Simulation and Evaluation of Iterative Methods in Correlations Attacks on Stream Ciphers

Daniel Olofsson

Department of Electrical and Information Technology
Lund University

Advisor: Thomas Johansson

June 11, 2013

Printed in Sweden
E-huset, Lund, 2013

Abstract

Breaking a stream cipher with a brute-force attack can take a very long time and in some cases be practically impossible. A better alternative could be fast correlation attacks. Two well-known fast correlation attacks are Meier and Staffelbach's Algorithm A and B, where the last is the most interesting because it is iterative. It has led to reformulations which are based on log likelihood ratios instead of probabilities which the original algorithm is based on.

In this thesis, two formulations of Algorithm B based on log likelihood ratios are evaluated and compared. By using a linear feedback shift register and a binary symmetric channel as the keystream generator, output sequences with different lengths will be generated for feedback polynomials of the degree 20 and 30. The generator matrix for the linear feedback shift register will then be used to generate a large number of parity-check relations of weight 4, for the positions in the output sequences. These relations will be used by the two algorithms, which will try to find the initial state of the linear feedback shift register, which is the key of the stream cipher.

The results of the simulations in this thesis show that one of the log likelihood ratio based algorithms has a slightly better success rate of finding the initial state of the linear feedback shift register, than the other. But this increase in success rate is very small, which suggests that using a larger number of relations of weight 4 makes the difference between the two log likelihood ratio based algorithms less noticeable.

Table of Contents

Chapter 1
Goals and Overview

1.1 Goals

The goal of this thesis is to implement iterative methods of performing fast correlation attacks. After the implementation these methods will be evaluated for different crossover probabilities and lengths of the output sequence.

1.2 Overview

This thesis will start of with an introduction to symmetric ciphers, by explaining what a symmetric cipher is and by giving a few examples of historical ciphers. Then it will continue on to explain stream ciphers, which is the ciphers of interest to this thesis. It will also explain what linear feedback shift registers are and how they are used to implement a stream cipher. The introduction part of the thesis will then be concluded by an explanation of what a correlation attack is and how it can be used to attack a stream cipher.

The main part of the thesis will then explain what a fast correlation attack is and how they are performed. First an explanation about the prerequisites to perform a fast correlation attack is covered, which includes a method of generating parity-check relations. Then two algorithms by Meier and Staffelbach, will be presented and explained. Next an explanation of low-density parity-check codes will be given and how to decode such codes with a message passing algorithm. This will lead to the explanation of two algorithms, based on log likelihood ratios. The main part of the thesis is then concluded with an explanation about how the generation of the parity-check relations and the algorithms have been implemented.

The thesis will be concluded by presenting the simulations results, received by running the implementation. Last the conclusion, drawn from the work and the result of this thesis will be presented.

Chapter 2
Introduction

2.1 Symmetric Ciphers

Encryption is a way to transform a plaintext to a ciphertext with the help of a secret key. When encryption of the plaintext and decryption of the ciphertext requires the same key, the cipher is a symmetric cipher, described as

$$c = e_k(m),$$
$$m = d_k(c),$$

where m is the plaintext, e is the encryption function, d is the decryption function, k is the secret key and c is the ciphertext. Historically ciphers like shift cipher, substitution cipher, Vigenère cipher and permutation cipher have been used.

A shift cipher works in the following way: First all letters in the alphabet is given a number, then a key number is decided. When encryption of a plaintext message is performed the key number is added to each letters number and the new number modulo the size of the alphabet used will be the new letter in the ciphertext message. When decrypting a ciphertext message the key number is simply subtracted from the letter's number and the new number modulo the size of the alphabet used will give the letter in the plaintext message. This limits the total number of keys to the size of the chosen alphabet.

Example 1. *Take the English alphabet A-Z and assign the numbers 0-25 to the letter in each corresponding position, now using the key number 3 would give the following result*

Plaintext: *ABCDEFGHIJKLMNOPQRSTUVWXYZ*
Ciphertext: *DEFGHIJKLMNOPQRSTUVWXYZABC*

A substitution cipher substitutes every letter in the alphabet with a different letter in the alphabet, which has not been used before. This results in a larger number of possible keys than for a shift cipher. The substitution ciphers key can be any permutation of the chosen alphabet.

Example 2. *Given the English alphabet A-Z, the total number of possible permutations would be*

$$26! \approx 2^{88}$$

and a possible permutation can for example look like

Plaintext: *ABCDEFGHIJKLMNOPQRSTUVWXYZ*
Ciphertext: *FGSPDKLQMHNOFTUIVWRXYZAJBC*

If the word ACCEPT should be encrypted with this permutation, it would result in the word FSSDIX.

A Vigenère cipher works the same way as a substitution cipher but instead of using one permutation, several permutations are used. This will increase the cipher's number of keys exponentially, with the number of permutations used. Now each letter in the plaintext message will be substituted with a different letter, depending on how many permutations are used and the place of the letter in the plaintext message. If two permutations would be used the first letter in the plaintext message would be substituted with a letter in the first permutation, the second letter in the plaintext message would be substituted with a letter from the second permutation, the third letter of the plaintext message would be substituted with a letter from the first permutation and so on until the whole plaintext message is turned into a ciphertext message.

Example 3. *Given the English alphabet A-Z, using a total of 5 permutations would result in that the total number of possible permutations would be*

$$(26!)^5 \approx 2^{441}$$

and a possible sequence of permutations can, for example, look like

Plaintext: *ABCDEFGHIJKLMNOPQRSTUVWXYZ*
Ciphertext 1: *FGSPDKLQMHNOFTUIVWRXYZAJBC*
Ciphertext 2: *FDGSPMHKNOFTUILVWRXYZAQJBC*
Ciphertext 3: *PMHFKNOFTDUILVWRGXYZAQSJBC*
Ciphertext 4: *SPMHFJKNOFTDBUILVCWRGXYZAQ*
Ciphertext 5: *SXPMHFJKNOYFTDBUILVCWRZGAQ*

If the word ACCEPT should be encrypted with these permutations, it would result in the word FGHFUX.

A permutation cipher works by first deciding a group and then a permutation of this group will be the key. When encrypting a plaintext message, the message is first divided into chunks with the same size as the chosen group and if necessary a number of random letters are padded to the end of the message to make an even multiple to the size of the group. The key permutation is then applied to each chunk of the the plaintext message to scramble the letters and last the spaces are removed to hide the size of the group.

Example 4. *Choose the group S_5 and the permutation*

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 3 & 5 \end{pmatrix} = (1243) \in S_5.$$

Now take a plaintext message

accept this as a plaintext message.

Break it up into chunks of 5 letters

accep tthis asapl ainte xtmes sage.

Add random letters to make the number of total letters a multiple of 5

accep tthis asapl ainte xtmes sageq.

Apply the permutation to the message

caecp htits aapsl natie mxets gseaq.

Now remove the spaces to obtain the ciphertext message

caecphtitsaapslnatiemxetsgseaq.

2.2 Stream Ciphers

In Section 2.1 four historical ciphers have been introduced. Taking a closer look at the key input to the shift cipher, substitution cipher and the Vigenère cipher, one will see that the key is a more or less random continuous stream, also refereed to as a keystream. This keystream is continuously added to a plaintext stream, resulting in a ciphertext stream. This kind of cipher is called a stream cipher and in Figure 2.1 a model of a stream cipher is shown.

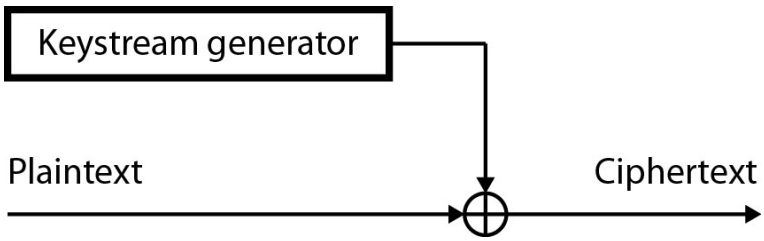

Figure 2.1: A model of a stream cipher.

A stream cipher usually performs encryption on bit level, according to

$$c_i = m_i \oplus z_i, \tag{2.1}$$

where c_i is the ciphertext bits, m_i is the plaintext bits and z_i is the keystream bits for $i = 0, 1, ...$ and decryption is performed by the same operation

$$m_i = c_i \oplus z_i. \tag{2.2}$$

In Example 5 an encryption and a decryption by a stream cipher is shown.

Example 5. *The plaintext bitstream sent into a stream cipher is*

$$\boldsymbol{m} = 101001101110110$$

and the corresponding bits in the output from the keystream generator is

$$\boldsymbol{z} = 111000010101111.$$

By using exclusive-or (XOR) with the plaintext bitstream and the keystream bits according to Formula 2.1 the ciphertext bits will be obtained

$$\begin{array}{r} 101001101110110 \\ \oplus 111000010101111 \\ \hline 010001111011001 \end{array}$$

hence $\boldsymbol{c} = 010001111011001$. *Using* $\boldsymbol{c}$ *and* $\boldsymbol{z}$ *in Formula 2.2, the plaintext bits will be obtained*

$$\begin{array}{r} 010001111011001 \\ \oplus 111000010101111 \\ \hline 101001101110110 \end{array}.$$

The security of a stream cipher depends on two things, the secrecy of the key to the keystream and of that this key is changed often. If the same keystream $\mathbf{z}$ is used more than once an adversary can get information about the XOR between plaintexts by adding the ciphertexts together,

$$c_1 \oplus c_2 = (m_1 \oplus \mathbf{z}) \oplus (m_2 \oplus \mathbf{z}) = m_1 \oplus m_2.$$

The keystream generator in Figure 2.1 is usually implemented by several linear feedback shift registers (LFSR) and a boolean function. LFSRs are often used in a keystream generator, because of their good statistical properties and because they usually have a large period before they start to repeat their output. When LFSRs are used to implement the keystream generator, the secret key is the initial state of the LFSRs. Figure 2.2 shows an example of a keystream generator.

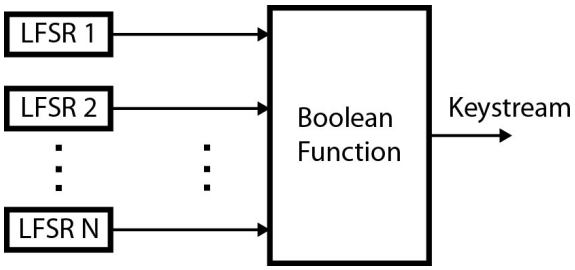

Figure 2.2: Example of a keystream generator with several LFSRs.

For an attacker, the keystream generator can also be modeled by one LFSR and a binary symmetric channel (BSC). The input bit to the BSC is the output bit from the LFSR. This bit will be correlated to the BSC's output bit by a probability p and by the probability $1 - p$ the input bit will be complemented. The probability p is determined from the properties of the boolean function, which the BSC is modeled after. In Figure 2.3 a keystream generator with a BSC is shown.

