1]

$$C(X)^p = c_0^p + (c_1X)^p + (c_2X^2)^p + \ldots + (c_nX^n)^p. \tag{A.5}$$

Proof. This is proved by induction on the degree of $C(X)$. Where the degree of the polynomial is 0 (i.e. $C(X)$ is constant) this is easily seen to be true.

Now for $C(X)$ where its degree is $n > 0$:

$$\begin{aligned} C(X)^p &= (c_0 + c_1X^1 + \ldots + c_nX^n)^p \\ &= (c_0 + X(c_1 + \ldots + c_nX^{n-1}))^p \end{aligned}$$

Using the binomial theorem this becomes

$$\begin{aligned} C(X)^p &= \sum_{i=0}^{p} \binom{p}{i} (c_0)^{p-i} X^i (c_1 + \ldots + c_nX^{n-1})^i \\ &= c_0^p + \sum_{i=1}^{p} \binom{p}{i} (c_0)^{p-i} X^i (c_1 + \ldots + c_nX^{n-1})^i \end{aligned}$$

The inductive hypothesis allows to rewrite this as

$$\begin{aligned} C(X)^p &= c_0^p + \sum_{i=1}^{p} \binom{p}{i} (c_0)^{p-i} X^i (c_1^i + \ldots + (c_nX^{n-1})^i). \\ &= c_0^p + \sum_{i=1}^{p} \binom{p}{i} (c_0)^{p-i} ((c_1X)^i + \ldots + (c_nX^n)^i). \end{aligned}$$

Now when $0 < i < p$,

$$\binom{p}{i} = \frac{p!}{i!(p-i)!}$$

contains no multiple of p in the denominator, since both i and $p - i$ are strictly less than p and p is prime. Thus p divides $\binom{p}{i}$ for $0 < i < p$, allowing us to rewrite the above as

$$\begin{aligned} C(X)^p &= c_0^p + \binom{p}{p} ((c_1X)^p + \ldots + (c_nX^n)^p) \\ &= c_0^p + (c_1X)^p + (c_2X^2)^p + \ldots + (c_nX^n)^p \end{aligned}$$

□

[1] This is called the freshman theorem after a common mistake beginner students make thinking that $(x+y)^p$ is equal to $(x^p + y^p)$ when working with integer coefficients. However when we are working in characteristic p (or equivalently modulo p) it turns out to be true.

Bibliography

[1] A. Canteaut and M. Trabbia. *Improved Fast Correlation Attacks Using Parity-Check Equations of Weight 4 and 5*, volume 1807 of *Lecture Notes in Computer Science*, pages 573–588. Springer Berlin / Heidelberg, 2000.

[2] V. Chepyzhov and B. Smeets. *On A Fast Correlation Attack on Certain Stream Ciphers*, volume 547 of *Lecture Notes in Computer Science*, pages 176–185. Springer Berlin / Heidelberg, 1991.

[3] F. Jonsson. *Some Results on Fast Correlation Attacks.* Ph.d. thesis, Lund University, 2002.

[4] Y. Lu and S. Vaudenay. *Faster Correlation Attack on Bluetooth Keystream Generator E0*, volume 3152 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin / Heidelberg, 2004.

[5] W. Meier and O. Staffelbach. Fast Correlation Attacks on Stream Ciphers. *Advances in Cryptology EUROCRYPT 88 Lecture Notes in Computer Science*, 330:301–314, 1988.

[6] W. Meier and O. Staffelbach. Fast Correlation Attacks on Certain Stream Ciphers. *J. Cryptology*, 1(3):159–176, 1989.

[7] A. J. Menezes, P. C. V. Oorschot, S. A. Vanstone, and R. L. Rivest. Handbook of applied cryptography, 1997.

[8] M. Mihaljevic and J. Golic. *A fast iterative algorithm for a shift register initial state reconstruction given the noisy output sequence*, volume 453 of *Lecture Notes in Computer Science*, pages 165–175. Springer Berlin / Heidelberg, 1990.

[9] M. Mihaljević and J. Golić. *A Comparison of Cryptanalytic Principles Based on Iterative Error-Correction*, volume 547 of *Lecture Notes in Computer Science*, pages 527–531. Springer Berlin / Heidelberg, 1991.

[10] R. M. Roth. *Introduction to Coding Theory.* Cambridge University Press, New York, 2006.

[11] T. Siegenthaler. Decrypting a Class of Stream Ciphers Using Ciphertext Only. *Computers, IEEE Transactions on*, C-34(1):81–85, 1985.