New Methods for Cryptanalysis of Stream Ciphers

Håvard Molland

The Selmer Centre
Department of Informatics
University of Bergen
Norway

18th May 2005

Acknowledgments

I would like to express my gratitude to several people for supporting me in completing this thesis.

Firstly, I would like to thank my supervisor professor Tor Helleseth for always having his door open, for his supervising, encouragement and help in writing this thesis. I also wish to thank him for creating a good and relaxed research environment at the Selmer Centre.

Many thanks to Matthew Parker and Håvard Raddum for proof reading and great help in the writing process.

I also wish to thank John Erik Mathiassen who worked with me on Section 3 in our paper “Improved Fast Correlation Attack Using Low Rate Codes”, and for coming up with the brilliant search algorithm in that section.

Thanks to professor Ed Dawson and the rest of the people at Information Security Research Centre in Brisbane for letting me stay for a whole year in a friendly environment. It has been a excellent year with many experiences to remember.

Finally, my deepest gratitude to Elisabeth for all her support and kindness, and for coming with me all the way to Australia for the last year of the Ph.D program.

Introduction to the Thesis

Håvard Molland

1 Background

The *Vernam* cipher [8], also known as *one time pad*, is remarkable simple in theory and provably unconditionally secure, which means that it is impossible to break given unlimited computation power. Let the *keystream* $\mathbf{k} = (k_0, k_1, \ldots, k_t, \ldots)$ be a truly random binary sequence known by the sender and receiver only. Then the binary message (or plaintext) $\mathbf{m}$ is encrypted to the cipher text $\mathbf{c}$ by

$$c_t = k_t \oplus m_t, \tag{1}$$

where $a \oplus b$ is the binary xor operation equivalent to $a + b \bmod 2$. After encryption the cipher text $\mathbf{c}$ is sent to the receiver, who decrypts the message by

$$m_t = k_t \oplus c_t. \tag{2}$$

It is easy to give a simple and intuitive argument for why the cipher is secure. If $\mathbf{k}$ is a truly random sequence, the probability for that m_t equals c_t is exactly $\frac{1}{2}$ in the binary example. In addition, since each bit in $\mathbf{k}$ is independent of all the other bits, knowing some bits in $\mathbf{k}$ will never reveal information about the rest of the keystream. Therefore, regardless of what was previously known about the plain text, knowing c_t gives no extra information.

It is also possible to work with higher alphabets. The letters A,B,C,...,Z can for example be encoded into numbers 0,1,2,...,25. Then $\mathbf{k}$, $\mathbf{m}$ and $\mathbf{c}$ are streams of letters and the encryption is done by $c_t = k_t + m_t \bmod 26$. A similar argument as above can be made to show that this is also unconditionally secure.

The problem with this scheme is that it is not very practical. The whole security is based on that the keystream $\mathbf{k}$, which must have the same length as the message, is secret. In addition we can only use each keystream once. This is easily shown by the following calculation: Let message one be $c_t = k_t \oplus m_t$ and message two be $c'_t = k_t \oplus m'_t$. The attacker can now add the two cipher texts and get

$$c_t \oplus c'_t = k_t \oplus m_t \oplus k_t \oplus m'_t = m_t \oplus m'_t.$$

Since messages often contain much redundancy, we can retrieve m_t and m'_t from $m_t \oplus m'_t$ by, among others techniques, using frequency analysis.

This means that we must send a new keystream to the receiver for each new message we want to transmit. Obviously, we cannot use encryption for this (in that case we could just have sent the message using the encryption algorithm in the first place). Therefore, the secret keystream must be sent using a courier, making the key distribution difficult.

Despite the fact that the one time pad is thought of as unpractical, a non binary version of it was used by the Soviet Union in the 1940's and 1950's to send home messages from the embassy in USA. The USA military stored these messages and revealed that sometimes the same one time pad had been used to encrypt more than one message. This allowed the *Venona* project[1], a highly secret collaboration between USA and Great Britain that lasted for 40 years, to decrypt a fair amount of the traffic. The execution of the couple Ethel Greenglass and Julius Rosenberg was a direct result of this project.

1.1 Stream Ciphers

A solution to the key distribution problem is to use *stream ciphers* which try to emulate the one time pad by expanding a short secret key into a long pseudo random keystream $\mathbf{z} = (z_0, z_1, \dots)$ which has as many of the properties of a truly random stream as possible. This model is not unconditionally secure, since given enough computer power, it is always possible to find the secret key by exhaustive search. Thus, we have lost some of the security, but gained practicality.