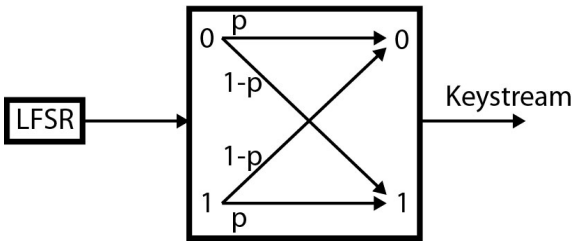

Figure 2.3: Example of a keystream generator with a BSC.

2.3 Linear Feedback Shift Registers

A LFSR with the length l is represented by l delay elements, a feedback polynomial and a clock signal. Each time unit the LFSR changes state by shifting all the current bits in the LFSR one step. When this shift occurs the most significant bit is sent as the output from the LFSR. The output bit from the LFSR then gets sent into the feedback polynomial, which is a linear feedback polynomial. The feedback polynomial then adds this bit with the other bits in the current state of the LFSR, according to the polynomial and this results in the next input bit to the LFSR. This bit will be the least significant bit of the new state of the LFSR. In Figure 2.4 a LFSR is shown.

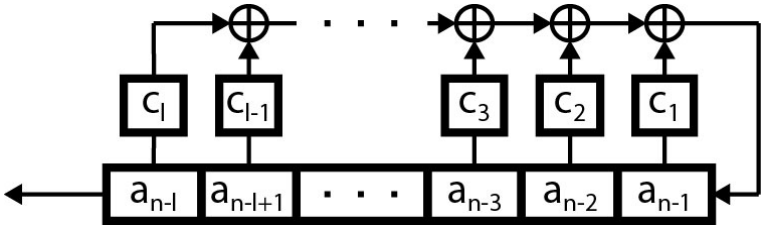

Figure 2.4: A model of a LFSR.

The next output bit from the LFSR can be calculated by the formula

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + ... + c_l a_{n-l}. \tag{2.3}$$

Formula 2.3 can also be written as

$$a_n = \sum_{i=1}^{l} c_i a_{n-i},\ n \geq l+1. \tag{2.4}$$

All the nonzero coefficients $c_1, c_2, c_3, ...$ from Formula 2.3, can be seen as taps in the feedback polynomial for the LFSR and by setting $c_0 = 1$ in the polynomial, the obtained linear feedback polynomial for the LFSR will look like

$$c(X) = 1 + c_1 X + c_2 X^2 + ... + c_l X^l, \tag{2.5}$$

which also is referred to as the connection polynomial of the LFSR.

Example 6. *Consider a LFSR of the length* 10*, the feedback polynomial* $c(x) = 1 + x^2 + x^{10}$ *and the initial state* $\boldsymbol{s} = 1101010101$*. Now by using Formula 2.3 and shifting* 10 *times the outputs and states in Table 2.1 are received.*

i	a_i	new state	output
0	-	1101010101	-
1	$0+1$	1010101011	1
2	$1+1$	0101010110	1
3	$1+0$	1010101101	0
4	$0+1$	0101011011	1
5	$1+0$	1010110111	0
6	$1+1$	0101101110	1
7	$1+0$	1011011101	0
8	$0+1$	0110111011	1
9	$1+0$	1101110111	0
10	$1+1$	1011101110	1

Table 2.1: Outputs and states for 10 shifts of the LFSR.

2.4 Correlation Attacks

A known plaintext attack is an attack where an adversary has knowledge about both the plaintext and the ciphertext generated by a cipher. The idea of the attack is to find a correlation between the plaintext and the ciphertext and then obtain the key which is used by the cipher. A correlation attack is one type of a known plaintext attack. When an adversary is attacking a stream cipher with a correlation attack, the state of the LFSRs are the keys which the adversary is trying to obtain. How a correlation attack works is better explained by Example 7, which is inspired by the explanation of a correlation attack in Nigel Smart's book [1].

Example 7. *Consider the Geffe keystream generator, which consists of three LFSRs, each with length* L_i *for* $i = 1, 2, 3$ *and the boolean function*

$$\boldsymbol{z} = f(x_1, x_2, x_3) = x_1 \cdot x_2 \oplus x_2 \cdot x_3 \oplus x_3, \tag{2.6}$$

where x_1, x_2 *and* x_3 *are the outputs from the LFSRs. The output from the boolean function is shown in Table 2.2. A closer look at Table 2.2 reveals that 6 of 8 outputs are matching between* x_1 *and* $\boldsymbol{z}$ *and between* x_3 *and* $\boldsymbol{z}$*. Then both* x_1 *and* x_3 *is correlated by the probabilities*

$$Pr(\boldsymbol{z} = x_1) = 0.75 \text{ and } Pr(\boldsymbol{z} = x_3) = 0.75.$$

Now by generating $2L_i$ *bits of output for the LFSR for each state of each primitive polynomial of the degree* L_i*, the correct generator polynomial and start state can be*

x_1	x_2	x_3	**z**
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: Inputs to and outputs from **z**

found. This is found when the output of $\boldsymbol{z}$ *and the generated bit sequence matches in at least* $Pr(\boldsymbol{z} = x_i)$ *of the positions in the sequences.*

For the correlation attack to work, the probability have to satisfy

$$Pr(\boldsymbol{z} = x_i) > 0.5.$$

Hence, the correlation attack will only work for x_1 *and* x_3*, in this example. The reason that the correlation attack can not be used on* x_2*, is because*

$$Pr(\boldsymbol{z} = x_2) = 1/2.$$

Because of this the initial state of the second LFSR needs to be found by brute-force, which will try to find a state that satisfies Formula 2.6.

Chapter 3
Fast Correlation Attacks

3.1 Generating Parity-Checks with Low Weight Polynomials

To be able to perform a fast correlation attack against a stream cipher, relations between the positions in the keystream sequence **z** have to be obtained. These relations are linear and correlates to the corresponding position in the LFSRs output sequence **a**. Meier and Staffelbach describe in [2] how to obtain these linear relations, by first taking a look at the sequence **a** and then transforming them to the sequence **z**.

The first method of obtaining relations for each position a_n is by shifting Formula 2.3 t times, as shown in Example 8. By only using the shifting $t+1$ relations for position a_n are obtained. To find more relations for position a_n, the multiples of the feedback polynomial from Formula 2.5 can be used. To obtain the multiples, the feedback polynomial is squared multiple times, according to $c(X)^j = c(X^j)$ for $j = 2^i$ and this can also be seen in Example 8.

Example 8. *Consider a LFSR of the length* $l = 10$ *and the feedback polynomial* $c(x) = 1 + x^3 + x^{10}$*, then the linear equation*

$$a_j = a_{j-3} + a_{j-10}, j \geq 10$$

is a relation for the position a_n*. Another linear relation can be obtained by squaring, as* $c(x)^2 = c(x^2) = 1 + x^6 + x^{20}$ *and this would give the linear equation*

$$a_j = a_{j-6} + a_{j-20}, j \geq 20.$$

Then three relations could be found by shifting a_j *as*

$$\begin{aligned} a_{j-10} + a_{j-3} + a_j &= 0 \\ a_{j-7} + a_j + a_{j+3} &= 0 \\ a_j + a_{j+7} + a_{j+10} &= 0 \end{aligned}$$

Each obtained relation for a position, which differs from all other relations containing the specified position, can be added to a list L of relations for the positions

it contains. In this list, each relation, L_i, is a parity-check for the involved positions and for each position the total number of parity-checks will sum up to an individual value of m relations. These relations may be written as

$$\begin{aligned} L_1 &= a_n + b_1 = 0, \\ L_2 &= a_n + b_2 = 0, \\ &\vdots \\ L_m &= a_n + b_m = 0, \end{aligned}$$

where a_n is a fixed position and b_i, $i = 1, 2, \ldots, m$, is the sum of the t terms of each linear relation generated by the methods above. Now by substituting the sequence $\mathbf{a}$ with the keystream sequence $\mathbf{z}$, each relation would now look like

$$L_i = z_n + y_i, \ i = 1, 2, \ldots, m,$$

where z_n corresponds to the position a_n and the positions in y_i correspond to the positions in b_i. The values of the positions will, however, not necessarily be the same for the involved positions. Because of this, is it necessary to know the total number of satisfied relations, h, for each position, to be able to separate the correct and incorrect positions from each other.

3.2 Meier and Staffelbach Algorithm A

Algorithm A is not an iterative algorithm, instead it is a one-pass algorithm. To make a correlation attack according to algorithm A, one must have knowledge about the keystream sequence $\mathbf{z}$ and its length N. It is also required to know the structure of the LFSR, which means the length l and the used feedback polynomial. Then the probability $p = \Pr(z = a) > 0.5$ is required, this probability is the correlation between the sequence $\mathbf{a}$ and the sequence $\mathbf{z}$.

To find the initial state of the LFSR, the probability p^* has to be calculated for each position of the sequence $\mathbf{z}$. If the probability p^* is large for a position, the more probable is it that the position is correct. Each p^* is calculated by the formula

$$\begin{aligned} p^* &= P(z = a | L_1 = \cdots = L_h = 0, L_{h+1} = \cdots = L_m = 1) \\ &= \frac{ps^h(1-s)^{m-h}}{ps^h(1-s)^{m-h} + (1-p)(1-s)^h s^{m-h}}, \end{aligned} \tag{3.1}$$

where $p = \Pr(z = a)$, m is the total number of relations for the position, h is the number of satisfied relations and s is calculated by the recursion

$$\begin{aligned} s(p,t) &= ps(p,t-1) + (1-p)(1-s(p,t-1)), \\ s(p,1) &= p. \end{aligned} \tag{3.2}$$

When all the new probabilities are calculated according to Formula 3.1, the l positions which have the highest p^* are chosen to be used in a system of linear

equations. By solving this system the initial state of the LFSR can be found. If there are equations in the system which depends on each other, more positions can be added to the system to make sure that there are l independent equations.

Algorithm A:

1. Calculate p^* for each position of the sequence $\mathbf{z}$ according to Formula 3.1.
2. The l positions with the highest p^* are chosen and put in a system of linear equations.
3. If possible, the system of linear equations is solved and then the initial state of the LFSR is found.
4. If the system of linear equations can not be solved, a modification of the l positions with a Hamming distance of $1, 2, \ldots$ are made and then go back to 3.

3.3 Meier and Staffelbach Algorithm B

Algorithm B is an iterative algorithm which tries to correct all positions of the sequence $\mathbf{z}$, to make it equal to the sequence $\mathbf{a}$. When $z_i = a_i$ for $i = 1, \ldots, N$, the initial state of the LFSR can be obtained from the first l positions of the corrected sequence. The algorithm iterates over all positions calculating the probability p^* according to Formula 3.1, with p set to each position's individual probability and s calculated by a modified version of the recursion in Formula 3.2. The modified recursion takes each involved position's individual probability into account and is defined as

$$\begin{aligned} s(p_1, \ldots, p_t, t) &= p_t s(p_1, \ldots, p_{t-1}, t-1) + (1-p_t)(1 - s(p_1, \ldots, p_{t-1}, t-1)), \\ s(p_1, 1) &= p_1. \end{aligned} \tag{3.3}$$

Each iteration will make the probabilities converge towards $p^* = 0$ or $p^* = 1$, depending on whether the position is incorrect or correct. The algorithm will iterate until a certain threshold is reached and then the positions in sequence $\mathbf{z}$, which have passed the threshold, will be complemented. These iterations will go on until $\mathbf{z} = \mathbf{a}$. This approach of Algorithm B will make hard decisions to decide the correctness of each position. In order for a soft decision to be made, the probability for all relations has to be taken into account for each position.

In order for all relations to contribute to the probability p^* for each position, all relations containing the position will be put in a set $\mathbb{M}$ and the relations which

are satisfied will also be put in a set $\mathbb{H}$. Formula 3.1 will be modified as

$$\begin{aligned} p^* &= \Pr[z_n = a_n | \text{ relation } j \text{ holds for } j \in \mathbb{H} \text{ but not for } j \in \mathbb{M} \setminus \mathbb{H}] \\ &= \frac{p \prod_{j\in\mathbb{H}}(s_j) \prod_{j\in\mathbb{M}\setminus\mathbb{H}}(1-s_j)}{p \prod_{j\in\mathbb{H}}(s_j) \prod_{j\in\mathbb{M}\setminus\mathbb{H}}(1-s_j) + (1-p) \prod_{j\in\mathbb{H}}(1-s_j) \prod_{j\in\mathbb{M}\setminus\mathbb{H}}(s_j)}. \end{aligned} \tag{3.4}$$

where s_j is calculated with Formula 3.3 and the positions involved in relation j.

Before the thresholds can be calculated, the average number of total relations m have to be estimated. If the relations are generated by the methods described in Section 3.1, then m can be approximated by the formula

$$m \approx \log\left(\frac{N}{2l}\right)(t+1), \tag{3.5}$$

where N is the length of the generated sequence $\mathbf{z}$, l is the length of the LFSR and t the number of feedback taps of the used feedback polynomial. The next thing which has to be determined before the thresholds can be calculated, is the maximum number of satisfied relations, h_{max}, which will give the maximum correction for each round of the algorithm. The maximum correction is obtained by finding a h such that $I(p, m, h)$ is maximized for the given p and approximated m. $I(p, m, h)$ is the relative increase of correct positions and calculated as

$$I(p, m, h) = W(p, m, h) - V(p, m, h), \tag{3.6}$$

where $W(p, m, h)$ is the probability that $z \neq a$ and that not more than h of m relations are satisfied, and calculated by the formula

$$W(p, m, h) = \sum_{i=0}^{h} \left(\begin{array}{c} m \\ i \end{array} \right) (1-p)(1-s)^i s^{m-i}. \tag{3.7}$$

Also, $V(p, m, h)$ is the probability that $z = a$ and that not more than h of m relations are satisfied, and calculated by the formula

$$V(p, m, h) = \sum_{i=0}^{h} \left(\begin{array}{c} m \\ i \end{array} \right) p s^i (1-s)^{m-i}. \tag{3.8}$$

The first threshold can now be calculated by the formula

$$p_{thr} = \frac{1}{2}(p^*(p, m, h_{max}) + p^*(p, m, h_{max} + 1)), \tag{3.9}$$

where $p^*(p, m, h_{max})$ is calculated with Formula 3.1. All positions of the sequence $\mathbf{z}$ which has $p^* < p_{thr}$ after the last iteration of each round of the algorithm will be complemented. Each round of the algorithm continues to iterate until either the number of iterations equals α (according to [2], $\alpha = 5$ is the most common choice) or the number of positions which will be complemented N_w is greater than the threshold N_{thr}. To decide this threshold it is necessary to know the probability

that not more than h of m relations are satisfied. This probability is calculated by the formula

$$U(p,m,h) = \sum_{i=0}^{h} \left(\begin{array}{c} m \\ i \end{array} \right) (ps^i(1-s)^{m-i} + (1-p)(1-s)^i s^{m-i}). \tag{3.10}$$

By using $h = h_{max}$ in Formula 3.10, the threshold can now be calculated by the formula

$$N_{thr} = U(p,m,h_{max}) \cdot N. \tag{3.11}$$

Algorithm B:

1. Calculate m with Formula 3.5.
2. Find $h = h_{max}$ such that $I(p,m,h)$ is maximized. Calculate p_{thr} and N_{thr} with Formula 3.9 and 3.11.
3. Set the iteration counter to $i = 0$.
4. For every position of the sequence $\mathbf{z}$ calculate the probabilities p^*, according to Formula 3.1 (or 3.4) and 3.3, taking each positions individual h and m into account. Count the number N_w of positions with $p^* < p_{thr}$.
5. If $N_w < N_{thr}$ or $i < \alpha$, increment i and go to 4.
6. Complement the positions of the sequence $\mathbf{z}$ with $p^* < p_{thr}$ and reset the probabilities of each position to the original probability p.
7. If there are positions of the sequence $\mathbf{z}$ which not satisfies Formula 2.3 go to 3.
8. Terminate with $\mathbf{z} = \mathbf{a}$.

3.4 Generating Low-Density Parity-Checks

In Section 3.1 a way of generating parity-checks by squaring and shifting has been presented, which finds the relations between the positions in the sequence $\mathbf{z}$. But if the feedback polynomial for the LFSR is of high-density, the parity-checks generated by those methods will also be of high-density. This property is not a property one desires to have for the generated parity-checks. The desired type of parity-checks is those with a low weight. Because of this desired property, low-density parity-check (LDPC) codes are suitable to use in the generation of the relations between the positions in the sequence $\mathbf{z}$. LDPC codes were introduced by Gallager in [3].

The type of LDPC codes used in the generation of the relations is the irregular type. This type of LDPC codes do not have the same weight in all columns or in all rows in the parity-check matrix $\mathbf{H}$. In order for the LDPC codes to be of low-density in a $n \times m$ matrix, the number of 1's in each column and row have to fulfill $w_c \ll n$ and $w_r \ll m$. In Example 9 it is shown how the generator matrix $\mathbf{G}$ can be calculated with knowledge about the parity-check matrix $\mathbf{H}$.

Example 9. *Given a* 3×7 *parity-check matrix*

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

The generator matrix $\boldsymbol{G}$ *can be obtained by first doing normal row operations to put the matrix* $\boldsymbol{H}$ *on the form* $[-\boldsymbol{P}^T|\boldsymbol{I}]$ *as*

$$\begin{aligned} \boldsymbol{H} &= \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \\ &\sim \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}. \end{aligned}$$

Now by shifting the sub matrix's to the form $[\boldsymbol{I}|\boldsymbol{P}]$*, the generator matrix* $\boldsymbol{G}$ *can be obtained*

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Once a generator matrix is obtained, relations between the positions can be found by XOR together a number of columns, which sums up to zero, equal to the desired weight of the parity-check relation.

Decoding of LDPC codes is based on belief propagation and performed by message passing between two type of nodes, called message nodes and check nodes. The messages sent between these nodes are probabilities, these probabilities can also

be seen as beliefs. Each check node has a belief about the correct value for the message nodes it is connected to. For an observed channel a, each check node will connect a number of observed channel symbols, a_i, which sums up to zero. The channel symbols, a_i, corresponds to the columns of the generator matrix for the channel a. Because of this, the check nodes will correspond to the parity-check relations for the generator matrix. Each check node will be on the form

$$a_n = a_{i_1} + \cdots + a_{i_{w-1}}, \tag{3.12}$$

where a_n is the current message node, each a_i a different message node and w the weight of the parity-check relation in the check node. Figure 3.1 show message nodes and a few check nodes obtained from the generator matrix in Example 9.

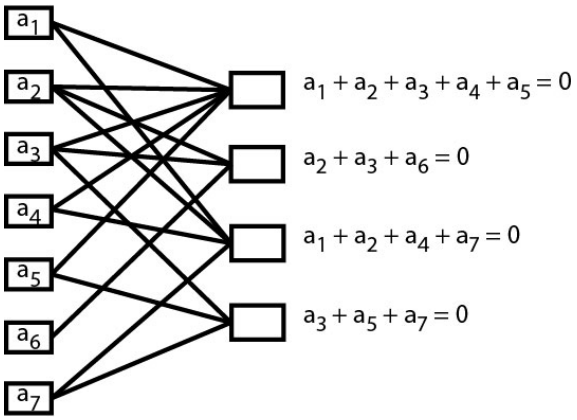

Figure 3.1: Message nodes (left) connected to a few check nodes (right) obtained from the generator matrix in Example 9.

When decoding LDPC codes with message passing, it is more advantageous to use log likelihood ratios (LLR) instead of probabilities. The LLR value for a random variable X, is defined as

$$\mathrm{LLR}(X) = \log \frac{\Pr(X=0)}{\Pr(X=1)}, \tag{3.13}$$

and the conditional LLR, given the variable Y, is defined as

$$\mathrm{LLR}(X|Y) = \log \frac{\Pr(X=0|Y)}{\Pr(X=1|Y)}, \tag{3.14}$$

Given a uniformly distributed variable x and d independent observed variables, y_i, the following formula can be derived

$$\mathrm{LLR}(x|y_1,\ldots,y_d) = \sum_{i=1}^{d} \mathrm{LLR}(x|y_i). \tag{3.15}$$

To calculate the LLR value which is sent from the check nodes to the message nodes, knowledge about how to compute $\mathrm{LLR}(x_1 \oplus x_2 \oplus \cdots \oplus x_d | y_1, \ldots, y_d)$ is needed. This LLR value is derived from $\mathrm{LLR}(x_i|y_i)$, $1 \leq i \leq d$ and the assumption that all observed y_i are independent of each other. By doing some derivations the following formula is received

$$\mathrm{LLR}(x_1 \oplus x_2 \oplus \cdots \oplus x_d | y_1, \ldots, y_d) = \log \frac{1 + (\prod_i \tanh(\mathrm{LLR}_i/2))}{1 - (\prod_i \tanh(\mathrm{LLR}_i/2))}, \tag{3.16}$$

where $\mathrm{LLR}_i = \mathrm{LLR}(x_i|y_i)$.

In round 0 of a message passing algorithm, the *a priori* information $\mathrm{LLR}(x_i|y_i)$, where the symbol y_i is observed by the channel, is calculated for each message node and sent to the check nodes. If a BSC with correlation probability p is used, $\mathrm{LLR}(x_i|y_i)$ is calculated by the formula

$$\mathrm{LLR}(x_i|y_i) = \log(p) - \log(1-p), \tag{3.17}$$

for $y_i = 0$ and the negative value if $y_i = 1$. For simplicity the conditional part will be left out from here on and only $\mathrm{LLR}(x_i)$ will be used.