Secure stream ciphers must produce keystreams that satisfy the following basic but important properties:

– **Good statistical properties**. There exist many standard statistical tests which can be used to check the statistical properties of a stream. It should not be possible to use any of theses tests to distinguish the keystream from a truly random stream.
– **Confusion and diffusion**. Confusion is to have complex dependency between the keystream $\mathbf{z}$ and the secret key $\mathbf{K}$. Diffusion ensures that a small change, say one bit, in the key give a totally new keystream and cipher text.
– **Guaranteed large period.** It is important that the keystream has a guaranteed large period, so that it does not repeat itself during encryption. This would be the same as using the same keystream twice, and would lead to vulnerabilities similar to those described for the one time pad.

Some concrete properties that will help satisfying the requirements above are

– **High degree of non-linearity**. It should not be possible to find linear approximations that opens up for many different types of attacks.
– **High algebraic degree**. High algebraic degree in equations that involve keybits. This ensures that the attacker cannot calculate the key from the keystream by simply building up and solve an equation set.
– **Balanced**. The average number of 1's and 0's in the stream should be equal. In addition, the average number of sub patterns of bits of different lengths should also be equal.
– **High linear complexity**. The length of the shortest possible linear feedback shift register that can produce the stream. It is important the linear complexity

is high, so that we cannot substitute the generator with a relatively short linear feedback shift register. The shift registers have weak security properties when used alone (See Section 3.2).

The properties above are very important, but may not be sufficient. A cipher can satisfy all the requirements and still be very weak since cryptanalysts typically develop new and previous unknown tests aiming the specific generator they analyze.

At last, a property that is not directly about security, but has a strong influence on it:

- **High efficiency**. In most applications it is very important that the cipher is fast and light, so that is does not slow down the communication, or does not require expensive computer power.

In fact, to day it is relatively easy to design a secure but slow cipher. The hard challenge is to design a cipher that has *both* maximum efficiency and strong security.

1.2 Attacking Stream Ciphers

Cryptanalysis is the science of breaking encryption algorithms or finding weaknesses in them. A method that breaks an encryption algorithm is called an attack. Although there have been a few cipher text only attacks on stream ciphers, see for example [24], most cryptanalysts focus on known plain-/cipher text attacks, where the attacker knows both the cipher text and some of the plain text. Assume we are given some plaintext $\mathbf{m} = (m_0, m_1, \ldots, m_{N-1})$ and all of the cipher text $\mathbf{c}$. We can easily obtain parts of the corresponding keystream $\mathbf{z}$ by $z_t = m_t \oplus c_t$. When this is done, we focus on the known parts of the keystream $\mathbf{z} = (z_0, z_1, \ldots, z_{N-1})$ to try to find the key.

There are different kinds of plain-/cipher text attacks on stream ciphers, where the strongest one is the *key recovery attack*. Given parts of $\mathbf{z}$, the objective is to retrieve the secret key or the secret initialization state, see Section 2. Having the key (or the initialization state), we can generate the unknown parts of $\mathbf{z}$ and decrypt the rest of the message. We can always find the secret key by exhaustive search, that is, we go through all the possible keys and stop when we find a key that produces the correct keystream. However, the key size is usually large enough to make such an attack computationally infeasible. A common technique is to do the attack in several stages, and determine only a few key bits in each stage. This is called the *divide and conquer* technique. The attack is an success if the sum of the runtime complexities for all the stages is smaller than exhaustive search of the whole key space.

A weaker attack is the *distinguishing attack* where the cryptanalyst tries to distinguish $\mathbf{z}$ from a truly random bit stream. Although this is does not break the cipher in the normal sense of the word, it can actually be used to get knowledge about the plain text in certain circumstances. Assume that the eavesdropper knows that one out of n documents is going to be encrypted and sent over an insecure channel. The eavesdropper already knows the content of all the documents, but he does not know which of the documents that has been sent. The eavesdropper can now xor the documents one by one with the cipher text, and do distinguishing tests on the resulting bit streams. When the correct document is chosen, the resulting bitstream will be the keystream and the test will be positive. Thus, the eavesdropper now know which of the documents that was sent.

To evaluate an attack properly, there are 4 main properties we need to consider.

– The runtime complexity.
– The computer memory complexity.
– The number of keystream bits complexity.
– The probability for success

Some attacks may have low runtime complexity, but at the same time they may use a vast amount of memory or keystream bits and have a low probability for success. Therefore, when we compare different attacks we must compare all the properties.

1.3 Overview Over the Thesis

In addition to the introduction paper, the thesis consists of five independent papers, where four of them have been accepted by refereed publications and conferences. The fifth paper is unpublished work which shows the status of our most recent research. All the papers are about cryptanalysis, and we mainly attack stream ciphers based on *linear feedback shift registers*. However, the two last papers papers are cryptanalysis of sequences generated by a recently proposed T-function generator.