From round 1 and on, the *extrinsic* information will be calculated for each message node a_n, by the parity-check relations of the check nodes and the formula

$$c_j = \mathrm{LLR}(a_{i_1} + \cdots + a_{i_{w-1}}),\ a_i \neq a_n, \tag{3.18}$$

where w is the weight of the parity-check relation. The message node a_n, will then use the received extrinsic information and calculate the new LLR value, $\mathrm{LLR}(a_n)$, according to

$$\mathrm{LLR}(a_n) = \mathrm{LLR}(a_n) + \sum_{j=1}^{m} c_j, \tag{3.19}$$

where c_j is the extrinsic information calculated with Formula 3.18 for each check node and m the total number check nodes connected to the message node.

3.5 Algorithm B LLR

Meier and Staffelbach's Algorithm B from Section 3.3 can be reformulated using decoding of LDPC codes, as described in Section 3.4, and was originally described by Canteaut and Trabbia in [4]. In this version of Algorithm B each position in the sequence $\mathbf{z}$ is a message node and all generated relations are check nodes, connected to message nodes in the relation. The initial LLR value for each message node a_n is calculated with Formula 3.17, after the initialization the LLR value is calculated for each round by Formula 3.18 and sent back to the corresponding message node.

Algorithm B LLR:

1. Set the probability of all positions to p and calculate the corresponding $\mathrm{LLR}(a_n)$ for $n = 1, \ldots, N$, according to Formula 3.17.

2. All message nodes sends the calculated LLR value to all connected check nodes.

3. Each check node calculates the extrinsic information according to Formula 3.18 for each $\mathrm{LLR}(a_n)$ and sends it back to the corresponding message node.

4. Each message node updates its LLR value by adding the received $\mathrm{LLR}(a_n)$. If the values have not converged enough in either direction, go back to 2.

3.6 Algorithm B LLR modified

A modified LLR version of Meier and Staffelbach's Algorithm B was presented by Ågren, Löndahl, Hell and Johansson, in [5]. In this version of the algorithm weight 3 and 4 relations are used. The relations are then divided into three different type nodes. The weight 3 relations are called type I, with the form in Formula 3.12 and their extrinsic information is calculated with Formula 3.18.

The relations of weight 4 are divided into two types II-a and II-b. Before the weight 4 relations can be divided into different types, they are divided into different lists of pairs. Each one of these lists are equal to a fixed point $e(k)$ which has a unknown value equal to each separate pair in the list. The lists will look like

$$e(k) = a_{i_1} + a_{i_2} = \cdots = a_{i_{v-1}} + a_{i_v}, \tag{3.20}$$

where each $i = 1, \ldots, N$ and no i equals any other i in the same list. These lists can now be classified as type II-a, if $v > 4$, or as type II-b, if $v = 4$.

When all weight 4 relations are classified, knowledge about how to do the computation of the extrinsic information is needed. The update each message node receives from a check node of type II-a will be $\mathrm{LLR}''(a_n)$, where a_n is the message node which will receive the update. To find this LLR value, one first calculates the LLR value for the fixed position $e(k)$ of a list as

$$\mathrm{LLR}(e(k)) = \mathrm{LLR}(a_{i_1} + a_{i_2}) + \cdots + \mathrm{LLR}(a_{i_{v-1}} + a_{i_v}). \tag{3.21}$$

The next step will be to compute $\mathrm{LLR}(e(k))$ for each pair in the list,

$$\mathrm{LLR}'(e(k)) = \mathrm{LLR}(e(k)) - \mathrm{LLR}(a_{i_{j-1}} + a_{i_j}), \tag{3.22}$$

where $(a_{i_{j-1}}, a_{i_j})$ is the current pair and j is even. Because $a_{i_j} = e(k) + a_{i_{j-1}}$, the extrinsic information will be calculated with

$$\mathrm{LLR}''(a_{i_j}) = \mathrm{LLR}(e(k) + a_{i_{j-1}}), \tag{3.23}$$

which is computed with $\mathrm{LLR}'(e(k))$ and $\mathrm{LLR}(a_{i_{j-1}})$. This LLR value is then sent back to the message node a_{i_j}. By switching a_{i_j} in Formula 3.23 with $a_{i_{j-1}}$ the extrinsic information for the message node $a_{i_{j-1}}$, will instead be calculated.

The check nodes of type II-b are on the form from Formula 3.12 and because of that the LLR value for each check node will be calculated with Formula 3.18.

Algorithm B LLR Modified:

1. Set the probability of all positions to p and calculate the corresponding $\mathrm{LLR}(a_n)$ for $n = 1, \ldots, N$, according to Formula 3.17.

2. All message nodes sends the calculated LLR value to all connected check nodes.

3. Type I check nodes on the form from Formula 3.12 with the weight 3, calculates their extrinsic information according to Formula 3.18 and sends the information back to the corresponding message node.

4. Type II-a check nodes on the form 3.20, calculates their extrinsic information in three steps. First $\mathrm{LLR}(e(k))$ is computed with Formula 3.21, secondly $\mathrm{LLR}'(e(k))$ is computed by using Formula 3.22. Finally the extrinsic value is calculated with Formula 3.23 and then sent back to the corresponding message node.

5. Type II-b check nodes are on the form from Formula 3.12 with weight 4, calculates the extrinsic information according to Formula 3.18 and sends the information back to the corresponding message node.

6. If the LLR values of the message nodes have not converged enough in any direction, go back to 2.

Chapter 4
Implementation

4.1 Preliminaries

To be able to perform the simulations, the following have been implemented:
A keystream generator modeled by a LFSR and a BSC.
A hash table to store, sort and handle pairs of positions.
A preprocessing function which generates the pairs of positions according to the shift and multiple method. This function is used for low wight polynomials, to find parity-check relations.
A preprocessing function which generates the pairs of positions by combining all possible pairs and storing them in the hash table. This function is used for high weight polynomials, to find low weight parity-check relations.
The algorithms A, B hard decisions, B soft decisions, original LLR and modified LLR.

4.2 Running the Program

To run the program, the command
*./Run 'poly-file' 'state-file' 'N' 'p' 'algorithm' 'output path' '#rounds' [options]** is used and the input parameters should be specified according to the following list:

poly-file	A file describing the polynomial, in this file each line contains the exponent of a x in the polynomial.
state-file	A file describing the initial state of the LFSR on a single line with 0's and 1's, starting with the least significant bit.
N	An integer value, the length of the output sequence from the LFSR will be be set to this value.
p	An integer value between $0 - 100$, this integer will define p in the crossover probability $1 - p$, used in the BSC.
algorithm	Decides the algorithm. Possible values are *A*, *Bhard*, *Bsoft*, *LLR* or *LLRm*
output path	The path to where the outputs from the program will be stored.

#rounds	The total number of times the chosen algorithm will be run.
printhash	(Optional) Makes the program print the hash table.
printpos	(Optional) Makes the program print the total number of relations, satisfied relations, type II-a relations and type II-b relations for each position.
no_w3	(Optional) Disables the use of weight 3 relations in the program.
no_a	(Optional) Disables the use of weight 4 relations of type II-a in the program.
no_b	(Optional) Disables the use of weight 4 relations of type II-b in the program.

Example 10 show what a correct command can look like.

Example 10. *The command used:*
./Run polynomial.txt initial-state.txt 5000 Bsoft 90 ./outputs 1000

The input file polynomial.txt *should look something like*

3
7
9
10

this would translate to the polynomial $f(x) = 1 + x^3 + x^7 + x^9 + x^{10}$.
The input file initial-state.txt *should look something like*

1010101011

this would set the initial state of the LFSR to 1010101011 *and the length* $l = 10$.
The input 5000 *will set the noisy sequence to the length* 5000.
The input Bsoft *will set the algorithm which is used to Meier and Staffelbach's Algorithm B with soft decisions.*
The input 90 *will set the probability to* $p = 0.9$ *and the crossover probability used in the BSC will then be* 0.1.
The input ./outputs *will tell the program that a outputs should be saved to the folder* outputs*, which should exist in the current folder. The last input* 1000 *tells the program to simulate 1000 runs of the chosen algorithm.*

When the program starts, it will first initialize variables and arrays needed in the program. When the initializations are done, the number of relations for each positions will be calculated by the function `calculateMeanM`. Next the program will check if any options are set in the input arguments to the program, if the option *printpos* is set the function `printPositions` will be called and print the total number of relations for each positions and if the option *printhash* is set the function `printHash` will be called and print the hash table to a file. Then it will start a `for`-loop which will run until the input #rounds is reached. For each round the random generator will be reset with a new seed value, which will be

the current number of seconds from 00:00 hours, Jan 1, 1970 UTC added with the current round number and then the new sequence **z** will be generated by the LFSR. The last thing the `for`-loop will do is to call the function `resetPositions`, which will reset each `position_t` structs variables, except for the variable `g_matrix`, to their initial values.

4.3 Preprocessing

4.3.1 LFSR and the Keystream Generator

To start the LFSR, the function `runLFSR` is called. This function generates the output sequence **a** from the LFSR, and also the output sequence **z** from the BSC. The LFSR is represented by a $64 - bit$ integer in which every bit represents a position in the LFSR. This representation of the LFSR limits the length to $l = 64$, but this is enough for this implementation.

The next state of the LFSR is obtained by the function `nextState` and shifts the $64 - bit$ integer one time and the next input bit will be determined by adding the output bit with the bits at the positions, which corresponds to a feedback tap in the feedback polynomial. The output bit are returned to `runLFSR` and stored in an array and will represent the output sequence $\mathbf{a} = a_1, a_2, \ldots, a_N$ from the LFSR. The function `bsc` represents a BSC with crossover probability $1 - p$ and $p > 0.5$ and will add some noise to the sequence **a** and return the noisy sequence $\mathbf{z} = z_1, z_2, \ldots, z_N$, which is the keystream.

4.3.2 Initialization

Before any relations between the positions can be obtained, some initializations have to be made. First the function `initPositions` will initialize N different structs of the type `position_t`, this struct contains the variables `m`, `h`, `g_matrix`, `llr` which are initialized to *0* and the variable `p`, which is initialized to the probability, which was given to the program as an argument.

The function `initGMatrix` will initialize the generator matrix, which is needed for the generation of the relations between the positions and it is also used by Algorithm A during the simulation. The last initialization function is `hashExists`, this function will check if the *hash* folder exists. If the folder exists the generation of the hash table will be skipped and the simulation of the chosen algorithm will start running. If the folder does not exist, it will be created and the generation of the hash table will start. The folder *hash* will then be filled with one file for each hash value in the hash table.

4.3.3 Finding Relations

There are two ways of finding relations between positions, the way used is chosen by determining the number of taps in the used feedback polynomial. If the number of taps is equal to two, the shift and squaring method described in Section 3.1 will be used. This will be done with a call to the function `makeHashTableLow`.

If the number of taps of the used feedback polynomial is greater than two, the function `makeHashTable` will be called. This function iterates over all positions and for each pair of positions, the `g_matrix` variables will be XOR-ed and if this XOR-ed sum does not equals zero, a `pair_t` struct is created. This struct has two variables, the variable `next`, which points at the next `pair_t` struct and the variable `pos`, which is an array holding the two positions for this pair. The created struct will then be added to the hash table.