Introduction. In this paper, we give a basic introduction to design and cryptanalysis of LFSR based stream ciphers. We also give a brief introduction to the new T-function generator. We stress that we mostly present the techniques which the research in this thesis is based on, and it is not meant to be a complete survey over all the ciphers and attacks that have been proposed.

Improved Fast Correlation Attack Using Low Rate Codes[21]. In this paper, we present a faster method for calculating the metrics in some of the fast correlation attacks. Instead of using relatively few strong parity check equations, we present an algorithm that has low run time complexity by using many weak parity check equations. We show both theoretically and by simulations that the algorithm is efficient even when using billions of equations.

Improved Linear Consistency Attack on Irregularly Clocked Keystream Generators[18]. In this paper, we present an improved linear consistency test (LCT) attack on ciphers where one shift register is clocked by another. The basic idea behind the attack is very simple but it has a very high run time complexity. However, we present an algorithm the makes this attack very efficient compared to previous LCT attacks.

An Improved Correlation Attack Against Irregularly Clocked and Filtered Keystream Generators[19]. This paper is an extension of the linear consistency attack, where we attack a cipher where the LFSR stream is filtered through a Boolean function in addition to the irregularly clocking. We show that the Boolean function is less secure than previously thought.

Linear Properties in the Klimov-Shamir T-function[20]. Recently Klimov and Shamir presented a T-function for generating maximum length sequences. In this paper, we prove that there exist linear properties in the sequences, and that the properties can be used for cryptanalysis of the Klimov-Shamir T-functions.

Algebraic Structures over the Binaries for the Klimov-Shamir T-function. This paper describes the status of our most recent and unpublished work. We show how to generate the Boolean functions from each bit in the initialization state to each bit in the sequence generated by the T-function. Using these functions, we show several interesting properties of the T-function sequences. However, the properties have not yet been proved.

2 The General Stream Cipher Model

The objective of stream ciphers is to expand a short key into a long and pseudo random keystream which emulates the one time pad. A stream cipher consists mainly of a *keystream generator* that generates the stream, and a *mixing function*, usually the simple xor function, which mixes the keystream with the plaintext.

Let the secret key of the cipher be $\mathbf{k}$. Using the denotation from [16], the keystream generator consists of a *finite state machine,* which has an internal state σ_t that is updated by a *next-state* function f. Next, the extraction function $z_t = g(\sigma_t, \mathbf{k})$ takes the internal state σ_t at time t as input, and outputs the keystream bit z_t. The finite state machine is initialized with σ_0, which is extracted from the key $\mathbf{k}$. Since a finite state machine has a limited number of different internal states, it will always repeat itself after a limited number of iterations. This number is the *period* of the cipher, and is the same as the maximum length of the keystream. The keystream $\mathbf{z}$ is mixed with the plaintext $\mathbf{m}$ in the function $c_t = h(m_t, z_t)$. Most stream ciphers are binary additive. See for example Figure 2, where the keystream generator produces a binary keystream $\mathbf{z}$, which is simply xored on plain text. The encryption is more precisely given by $c_t = h(m_t, z_t) = z_t \oplus m_t$.