4.3.4 The Hash Table

The hash table is allocated by the function `allocateHashTable` and will have a size of $2^{\frac{l}{2}}$, where l is the length of the LFSR. The hash function used for this hash table is

$$hash\,value \;=\; sum\;\&\;(size\,of\,hash\,table - 1)$$

and is calculated by a call to the function `hash`. Each hash value in the hash table points to a linked list of `head_t` structs, this struct has four variables, `next` which points at the next `head_t`, `first_pair` which points at the first `pair_t` struct, `nbr_of_pairs` which is the number of `pair_t` structs in the linked list and `sum` which is the XOR-ed sum for any `pair_t` in the linked list, pointed to by `first_pair`

When a new `pair_t` struct is going to be added to the hash table, the function `addPair` will be called. This function will use the function `lookupHead` to check if there already exists a `head_t` struct, which has a sum equal to that of the new `pair_t`. If such a `head_t` struct exists the new `pair_t` will be added first to the linked list pointed to by `first_pair` and if it does not exist a new `head_t` struct will be created to store the new `pair_t` struct.

4.3.5 Writing to and Reading from the Disk

When running the program on long LFSR output sequences, the creation of the hash table can take a very long time and sometimes it is not possible to keep all possible pairs in the hash table in the RAM at the same time. Therefore the hash table needs to be written to the disk. This process is started by a call to the function `memoryToDisk`. This function will iterate through the hash table and for each hash value, print the positions for the current `pair_t` struct to a file with the hash value as the filename.

All hash values have a corresponding file. These files are created by the function `createFiles`, before the hash table is generated. Each of these files has

the following structure, the first line has a value which will tell the program how many pairs the file contains and then each line contains a pair on the form "positions1:positions2". When all pairs have been generated and written to the corresponding file the function `addNbrOfPairs` will be called and update the total number of pairs in each file. If this function finds a file which does not have any pairs, it will remove the file.

When the hash table is complete and is going to be used for the computations, a function with the name `readFiles` will be called. This function reads files and store the pairs in `pair_t` structs, which then is inserted into the hash table. The function keeps reading files until the max number of `pair_t` structs has been reached or there is no more files to be read.

4.3.6 Handle Useless Pairs

When generating the hash table, pairs that is not part of any weight 3 or weight 4 relations will occur. These pairs are of no use, and will only waste memory and make the computations slower, therefore the function `removeUnusedPairs` is called to remove these pairs.

The function `removeUnusedPairs` will first read as many hash table files as possible and add the pairs to the hash table. Then the function will iterate through the hash table and each `head_t` struct which only have one `pair_t` struct, will be removed from the hash table and put into a temporary list. All the `head_t` structs in this temporary list, have no pair which is part of a weight 4 relation, but they can still be part of a wright 3 relation. Therefore, each pairs XOR-ed sum will be compared to each positions `g_matrix` variable, if a match is found the `head_t` struct will be put back into the hash table. All `head_t` structs which still exist in the list after these two checks are finished, will be permanently removed from the hash table.

Now a new file with the name *read*#, where # is a number, will be created by the function `createFiles` and this file will be filled with all pairs in the `pair_t` structs, currently in the hash table. This will create a large file, which contains all the pairs which can be put into the hash table at the same time without overflowing the memory. Having this large file instead of all the smaller files, will make only one read necessary the next time all pairs need to be loaded into the hash table. The merging of the files also reduce the number of files to the point when it is possible to keep all files open until they are not needed any more and this will reduce the time it takes to read in a new part of the hash table.

4.4 Algorithms

4.4.1 Algorithm A

Algorithm A is started with a call to the function `runAlgA`. This function will first call the function `computeS`, which will calculate the s value for the given probability p, according to Formula 3.2. Then the new probability p^* will be calculated for each position with the function `computePStar`, according to Formula 3.1. Next a call to the function `findInitState` will start the algorithm to find the initial state of the LFSR.

The function `findInitState` will call the function `getSortedPos`, which will sort all the positions by their probabilities, with the highest at positions 0 of the array and the lowest at the last position of the array. The l positions with highest probability will then be chosen, where l equals the length of the LFSR, and each of these positions `g_matrix` variable will then be shifted one step to the left and added to an array. The variables stored in the array will be XOR-ed with the value of the corresponding positions bit value, in the sequence $\mathbf{z}$.

The array will now contain l bit sequences all containing $l+1$ bits and now the bits $2, \ldots, l+1$ will now form a system of linear equations. Solving this system will make the least significant bits take the value of the initial state of the LFSR. If there is a dependency between any of the equations in the array, more equations will be added to make the system independent. If an independent system can not find the initial state of the LFSR, a few positions believed to be correct can have the wrong value. Then the program will try to complement these bits and then retry to find the initial state of the LFSR.

4.4.2 Algorithm B

When a simulation with Algorithm B is run, either a hard decisions version or a soft decisions version can be called with the functions `runAlgBHard` and `runAlgBSoft`. The only difference between the two functions, are during the iterations in each round of the algorithm and therefore the functions will be referred to as one, except for the explanation of the iteration.

In each iteration, all *read#* files will be read by the function `readFiles` and the necessary calculations will be performed. If all pairs can be in the hash table at the same time, the pairs will not be removed until the simulations are finished.

When the function starts, it have to calculate the thresholds p_{thr} and N_{thr}. This is done by a call to the function `computeThresholds`, this function will first generate a binomial coefficients table by a call to the function `computeCoeff` and then the thresholds will be calculated according to Formula 3.9 and 3.11. The generated table will set some limitations on the program, because the average number of relations for each positions, calculated with Formula 3.5, can not be larger than 50.

When the thresholds are calculated the rounds of the algorithm will start, each round has up to five iterations as described in Section 3.3. The hard decision version, that is the function `runAlgBHard`, will for each position use the relations formed by the t feedback taps of the feedback polynomial and with the probabilities of the positions in this iteration, not equal to the current position, calculate the s value with the function `computeS`, according to Formula 3.3. The new p^* will now be calculated with the function `computePStar`, and the new probability will be stored in a temporary array. If $p^* < p_{thr}$, the position will be marked for change. When all positions have calculated a new p^* probability, the `p` variable in each positions `position_t` struct will be set to the corresponding value in the temporary array. If five iterations have been made or the number of positions marked for change exceeds N_{thr}, the current rounds iterations are finished.

The soft decisions version of the algorithm, called by the function `runAlgBSoft`, will consider all the generated relations for a position when calculating the new probabilities. A s value will be calculated for each relation with the function *computeS* and depending on if the relation is satisfied or not, it will be multiplied to 2 of 4 variables. These variables represents the products $\prod_{j\in\mathbb{H}}(s_j)$, $\prod_{j\in\mathbb{H}}(1-s_j)$, $\prod_{j\in\mathbb{M}\setminus\mathbb{H}}(s_j)$ and $\prod_{j\in\mathbb{M}\setminus\mathbb{H}}(1-s_j)$. When all the products of the s values have been calculated, the p^* probability for each position will be calculated according to Formula 3.4 and stored in a temporary array. If $p^* < p_{thr}$, the position will be marked for change. Each positions `p` variable will then be set to the new p^*. If five iterations have been made this round or the number of positions marked for change exceeds N_{thr}, the current rounds iterations are finished.

After the iterations of a round are finished, the positions in **z**, with a probability below p_{thr}, will be complemented and if the function called was `runAlgBHard`, the number of relations which is satisfied for each position will also be recalculated. In the end of the round, the number of equal positions in the two sequences **z** and **a** will be computed and if all positions are equal the algorithm is done, otherwise next round of the algorithm will be started.

4.4.3 Algorithm B LLR Version

When running the LLR version of Algorithm B, the function `runAlgBLLR` is called. The first step in this function will be to initialize all positions LLR value. This is done for each position by calculating Formula 3.17 and then checking the positions corresponding bit value in the sequence **z**. If the position has a bit value of 1, the calculated value will be stored as a negative value and if the position has a bit value of 0 the positive value will be stored. The value will be stored in the `llr` variable, in the `position_t` struct for each positions.

When the iterations starts, each file *read#* will be read by the function `readFiles` and for each part of the hash table the functions `weight3LLR` and `weight4LLROrg` will be called, this part will be skipped if all pairs can be in the hash table at the same time. Both functions will calculate the LLR values for each position according to Formula 3.18 and 3.19, and the new values will be added to a temporary

array, which holds the update for each positions LLR value calculated so far in the current iteration.

The next step is to add the update in the temporary array to the corresponding `position_t` structs variable `llr` to get the new LLR value for the position. Then each positions bit value in the sequence $\mathbf{z}$ will be changed to have the new believed bit value. If $\mathbf{z} = \mathbf{a}$ the algorithm will stop iterating, else the algorithm will continue with the next iteration, using the new LLR values at each position.

4.4.4 Algorithm B LLR Version Modified

The modified version of the LLR version of Algorithm B is called by the function `runAlgBLLRModified`, this function works the same way as the function `runAlgBLLR`, except for the calculations of the LLR values for the weight 4 relations. These relations will instead be calculated by the function `weight4LLRMod`.

The function `weight4LLRMod` will check the variable `nbr_of_pairs` in the `head_t` struct, and then perform the calculations. If the value equals 2, Formula 3.18 and 3.19 will again be used to calculate the LLR value for each position in that relation. If the value is greater than 2, the method described in Section 3.6 will be used for the calculations of the LLR value.

Chapter 5
Result

The simulations presented in this section will test the success rate of finding the initial state of the LFSR with the LLR versions of Algorithm B. This will be done by generating three output sequences for four different primitive feedback polynomials of the degrees 20 and 30.

For each generated sequence, a hash table will be generated which will hold lists of position pairs, which is part of a weight 3 or weight 4 relation. The weight 3 relations will not be used, because they gives the same contribution to the LLR values independent of the used algorithm. For each sequence length and polynomial, the highest crossover probability $1-p$ which has a success rate of at least 10%, will be obtained by doing 1000 simulations for different values of p with the LLR version of Algorithm B. When the crossover probability is obtained, this value will be used to do a larger number of simulations with the LLR version and the modified LLR version of Algorithm B. This larger number of simulations with each algorithm will be performed to make it possible to see a difference in the success rates of the two algorithms.