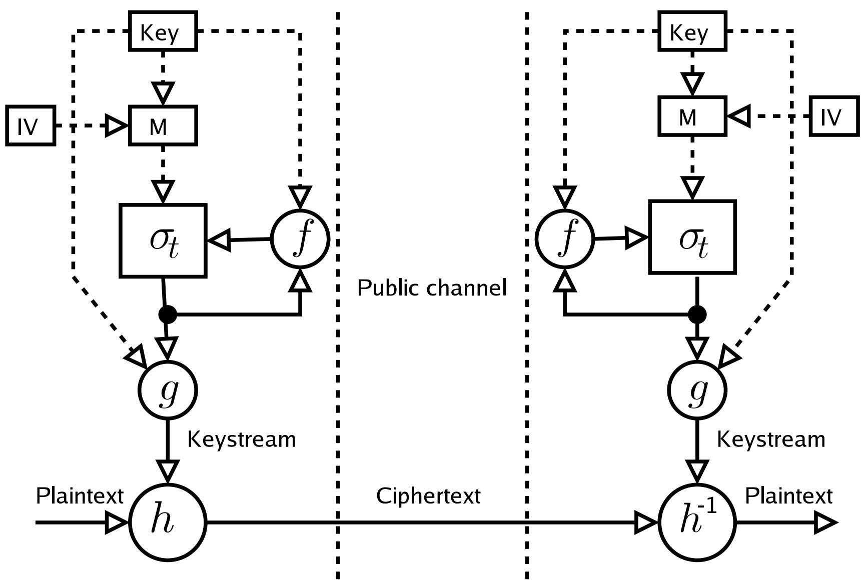

Fig. 1. The general synchronous stream cipher model

In the literature, stream ciphers are often divided in two main types, namely *self- synchronizing* stream ciphers and *synchronous* stream ciphers. The model above is the synchronous stream cipher, which means that both sender and receiver must be synchronized to be able to encrypt and decrypt properly. This can be a problem when sending over a channel where bits may be deleted. If only one bit is deleted, the positions of the following bits are changed, and the rest of the message will be corrupted during the decryption. The result is that the whole message has to be resent. One way to get around this is to send the message in relatively small packages, and re-synchronize for every new package. On the other hand, a self- synchronizing cipher takes the message as input to the keystream generator, and uses the message text to synchronize itself, see [16] for details. However, in this thesis we focus on the synchronous stream ciphers.

Another problem is (as with the one time pad) that a keystream can only be used once. Therefore, we need a new key for every package, which leads to problems with the key distribution. This problem can be fixed by mixing the secret key with an *initialization vector* **IV** that is changed for every new message. The mix $\sigma_0 = M(\mathbf{k}, \mathbf{IV})$ between **k** and **IV** can then be used as an initialization state for the cipher. The **IV** can be public or secret, and it changes following a public known procedure that can be as simple as $\mathbf{IV} \leftarrow \mathbf{IV} + 1$. Since **IV** may be public, it should not be possible to get any information about σ_0 given **IV** only, and the mixing function $M(\mathbf{k}, \mathbf{IV})$ must be very secure. Often the cipher itself, or a smaller version of it, is involved in the mixing function. The function can also work as a 'key expander', which enable us to have bigger internal state than

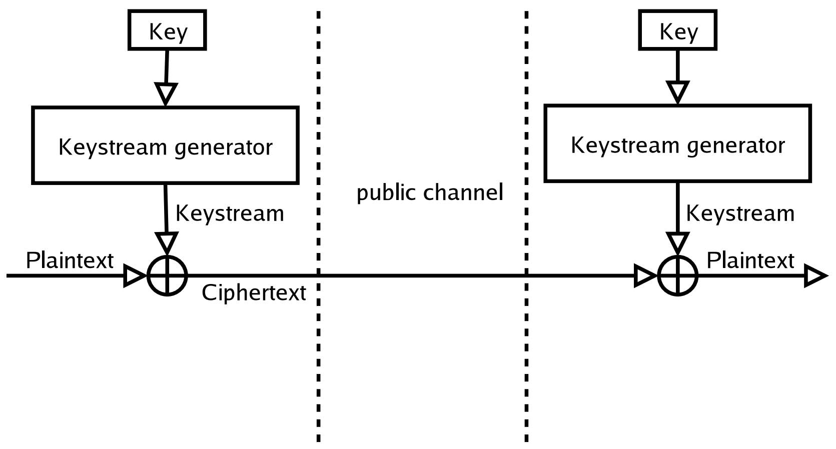

Fig. 2. The synchronous additive stream cipher using $c_t = h(m_t, z_t) = m_t \oplus z_t$ as mixing function

key length. Having a big internal state is in most cases more secure, but requires more computer power.

To summarize, the general synchronous stream cipher is formally described by

$$\begin{aligned}\sigma_0 &= M(\mathbf{IV}, \mathbf{k})\\ \sigma_t &= f(\sigma_{t-1}, \mathbf{k})\\ z_t &= g(\sigma_t, \mathbf{k})\\ c_t &= h(m_t, z_t)\end{aligned}$$

where decryption is done by $m_t = h^{-1}(c_t, z_t)$, as shown in Figure 1.

3 Shift Register Based Stream Ciphers

3.1 Basic Properties of Linear Feedback Shift Registers

A very important requirement for stream ciphers is that its period is as large as possible. *Linear feedback shift registers* (LFSRs) can easily be designed to produce maximum length streams, with good statistical properties. They are also simple to implement and fast in hardware. Due to this, the LFSRs have been popular building blocks for stream ciphers. In this thesis we concentrate on bit based LFSRs, although using larger alphabets is possible. LFSRs with larger alphabet (also called word based LFSRs) produce more bits per iteration, and are therefore much faster than the bit based LFSRs. Since speed is a crucial factor, recent stream cipher designs tend to have word based LFSR, see for example [4,22].