The primitive polynomials p_{21} and p_{22} will be used for the simulations with a LFSR of length 20,

$$\begin{aligned} p_{21} &= x^{20} + x^{19} + x^{16} + x^{13} + x^{10} + x^8 + 1, \\ p_{22} &= x^{20} + x^{19} + x^{18} + x^{17} + x^{16} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^4 \\ &\quad + x^3 + x^2 + x + 1. \end{aligned}$$

These polynomials will generate hash tables for sequences of the lengths 500, 750 and 1000. For the simulations with a LFSR of length 30, the primitive polynomials p_{31} and p_{32} will be used,

$$\begin{aligned} p_{31} &= x^{30} + x^{29} + x^{25} + x^{24} + x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 \\ &\quad + x + 1, \\ p_{32} &= x^{30} + x^{29} + x^{28} + x^{25} + x^{24} + x^{21} + x^{20} + x^{19} + x^{16} + x^{15} + x^{14} + x^{13} \\ &\quad + x^{12} + x^{11} + x^8 + x^6 + x^4 + x^2 + 1. \end{aligned}$$

The polynomial p_{31} will generate hash tables for sequences of the lengths 6500, 7500 and 8500. The polynomial p_{32} do not have any type II-a relations for the

sequences 6500 and 7500, therefore this polynomial will instead generated hash tables for sequences of the lengths 8000, 9000 and 10000.
Information about the total number of relations for each sequence length and polynomial of degree 20 and 30, can be found in Table 5.1 and 5.4 respectively. The tables shows the average number of relations for each position, m, the total number of weight 4 relations and the total number of type II-a and type II-b check nodes the hash table for each sequence of the length N contains.

The simulation results for p_{21} and p_{22} are shown in Table 5.2 and 5.3, and for p_{31} and p_{32} in Table 5.5 and 5.6. The numbers in the parentheses are the normal approximation of the standard deviation. The standard deviation will be used to determine the significance of the decimals in the success rate of the algorithms.

Polynomial	N	m	Weight 4	Type II-a	Type II-b
p_{21}	500	56	6978	176	6450
	750	191	35799	2254	28866
	1000	471	117627	11849	77296
p_{22}	500	111	13860	1332	9864
	750	292	54786	6578	32871
	1000	557	139230	17810	73567

Table 5.1: A summary of the total number of relations for each output sequence of the length N, generated by p_{21} and p_{22}. Weight 4 is the total amount and m is the average per position.

N	$1-p$	Algorithm	Number of successful runs of 65536	Success rate
500	0.29	LLR	9094($\pm$89)	0.139($\pm$0.001)
		LLR mod	9309($\pm$90)	0.142($\pm$0.001)
750	0.36	LLR	7983($\pm$84)	0.122($\pm$0.001)
		LLR mod	8216($\pm$85)	0.125($\pm$0.001)
1000	0.39	LLR	7312($\pm$81)	0.112($\pm$0.001)
		LLR mod	7327($\pm$81)	0.112($\pm$0.001)

Table 5.2: Simulation results for p_{21}.

N	$1-p$	Algorithm	Number of successful runs of 65536	Success rate
500	0.34	LLR	10196($\pm$93)	0.156($\pm$0.001)
		LLR mod	10335($\pm$94)	0.158($\pm$0.001)
750	0.38	LLR	7344($\pm$81)	0.112($\pm$0.001)
		LLR mod	7391($\pm$81)	0.113($\pm$0.001)
1000	0.39	LLR	11121($\pm$97)	0.170($\pm$0.001)
		LLR mod	11084($\pm$96)	0.169($\pm$0.001)

Table 5.3: Simulation results for p_{22}.

Polynomial	N	m	Weight 4	Type II-a	Type II-b
p_{31}	6500	112	181653	164	181161
	7500	184	345450	2062	339264
	8500	281	596619	7338	574605
p_{32}	8000	246	492540	387	491379
	9000	367	824868	6899	804171
	10000	520	1299435	17949	1242570

Table 5.4: A summary of the total number of relations for each output sequence of the length N, generated by p_{31} and p_{32}. Weight 4 is the total amount and m is the average per position.

N	$1-p$	Algorithm	Number of successful runs of 25536	Success rate
6500	0.32	LLR	2942($\pm$52)	0.115($\pm$0.002)
		LLR mod	3010($\pm$52)	0.118($\pm$0.002)
7500	0.34	LLR	3169($\pm$53)	0.124($\pm$0.002)
		LLR mod	3162($\pm$53)	0.124($\pm$0.002)
8500	0.35	LLR	8443($\pm$76)	0.331($\pm$0.003)
		LLR mod	8595($\pm$76)	0.337($\pm$0.003)

Table 5.5: Simulation results for p_{31}.

N	$1-p$	Algorithm	Number of successful runs of 25536	Success rate
8000	0.35	LLR	3364($\pm$55)	0.132($\pm$0.002)
		LLR mod	3357($\pm$54)	0.131($\pm$0.002)
9000	0.36	LLR	7372($\pm$73)	0.289($\pm$0.003)
		LLR mod	7196($\pm$72)	0.282($\pm$0.003)
10000	0.37	LLR	9250($\pm$77)	0.362($\pm$0.003)
		LLR mod	9307($\pm$77)	0.364($\pm$0.003)

Table 5.6: Simulation results for p_{32}.

Chapter 6
Conclusion

6.1 About the Excluded Algorithms

Meier and Staffelbach's Algorithm A, B hard decisions and B soft decisions were excluded from the simulations for several reasons. A lot of time have, however, been spent on working with the algorithms and they are a crucial part in the understanding of fast correlation attacks, and therefore it is worth mentioning why they were excluded.

Algorithm A was excluded from the simulations because it is not an iterative algorithm. Also the success of the algorithm depends on the existence of at least L positions independent of each other and the number of bits allowed to be complemented in the last step of the algorithm. Therefore Algorithm A can have different success rates for the same crossover probability. It will have a high success rate, if enough bits are allowed to be complemented, and a low success rate, if only a few bits are allowed to be complemented, for the same crossover probability.

Algorithm B, both hard and soft decisions, were excluded from the simulations because of the way these two algorithms is implemented. First of all the amounts of relations between the positions generated by the implemented preprocessing, even for a small output sequence, gives a large average number of relations, m, for a position. If the calculations of the thresholds are implemented as Meier and Staffelbach described them in [2]. If the value m is big, the implementation have to handle big integers, which this implementation does not. Also the implementation of the algorithms does only handle relations of weight 3, as Meier and Staffelbach described the Algorithm B, and therefore they would not be able to be compared to the LLR versions of the algorithm, which handles relations of both weight 3 and weight 4.

These three algorithms, or at least Algorithm B hard and soft decisions, could of course have been properly compared to the LLR versions with a different implementation approach.

6.2 About the Implementation

Looking at the implementation of the preprocessing, it is easy to see that it is not the optimal approach, regarding time efficiency and the amount of memory needed. But this approach is guaranteed to generate a large number of relations for each position and this was a desired property of the implementation.

The first phase of the preprocessing is the generation of the pairs used to form the relations and storing them from the memory to the hard drive. The generation is performed by two `for`-loops, which have a time complexity of $\mathcal{O}(N^2)$ and will generate approximately $N^2/2$ pairs, where N is the length of the output sequence. All these pairs then have to be written from the memory to the hard drive, which will force another iteration over all pairs to be performed, which also has a time complexity $\mathcal{O}(N^2)$, given a constant write operation.

The second phase of the preprocessing is the removing of pairs which is not used. This will first make all pairs being read into memory and then iterated over to temporary remove all relations, which are not part of a weight 4 relation, both actions have a time complexity of $\mathcal{O}(N^2)$. Last the temporary removed relations need to iterate over all N positions and putting back all pairs which are part of a weight 3 relations, this gives a time complexity of $\mathcal{O}(R \cdot N)$, where R is the temporary removed pairs.

At first there will be approximately $N^2/2$ pairs, but this number will be reduced to approximately $N^2/2 - R = X$ pairs. If the used feedback polynomial is primitive and the output sequence is shorter than the period of the LFSR, a $X \ll N^2/2$ will be guaranteed.

When a run of Algorithm B LLR or Algorithm B LLR modified is started, they will first need to read all X pairs into memory form the hard drive, which approximately has a time complexity of $\mathcal{O}(X)$. Then all the weight 3 relations need to be handled, which approximately have a time complexity of $\mathcal{O}(N)$. Last the weight 4 relations should be handled, which have an approximate time complexity of $\mathcal{O}(X)$.

The preprocessing could obviously be improved by using a different method to generate the pairs. By looking in Table 5.1 and 5.4, one can clearly see that the generated number of relations is really large. Even if a large number of relations were desired, this could have been lowered with a different approach. One approach could, as example, be to set a limit on how many relations a position was allowed to be a part of. Another approach could have been to implement any of the methods for generation of parity-check relations, described in [5].

The algorithms do not have any obvious way of improving their performance speed. The bottleneck here is instead the break condition for a failed simulation. A simulation fails if it does too many iteration without increasing the correct number of bits in the output sequence or by hitting the maximum allowed iterations limit. This means that a failed simulation has to do a lot more work than a successful

simulation. The simulation would gain in performance speed by lowering these two break conditions, but this could increase the number of false negatives in the results. It would also be possible to gain some performance speed by adding a third break condition, which compares the number of correct bits in the output sequence from the last iteration with the number of correct bits in the output sequence for the current iteration. If the correct number of bits for the current iteration is less than the number of corrects bits for the last iteration, the simulation should be counted as a fail. This would, however, increase the number of false negatives in the result.

An improvement that would benefit all phases of the program is faster writing to and reading from the hard drive. This would greatly decrease the time for the preprocessing phase and it would also decrease the time for an iteration of the algorithms, when the combination of the feedback polynomial and the length of the output sequence generates more pairs than the memory can handle at the same time.

6.3 About the Results

The total number of obtained relations will depend on the output sequence length and the used feedback polynomial. This means that the algorithms cannot guarantee a certain success rate for a crossover probability, when two output sequences generated by different feedback polynomials are used. This can be seen in Table 5.2 and 5.3 and by looking in Table 5.1, one can see that this is a result of the difference in the total number of relations for the output sequence.

By looking in Table 5.2 and 5.3, one can see that in most of the simulations, with a feedback polynomial of degree 20, the modified LLR algorithm outperforms the original LLR algorithm. This difference is, however, small and when the total number of relations is higher than 35799 and one considers the standard deviation for the success rates of the two algorithms, they do approximate each other.

The simulations with a feedback polynomial of degree 30, give more varying success rates and in all but one of the cases, the success rates are in reach of each other when taking the standard deviations into consideration. This can be seen by looking in Table 5.5 and 5.6. The reason that the standard deviations are larger, is due to the smaller number of performed simulations. For the feedback polynomial p_{32}, sequence length 9000, the original algorithm has a better success rate than the modified algorithm. Also, the success rates are not in reach of each other when the standard deviation is considered. Additional simulations for this specific case showed that the gap between the algorithms success rates closes. Because of this, it should be safe to assume that the original LLR algorithm has been a bit more lucky in the presented simulation.

The results do suggest, that when a large number of relations between the positions are used, as in these simulations, the success rate of finding the initial state for the

LFSR is approximately the same for both algorithms. To verify this, however, a lot more simulations have to be performed. It would especially be needed for the output sequences generated by the feedback polynomials of degree 30, where only a small number of simulations have been performed and the standard deviation is high. Running simulations with feedback polynomials of an even higher degree would also be suitable, to further compare the algorithms.

The small number of simulations for the feedback polynomials of degree 30 and that no simulations of feedback polynomials with a degrees higher than 30 was performed in this thesis was due to time limitations.