A shift register contains l memory elements, each capable of storing q bits. The elements are ordered in a table, and at each time unit the shift register is

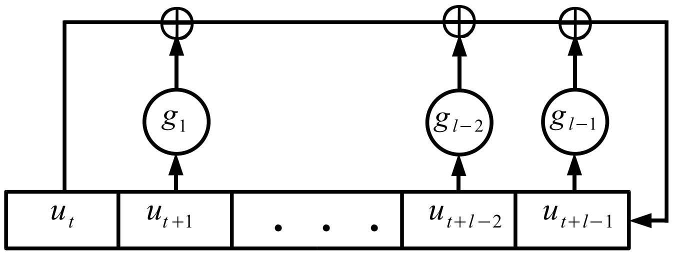

Fig. 3. The linear feedback shift register over the binaries

capable of shifting all the elements one step to the left. Before the shift, a subset of the elements is summed and the sum is fed in to the shift register from the right. The leftmost element before the shift is done is taken as the next element in the sequence generated by the shift register.

There are several mathematical models to describe a binary linear feedback shift register. The most intuitive one is the linear recursion model where $\mathbf{u} = (u_0, u_1, \dots)$ is the bit sequence generated by the linear recursion

$$u_{t+l} = u_t + g_1 u_{t+1} + \cdots + g_{l-1} u_{t+l-1} = u_t + \sum_{i=1}^{l-1} g_i u_{t+i} \bmod 2, \tag{3}$$

where u_t and g_i are in F_2, and F_2 is the finite field with two elements $\{0,1\}$. We see that u_{t+l} is a sum of a subset of $(u_t, u_{t+1}, \dots, u_{t+l-1})$, where the subset is given by the constants $(g_0 = 1, g_1, g_2, \dots, g_{l-1})$. The positions u_{t+i} where $g_i = 1$ are also called the *tapping positions* of the LFSR. The bits $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, u_2, \dots, u_{l-1})$ are the initialization bits or the *initialization state* for the shift register.

We can also denote the LFSR as a generator polynomial,

$$g(x) = 1 + g_1 x + g_2 x^2 + \cdots + g_{l-1} x^{l-1} + x^l = \sum_{i=0}^{l-1} g_i x^i \text{ in } F_2[x], \tag{4}$$

where $g_0 = g_l = 1$ and $F_2[x]$ means all polynomials of x with coefficients in F_2. Using this definition, we can define α by setting $g(\alpha) = 0$, from which we get the reduction rule $\alpha^l = g_{l-1}\alpha^{l-1} + g_{l-2}\alpha^{l-2} + \cdots + g_1\alpha + 1$. By multiplying both sides by α^t we get that

$$\alpha^{t+l} = g_{l-1}\alpha^{t+l-1} + g_{l-2}\alpha^{t+l-2} + \cdots + g_1\alpha^{t+1} + \alpha^t. \tag{5}$$

Equation (5) clearly shows the connection between the linear recursion model (3) and the polynomial model (4).

If $r = 2^l$ is the lowest r that gives $\alpha^r = \alpha$, then the elements in the set $\alpha^1, \dots, \alpha^{2^l-1}$ are all distinct, the element α is a *primitive element* and $g(x)$ is called a primitive polynomial. It follows that the shift register will produce a sequence of maximum length $2^l - 1$ if and only if α is a primitive element. Using the reduction rule (5), we can write all α^{t+l}, $t > 0$, as sums of subsets of the elements $1, \alpha, \alpha^1, \dots, \alpha^{l-1}$. Let $\mathbf{h}_t^{\mathrm{T}} = (h_{0,t}, h_{1,t}, \dots, h_{l-1,t})$ be the coefficients of α^t given by $\alpha^t = h_{0,t} + h_{1,t}\alpha + \cdots + h_{l-1,t}\alpha^{l-1}$. We regard $(\mathbf{h}_0, \mathbf{h}_1, \dots)$ as bit columns. Let I_l be the identity matrix of size $l \times l$. We can now use α to construct an $l \times N$ generator matrix $G = [\alpha^0, \alpha^1, \dots, \alpha^{N-1}]$ for a stream of length N:

$$\begin{aligned} G &= [\mathbf{I}_l \mid \mathbf{h}_l, \mathbf{h}_{l+1}, \mathbf{h}_{l+2}, \dots, \mathbf{h}_{N-1}] \\ &= \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & h_{0,l} & h_{0,l+1} & \cdots & h_{0,N-1} \\ 0 & 1 & \ddots & 0 & 0 & h_{1,l} & h_{0,l+1} & \cdots & h_{1,N-1} \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ddots & 1 & 0 & h_{l-2,l} & h_{l-2,l+1} & \cdots & h_{l-2,N-1} \\ 0 & 0 & \cdots & 0 & 1 & h_{l-1,l} & h_{l-1,l+1} & \cdots & h_{l-1,N-1} \end{bmatrix}. \end{aligned} \tag{6}$$

The stream of length N is now generated by

$$\mathbf{u} = \mathbf{u}^{\mathrm{I}} G = [u_0, u_1, \dots, u_{l-1}, \mathbf{u}^{\mathrm{I}} \cdot \mathbf{h}_l\,,\ \mathbf{u}^{\mathrm{I}} \cdot \mathbf{h}_{l+1}\,, \dots,\ \mathbf{u}^{\mathrm{I}} \cdot \mathbf{h}_{N-1}].$$

In Section 2.1 of our paper "Improved Fast Correlation Attack Using Low Rate Codes" [21] in this thesis, we show an example of a small generator matrix.

3.2 Linear Properties and Weaknesses in LFSRs

Using the equation $\mathbf{u} = \mathbf{u}^{\mathrm{I}} G$, we can set up a set of linear equations that can be used to calculate $\mathbf{u}^{\mathrm{I}}$ given only l bits from the LFSR stream. By multiplying $\mathbf{u}^{\mathrm{I}}$ by G we get the following linear equation set for $0 \le t_0 < t_1 < \cdots < t_{l-1} < N$:

$$\begin{aligned} u_{t_0} &= h_{0,t_0} u_0 + h_{1,t_0} u_1 + \cdots + h_{l-1,t_0} u_{l-1} \\ u_{t_1} &= h_{0,t_1} u_0 + h_{1,t_1} u_1 + \cdots + h_{l-1,t_1} u_{l-1} \\ &\vdots \\ u_{t_{l-1}} &= h_{0,t_{l-1}} u_0 + h_{1,t_{l-1}} u_1 + \cdots + h_{l-1,t_{l-1}} u_{l-1} \end{aligned} \tag{7}$$

Assume we know the bits $u_{t_0}, u_{t_1}, \dots, u_{t_{l-1}}$. If the equations in the set are linearly independent, the set is easily solved using the Gaussian reduction algorithm which has runtime complexity $O(l^3)$. If the set is not linearly independent, we can exchange some bits, put in the new equations, and try again until the equation set is solved.

Even if the generator polynomial $g(x)$ is not known, we can find the initialization state using the Berlekamp Massey algorithm [13,2]. The algorithm needs $2l$ consecutive bits as input and it outputs $g(x)$. Knowing $g(x)$ we can calculate $\mathbf{u}^{\mathrm{I}}$ using the method above. However, since the bits are consecutive, we can also simply use $g(x)$ and Equation (3) to generate back to the initialization state by reversing the recursion (3) to $u_t = u_{t+l} + g_{l-1}u_{t+l-1} + \cdots + g_1 u_{t+1} \bmod 2$.

This shows one of the main problems of using shift registers in stream ciphers. In Section 3.3 we show some techniques that are often used to destroy the linearity properties in the output of shift registers.

The weight of the generator polynomial $g(x)$ is crucial for the security in many stream ciphers, since many different types of attacks on LFSR based ciphers can be done if $g(x)$ has low weight, see Section 4.5. However, if we can find a multiple of $g(x)$ that has low weight, the attacks may still be possible. Assume there exists a function $a(x)$ that for some $b(x)$ gives

$$a(x) = g(x) \cdot b(x) = 1 + x^{i_1} + x^{i_2} + \cdots + x^{i_{w-1}} \text{ in } F_2[x], \tag{8}$$

where w is the *weight* of the equation. This corresponds to the linear equation

$$u_t + u_{t+i_1} + u_{t+i_2} + \cdots + u_{t+i_{w-1}} = 0 \bmod 2$$

that will hold over all bit streams generated by $g(x)$. Since we have that

$$\begin{aligned} & u_t + u_{t+i_1} + u_{t+i_2} + \cdots + u_{t+i_{w-1}} \\ &= \mathbf{u}^{\mathrm{I}}\mathbf{h}_t + \mathbf{u}^{\mathrm{I}}\mathbf{h}_{t+i_1} + \cdots + \mathbf{u}^{\mathrm{I}}\mathbf{h}_{t+i_{w-1}} \\ &= \mathbf{u}^{\mathrm{I}}(\mathbf{h}_t + \mathbf{h}_{t+i_1} + \cdots + \mathbf{h}_{t+i_{w-1}}), \end{aligned} \tag{9}$$