References

[1] N. Smart. Cryptography: An Introduction, 3rd Edition. 3:37-48, 115-116.

[2] W. Meier and O. Staffelbach. Fast Correlation Attacks on Certain Stream Ciphers. *Journal of Cryptology*, 1:159-176, 1989. December 19, 2008.

[3] R. G. Gallager. Low-Density Parity-Check codes. *IEEE Transaction on Information Theory*, 21-28, 1962.

[4] A. Canteaut and M. Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In B. Preneel, editor, *Advances in Cryptology-EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science* 573-588. Springer-Verlag 2000.

[5] M. Ågren, C. Löndahl, M. Hell and T. Johansson. A Survey on Fast Correlation Attacks. *Dept. of Electrical and Information Technology*, Lund. 1-30.

Appendix A

Implementation Documentation

A.1 run.c

Contains the `main`-function and a thread function to print the progress during the generation of the hash table.

```
void* printStatus();

int main(int argc, char* argv[]);
```

`printStatus`	This function is started in a separate thread and waits for the user to request the status. To print the status a single 'u' should be sent as input.
`main`	The `main`-function, allocates the arrays and the hash table which are needed for a run of the program.

A.2 structs.h

Only a header file, in which all of the structs are declared. All other files should include this header.

```
typedef struct position_t position_t;

typedef struct pair_t pair_t;

typedef struct head_t head_t;

typedef struct hash_table_t hash_table_t;

typedef struct buffer_t buffer_t;
```

`position_t` Represents a position in the output sequence from the LFSR. The struct contains the following variables:

- `m` The total number of relations this position is involved in.
- `h` The number of relations this position is involved in which are satisfied.
- `p` The probability that this position has the correct bit value.
- `g_matrix` The corresponding generator matrix column.
- `llr` The log likelihood ratio that this position hash the correct bit value.

`pair_t` Represents a pair of positions. The struct contains the following variables:

- `next` A pointer to the next `pair_t` struct.
- `pos[2]` The position pair in this struct. Always stored as `pos[0] < pos[1]`.

`head_t` This struct is used to sort the `pair_t` structs on their XOR-ed sum. The struct contains the following variables:

- `next` A pointer to the next `head_t` struct.
- `first_pair` A pointer to the first `pair_t` struct in a linked list of pairs.
- `sum` The XOR-ed sum of each pair, pointed to by `first_pair`.
- `nbr_of_pairs` The total number of pairs in the linked list `first_pair` points to.

`hash_table_t` Represents the hash table. The struct contains the following variables:

- `size` The size of the hash table.
- `table` An array of pointers for each hash value in the hash table.

`buffer_t` This struct is used when writing to and reading from files. The struct contains the following variables:

- `file` The filename of the file which should be opened, not always used.
- `nbr_of_pairs` The number of pairs this buffer holds.
- `pairs` An array of `pair_t` structs, which are present in the buffer.

A.3 lfsr.h

Handles the LFSR and BSC functions.

```
void initLFSR(char* poly_path, char* state_path, int* tap_pos,
              int* taps, uint64_t* lfsr, int* lfsr_len);

int nextState(uint64_t* lfsr, int* tap_pos, int taps,
              int lfsr_len);

int BSC(int bit, long double p, int* changed);

void runLFSR(char* poly_path, char* state_path, int* z_seq,
             int z_len, int* tap_pos, int* taps, int* lfsr_len,
             long double p, int* a_seq);
```

`initLFSR` Initiates the LFSR according to the polynomial and state which the user specified.

- `poly_path` The filename of the file describing the polynomial.
- `state_path` The filename of the file describing the initial state.
- `tap_pos` The array the taps of the feedback polynomial will be stored in.
- `taps` The number of feedback taps.
- `lfsr` The variable representing the LFSR.
- `lfsr_len` The length of the LFSR.

`nextState` Shifts to the next state of the LFSR. Returns the output bit.

- `lfsr` The variable representing the LFSR.
- `tap_pos` The array the taps of the feedback polynomial will be stored in.
- `taps` The number of feedback taps.
- `lfsr_len` The length of the LFSR.

`BSC` Represents the binary symmetric channel. Returns the new value of the input bit.

- `bit` The input bit to the BSC.
- `p` The probability positions has the correct bit value.
- `changed` A counter to keep track of the number of changed bits.

`runLFSR`	Runs the LFSR.	
	`poly_path`	The filename of the file describing the polynomial.
	`state_path`	The filename of the file describing the initial state.
	`z_seq`	The array were all output bits from the BSC will be stored.
	`z_len`	The length of the output sequence.
	`tap_pos`	The array the taps of the feedback polynomial will be stored in.
	`taps`	The number of feedback taps.
	`lfsr_len`	The length of the LFSR.
	`p`	The probability positions has the correct bit value.
	`a_seq`	The array were all output bits from the LFSR will be stored.

A.4 position.h

Handles the allocation and removing of the positions. The checking of satisfied relations are also done here.

```
void initPositions(position_t** positions, int z_len,
                   long double p);

void freePositions(position_t** positions, int z_len);

void checkSatisfied(position_t** positions, int* eq, int weight,
                    int* z_seq);

void checkW3(hash_table_t* hash_table, position_t** positions,
             int* z_seq, int z_len);

void checkW4(hash_table_t* hash_table, position_t** positions,
             int* z_seq);

void calculateMeanM(hash_table_t* hash_table,
                    position_t** positions, int* z_seq,
                    int z_len, const char* path);

void resetPositions(position_t** positions, int z_len,
                    long double p);

void printPositions(position_t** positions, int z_len);
```

`initPositions` Allocates and initiates all `z_len` positions.
- `positions` The array of pointers to all positions.
- `z_len` The length of the output sequence.
- `p` The probability positions has the correct bit value.

`freePositions` Frees all `z_len` positions.
- `positions` The array of pointers to all positions.
- `z_len` The length of the output sequence.

`checkSatisfied` Checks if a relations is satisfied.
- `positions` The array of pointers to all positions.
- `eq` The relations which is going to be checked.
- `weight` The weight of the relation.
- `z_seq` The output sequence with noise.

`checkW3`	Checks and counts weight 3 relations.
	`hash_table` The hash table.
	`positions` The array of pointers to all positions.
	`z_seq` The output sequence with noise.
	`z_len` The length of the output sequence.
`checkW4`	Checks and counts weight 4 relations.
	`hash_table` The hash table.
	`positions` The array of pointers to all positions.
	`z_seq` The output sequence with noise.
`calculateMeanM`	Iterates over the hash table and counts the total number of relations for each position.
	`hash_table` The hash table.
	`positions` The array of pointers to all positions.
	`z_seq` The output sequence with noise.
	`z_len` The length of the output sequence.
	`path` The path where the hash table is stored.
`resetPositions`	Resets the `position_t` structs variables to their initial values.
	`positions` The array of pointers to all positions.
	`z_len` The length of the output sequence.
	`p` The probability positions has the correct bit value.
`printPositions`	Print the total number of relations for each position.
	`positions` The array of pointers to all positions.
	`z_len` The length of the output sequence.

A.5 preproc.h

Handles the preprocessing, which is finding relations between the positions by generating the hash table.

```
void printPreProcStatus();

void initGMatrix(int z_len, int lfsr_len, int* tap_pos,
                 int taps, position_t** positions);

void makeHashTableLow(int z_len, int* tap_pos, int taps,
                      int lfsr_len, position_t** positions,
                      const char* path,
                      hash_table_t* hash_table);

void makeHashTable(hash_table_t* hash_table,
                   position_t** positions, int z_len,
                   const char* path);

void hashExists(const char* path, bool* make_hash,
                hash_table_t* hash_table);

void removeUnusedPairs(hash_table_t* hash_table,
                       position_t** positions, int z_len,
                       const char* path);

void runPreProc(double p, int z_len, position_t** positions,
                int* tap_pos, int taps, int lfsr_len,
                const char* path, hash_table_t* hash_table);
```

`printPreProcStatus` Called by the `printStatus`-function and prints the status of the generation of the hash table.

`initGMatrix` Initializes the generator matrix and store each column in the corresponding `position_t` struct.

`z_len`	The length of the output sequence.
`lfsr_len`	The length of the LFSR.
`tap_pos`	The array the taps of the feedback polynomial will be stored in.
`taps`	The number of feedback taps.
`positions`	The array of pointers to all positions.

`makeHashTableLow` Generates the hash table for a polynomial with a weight equal to 2. The function uses the square and shift method.

`z_len`	The length of the output sequence.
`tap_pos`	The array the taps of the feedback polynomial will be stored in.
`taps`	The number of feedback taps.
`lfsr_len`	The length of the LFSR.
`positions`	The array of pointers to all positions.
`path`	The path where the hash table should be stored on the disk.
`hash_table`	The hash table.

`makeHashTable` Generates the hash table by making pairs of each combination of positions in the output sequence.

`hash_table`	The hash table.
`positions`	The array of pointers to all positions.
`z_len`	The length of the output sequence.
`path`	The path where the hash table should be stored on the disk.

`hashExists` Checks if there already exists a hash table, otherwise the necessary files will be created.

`path`	The path where the hash table is stored, if it exists.
`make_hash`	The output sequence with noise.
`hash_table`	The hash table.

`removeUnusedPairs` Removes the pairs which will not contribute to the computation in the algorithms from the hash table. The function also merges the smaller hash value files to fewer larger files.

`hash_table`	The hash table.
`positions`	The array of pointers to all positions.
`z_len`	The length of the output sequence.
`path`	The path were the hash table should exist and the new files will be stored.

`runPreProc`	Run the preprocessing.	
	`p`	The probability a position has the correct bit value.
	`z_len`	The length of the output sequence.
	`positions`	The array of pointers to all positions.
	`tap_pos`	The array the taps of the feedback polynomial will be stored in.
	`taps`	The number of feedback taps.
	`lfsr_len`	The length of the LFSR.
	`path`	The path were the hash table should exist.
	`hash_table`	The hash table.

A.6 hash.h

Handles the allocation, freeing and other operations of the hash table.

```
hash_table_t* allocateHashTable(int lfsr_len);

unsigned int hash(hash_table_t* hash_table, uint64_t sum);

head_t* lookupHead(hash_table_t* hash_table, uint64_t sum);

int addPair(hash_table_t* hash_table, uint64_t sum,
            pair_t* new_pair);

void emptyHashTable(hash_table_t* hash_table,
                    pair_t** pairs_ptrs);

void freeHashTable(hash_table_t* hash_table);
```

`allocateHashTable` Allocates and returns the hash table.
- `lfsr_len` The length of the LFSR.

`hash` Calculates and returns the hash value for the given value.
- `hash_table` The hash table.
- `sum` The value which should be hashed.

`lookupHead` Checks if a certain XOR-ed sum of two pairs is present in the hash table.
- `hash_table` The hash table.
- `sum` The value which should be check if it exists in the hash table.

`addPair` Add a new `pair_t` struct to the hash table.
- `hash_table` The hash table.
- `sum` The XOR-ed sum of the pair.
- `new_pair` The new pair which should be added to the hash table.