we will find this low weight multiple by searching for w columns, $\mathbf{h}_t, \mathbf{h}_{t+i_1}, \mathbf{h}_{t+i_2}, \ldots, \mathbf{h}_{t+i_{w-1}}$, in G that xor to the all zero column $(0, 0, \ldots, 0)^{\mathrm{T}}$ for a chosen t. The equation is cyclic and will hold for all t since Equation (9) will sum to zero for all $\mathbf{u}^{\mathrm{I}}$ when $\mathbf{h}_t + \mathbf{h}_{t+i_1} + \cdots + \mathbf{h}_{t+i_{w-1}} = 0$. For simplicity we often set $t = 0$, which means that it is sufficient to find the $w-1$ columns $\mathbf{h}_{i_1}, \mathbf{h}_{i_2}, \ldots, \mathbf{h}_{i_{w-1}}$ that gives

$$\mathbf{h}_{i_1} + \mathbf{h}_{i_2} + \cdots + \mathbf{h}_{i_{w-1}} = \mathbf{h}_0 = (1, 0, \ldots, 0)$$

3.3 Fixing the Linearity Property Problem in LFSRs

Due to the linear properties in the LFSRs, we must combine them with other cryptographic primitives to be able to design a secure keystream generator. The goal is to remove the linear properties in the bit streams generated by the LFSRs, so that the streams satisfy all the requirements in Section 1.1. There are many methods for removing the linear properties and in this thesis we attack ciphers that use *Boolean functions* and *irregularly clocking.*

Boolean functions. In cryptography, a Boolean function is defined as a function that takes n bits as input variables and outputs one bit. A simple but well known function is the *Geffe* function [6]

$$f(x_1, x_2, x_3) = x_1 + x_1x_2 + x_2x_3 \bmod 2.$$

Let $f(\mathbf{x})$ be a Boolean function with $\mathbf{x} = x_1, x_2, \ldots, x_n$ as input variables. A product of r distinct variables is called a *r'th order product* of the variables. All Boolean functions can be written as sums of such products mod 2. When the products are arranged in a specific order, the function is in the *Algebraic Normal Form* (ANF) given by

$$f(\mathbf{x}) = a_0 + a_1x_1 + a_2x_2 + \cdots + a_nx_n + a_{n+1}x_1x_2 \\ +a_{n+2}x_1x_3 + \cdots + a_{2^n-1}x_1x_2\cdots x_n \bmod 2.$$

The ANF denotation can be represented by the table $T_{\text{anf}} = (a_0, a_1, \ldots, a_{2^n-1})$. The *algebraic degree,* also called the *nonlinear order* of the function, is the maximum order of the terms in the function. For the simple Geffe function, the ANF table is $T_{\text{anf}} = (0, 1, 0, 0, 1, 0, 1, 0)$, and it has algebraic degree 2.

Boolean functions can also be defined by a *truth table* $T_{\text{truth}} = (b_0, b_1, \ldots, b_{2^n-1})$, where b_i is in F_2. The truth table is a table of the output of $f(\mathbf{x})$ given the input $\mathbf{x}$. Given a truth table for $f(\mathbf{x})$, the output is calculated as

$$f(\mathbf{x}) = T_{\text{truth}}(x_1 + x_22 + x_32^2 + \cdots + x_n2^{n-1}).$$

The truth table for the Geffe function is $T_{\text{truth}} = (0, 0, 0, 1, 1, 1, 0, 1)$.

A Boolean function $f(\mathbf{x})$ is balanced if $\Pr(f(\mathbf{x}) = 0) = \Pr(f(\mathbf{x}) = 1) = 0.5$. Further on it is *correlation immune* (CI) if $\Pr(x_i = f(x_1, x_2, \ldots, x_n)) = 0.5$ for all i, $1 \le i \le n$. That is, the Boolean function is correlation immune if there is no correlation between $f(\mathbf{x})$ and any of the input bits. The Geffe generator is not correlation immune since $\Pr(x_3 = f(x_1, x_2, x_3)) = \Pr(x_1 = f(x_1, x_2, x_3)) = 0.75$. A function is *resilient* if it is both correlation immune and balanced.

A Boolean function is known to be m'th order correlation immune if there does not exist any linear combination of $k \le m$ input bits that have a correlation with the output of the function. That is, $\Pr(x_{j_1} + x_{j_2} + \cdots + x_{j_k} = f(x_1, x_2, \ldots, x_n)) = 0.5$ for $1 \le k \le m$ and $1 < j_1, j_2, \ldots, j_k \le n$. If the function is balanced and has m'th order correlation immunity, then it is $m-$resilient.