`emptyHashTable` Empties the hash table without removing the hash table itself.
- `hash_table` The hash table.
- `pairs_ptrs` Pointers to the allocated arrays of `pair_t` structs which also should be freed. Set to `NULL` if no such arrays exists.

`freeHashTable` Frees the hash table.
- `hash_table` The hash table.

A.7 io.h

Handles the writing to and reading from the disk.

```
void memoryToDisk(hash_table_t* hash_table, char* path,
                  int readfile);

void addNbrOfPairs(int* nbr_of_pairs, int start, int end,
                   const char* path);

int readFiles(hash_table_t* hash_table, position_t** positions,
              struct dirent** filelist, int nbr_of_files,
              int* curr_file, int* count, const char* path,
              pair_t* pairs_ptrs, int pre_proc,
              FILE** open_file);

void createFiles(int start, int end, const char* path,
                 struct dirent** filelist, int* nbr_of_pairs,
                 int readfile);

void printHash(hash_table_t* hash_table, position_t** positions,
               const char* path);
```

`memoryToDisk` Iterates over the hash table and calls `writeBuffer`-function to write the hash table to the disk.

- `hash_table` The hash table.
- `path` The path to were the hash table should be stored.
- `readfile` The index of the next large file to write to during the merging step.

`addNbrOfPairs` Add the total number of `pair_t` structs in a file, at the top of the file.

- `nbr_of_pairs` An array with all files total numbers.
- `start` From which index to start writing.
- `end` To which index to end writing.
- `path` The path to the files.

`readFiles` Calls the `readBuffer`-function and adds the `pair_t` structs to the hash table.

`hash_table`	The hash table.
`positions`	The array of pointers to all positions.
`filelist`	A list of the filenames in the current directory.
`nbr_of_files`	The number of files in the current directory.
`curr_file`	The index of the current file to read.
`count`	A counter of the number of files read.
`path`	The path to the current directory.
`pairs_ptrs`	An array of pointers to keep track of the all the arrays allocated by the `readBuffer`-function.
`pre_proc`	Tells the function whether or not the call is made from the preprocessing step.
`open_file`	An array of open files to read from.

`createFiles` Creates new files and reserves the first 4 bytes for the count of the total number of pairs in the file.

`start`	The index to start at.
`end`	The index to stop at.
`path`	The path were the files should be created.
`filelist`	A list of filenames in the current directory.
`nbr_of_pairs`	An array of the total number of pairs in the different files.
`readfile`	The index of the next large file to write to during the merging step.

`printHash` Iterates over the hash table an prints all values to a file placed at the output path.

`hash_table`	The hash table.
`positions`	The array of pointers to all positions.
`path`	The output path were the file should be placed.

A.8 a.h

Handles the simulations of Meier and Staffelbach's Algorithm A.

```
void printPosWithHighP(position_t** positions, int* pos_by_prob,
                       int* a_seq, int* z_seq, int lfsr_len);

void getSortedPos(int* array, position_t** positions,
                  int z_len);

void findInitState(position_t** positions, int* start_state,
                   int* z_seq, int z_len, int lfsr_len,
                   int* a_seq, long double p);

void runAlgA(int* z_seq, int z_len, int taps, int lfsr_len,
             long double p, position_t** positions,
             int* start_state, int* a_seq);
```

`printPosWithHighP` Prints the `lfsr_len` positions with the highest probability.

`positions`	The array of pointers to all positions.
`pos_by_prob`	The ordered positions.
`a_seq`	The output sequence from the LFSR without noise.
`z_seq`	The output sequence with noise.
`lfsr_len`	The length of the LFSR.

`getSortedPos` Sorts the positions according to their probability, highest first.

`array`	The array where the sorted positions will be stored.
`positions`	The array of pointers to all positions.
`z_len`	The length of the output sequence.

`findInitState` Finds the initial state of the LFSR by solving a system of linear equations.

`positions`	The array of pointers to all positions.
`start_state`	The array where the initial state will be stored.
`z_seq`	The output sequence with noise.
`z_len`	The length of the output sequence.
`lfsr_len`	The length of the LFSR.
`a_seq`	The output sequence from the LFSR without noise.
`p`	The probability a position has the correct bit value.

`runAlgA` Runs a simulation of algorithm A.

`z_seq`	The output sequence with noise.
`z_len`	The length of the output sequence.
`taps`	The number of feedback taps.
`lfsr_len`	The length of the LFSR.
`p`	The probability a position has the correct bit value.
`positions`	The array of pointers to all positions.
`start_state`	The array where the initial state will be stored.
`a_seq`	The output sequence from the LFSR without noise.

A.9 b.h

Handles the simulations of Meier and Staffelbach's Algorithm B, both the soft decisions and the hard decisions version.

```
void computeCoeff(unsigned long long** table, int max);

void computeThresholds(long double p, int lfsr_len, int taps,
                       int* N_thr, long double* P_thr,
                       int z_len);

void runAlgBSoft(int* z_seq, int z_len, int taps, int lfsr_len,
                 long double p, position_t** positions,
                 hash_table_t* hash_table, const char* path,
                 int* a_seq);

void runAlgBHard(int* z_seq, int z_len, int* tap_pos, int taps,
                 int lfsr_len, long double p,
                 position_t** positions,
                 hash_table_t* hash_table, const char* path,
                 int* a_seq);
```

`computeCoeff` Computes the binomial coefficients up to `max` and stores them in a table.
- `table` The binomial coefficients table.
- `max` The max of the table.

`computeThresholds` Computes the thresholds needed in algorithm B.
- `p` The probability a position has the correct bit value.
- `lfsr_len` The length of the LFSR.
- `taps` The number of feedback taps.
- `N_thr` The threshold for the number of changed positions.
- `P_thr` The threshold for the probability.
- `z_len` The length of the output sequence.

`runAlgBSoft` Runs a simulation of algorithm B soft decisions.
- `z_seq` The output sequence with noise.
- `z_len` The length of the output sequence.
- `taps` The number of feedback taps.
- `lfsr_len` The length of the LFSR.
- `p` The probability a position has the correct bit value.
- `positions` The array of pointers to all positions.
- `hash_table` The hash table.
- `path` The path to where the hash table exists.
- `a_seq` The output sequence from the LFSR without noise.

`runAlgBHard` Runs a simulation of algorithm B hard decisions.

`z_seq`	The output sequence with noise.
`z_len`	The length of the output sequence.
`tap_pos`	The array the taps of the feedback polynomial will be stored in.
`taps`	The number of feedback taps.
`lfsr_len`	The length of the LFSR.
`p`	The probability a position has the correct bit value.
`positions`	The array of pointers to all positions.
`hash_table`	The hash table.
`path`	The path to where the hash table exists.
`a_seq`	The output sequence from the LFSR without noise.

A.10 bllr.h

Handles the simulations of the LLR versions of Algorithm B.

```
void weight3LLR(hash_table_t* hash_table,
                position_t** positions, int z_len,
                long double* new_llr);

void weight4LLROrg(hash_table_t* hash_table,
                   position_t** positions,
                   long double* new_llr);

void weight4LLRMod(hash_table_t* hash_table,
                   position_t** positions,
                   long double* new_llr);

void runAlgBLLR(int* z_seq, int z_len, int lfsr_len,
                long double p, position_t** positions,
                int* a_seq, const char* path,
                hash_table_t* hash_table);

void runAlgBLLRModified(int* z_seq, int z_len, int lfsr_len,
                        long double p, position_t** positions,
                        int* a_seq, const char* path,
                        hash_table_t* hash_table);
```

`weight3LLR` Calculates the LLR value for weight 3 relations.

- `hash_table` The hash table.
- `positions` The array of pointers to all positions.
- `z_len` The length of the output sequence.
- `new_llr` An array to store the new LLR values in.

`weight4LLROrg` Calculates the LLR value for weight 4 relations according to the original approach.

- `hash_table` The hash table.
- `positions` The array of pointers to all positions.
- `new_llr` An array to store the new LLR values in.

`weight4LLRMod` Calculates the LLR value for weight 4 relations according to the modified approach.

- `hash_table` The hash table.
- `positions` The array of pointers to all positions.
- `new_llr` An array to store the new LLR values in.

`runAlgBLLR`	Run a simulation of the original LLR approach.	
	`z_seq`	The output sequence with noise.
	`z_len`	The length of the output sequence.
	`lfsr_len`	The length of the LFSR.
	`p`	The probability a position has the correct bit value.
	`positions`	The array of pointers to all positions.
	`a_seq`	The output sequence from the LFSR without noise.
	`path`	The path to where the hash table exists.
	`hash_table`	The hash table.
`runAlgBLLRModified`	Run a simulation of the modified LLR approach.	
	`z_seq`	The output sequence with noise.
	`z_len`	The length of the output sequence.
	`lfsr_len`	The length of the LFSR.
	`p`	The probability a position has the correct bit value.
	`positions`	The array of pointers to all positions.
	`a_seq`	The output sequence from the LFSR without noise.
	`path`	The path to where the hash table exists.
	`hash_table`	The hash table.

A.11 calculation.h

Handles different calculations.

```
long double LLR(void* pos_or_val, int nbr_of_pos,
                position_t** positions);

long double computeS(long double** probs, int t);

int computeM(int N, int k, int t);

long double computePStar(long double p, long double s, int h,
                         int m);
```

`LLR` Calculates the LLR sum value for the given LLR values or the given positions.

- `pos_or_val` Either an array of LLR values or an array of positions.
- `nbr_of_pos` The number of LLR values or positions in the array.
- `positions` The array of pointers to all positions. This is set to `NULL` if the array contains LLR values.

`computeS` Calculates the s value, which algorithm A and B needs.

- `probs` The probabilities for the positions in the relation.
- `t` The number of probabilities in `probs`.

`computeM` Calculates an approximation of the total number of relations for each position.

- `N` The length of the LFSR output sequence.
- `k` The length of the LFSR.
- `t` The number of feedback taps.

`computePStar` Calculates the new probability for a position.

- `p` The current probability for the position.
- `s` The s value for the probabilities of the positions in relations to the current position.
- `h` The total number of satisfied relations for the current position.
- `m` The total number of relations for the position.

A.12 util.h

Functions used in different source files.

```
const char* printBinary(uint64_t x, int nbr_of_bits);

void* xmalloc(size_t size);

void openManyFiles(FILE** f, struct dirent** filelist,
                   int nbr_of_files, const char* path);
```

`printBinary` Prints the binary representation of the given integer `x`.

`x`	The integer to print in binary form.
`nbr_of_bits`	The number of bits of the integer to print.

`xmalloc` Allocates memory and returns a pointer to it. It also handles out of memory errors.

`size`	The size of the memory which should be allocated.

`openManyFiles` Open all files in the directory `path` listed in `filelist`.

`f`	An array where the pointers to the open files will be placed.
`filelist`	A list of all the files in a directory.
`nbr_of_files`	The number of files in `filelist`.
`path`	The path to the directory where the files are located.