In the stream cipher literature, there are two main methods for combining the Boolean function with LFSRs. The first method is to filter only one shift register by taking n bits from positions in the internal state of the LFSR given by $j_1, j_2, \ldots, j_n$ at time t as input to the Boolean function. More precisely, the keystream is generated by

$$z_t = f(u_{t+j_1}, u_{t+j_2}, \ldots, u_{t+j_n}).$$

The second method is to use n LFSRs, and take the output bits from each LFSR at time t as input to the Boolean function. Each LFSR$_i$ generates a bit stream $\mathbf{u}^i = (u_0^i, u_1^i, \ldots)$ and the keystream is generated by

$$z_t = f(u_t^1, u_t^2, \ldots, u_t^n).$$

Figure 4 is a graphical representation of the two methods.

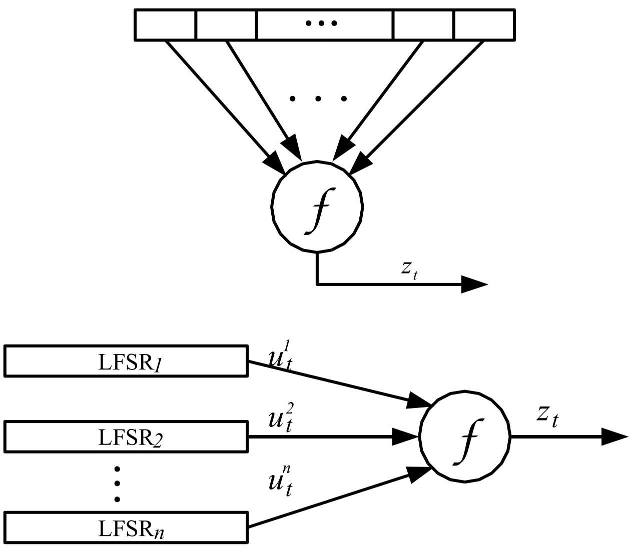

Fig. 4. The methods for combining LFSRs with a Boolean function f with n inputs

Irregularly clocked LFSRs. Instead of filtering the LFSR stream through a Boolean function, we can destroy the linear properties in the bit stream by deleting or copying bits in an irregular way. This is called irregularly clocking. Often this is done by letting a LFSR$_\mathrm{s}$ produce a bit stream $\mathbf{s}$ that by some method is turned into a *clock control sequence* of integers, $\mathbf{c} = (c_0, c_1, \dots, c_{N-1})$ where $0 \leq c_t$, which is then used to clock a LFSR$_\mathrm{u}$.

Let t be the clock of the whole generator, and let v be the clock of $LFSR_\mathrm{s}$ of length l_s. Let $(s_v, s_{v+1}, \dots, s_{v+l_\mathrm{s}-1})$ be the inner state of LFSR$_\mathrm{s}$ at time v. Then we get the sequence $\mathbf{c}$ by

$$c_t = D(s_v, s_{v+1}, \dots, s_{v+l_\mathrm{s}-1})$$

for some function D that takes the inner state of LFSR$_\mathrm{s}$ as input and outputs an integer ≥ 0. The generator is often synchronized with LFSR$_\mathrm{s}$, such that $v = t$. Let the shift register LFSR$_\mathrm{u}$ produce a bit stream $\mathbf{u}$ =$(u_0, u_1, \dots)$. Using the clock control sequence we let the output of the generator be

$$z_t = u_{k(t)}$$

where $k(t) = \sum_{i=0}^{t} c_i$. Thus, LFSR$_\mathrm{s}$ controls the number of times that LFSR$_\mathrm{u}$ is clocked before the output is taken as keystream bit.

A typical example of the D function is the *step-1/step-2* function where $D(s_t) = 1 + s_t$. Another example is the well known *shrinking generator* where the output u_k of LFSR$_\mathrm{u}$ is discarded if the output of LFSR$_\mathrm{s}$ is $s_k = 0$ and taken as output z_t if $s_k = 1$. Using our model this means that the clock sequence is given by $D(s_{k(t-1)}, s_{k(t-1)+1}, \ldots, s_{k(t-1)+l_s-1}) = y$ where the $y-1$ is the number of consecutive zeros from the entry $s_{k(t-1)}$ in the stream $\mathbf{s}$, and $k(-1) = 0$. We see that in this special case $\mathrm{LFSR_s}$ and $\mathrm{LFSR_u}$ are synchronized, that is $v = k(t)$.

4 Cryptanalysis of Stream Ciphers Based on LFSRs

There are many different types of attacks on stream ciphers based on LFSRs. To give some theoretical background, we will in this section go through some of the attacks and cryptanalysis that are relevant to the attacks we present in this thesis.

4.1 Using Coding Theory in Cryptanalysis

Many attacks on stream ciphers use coding theory to determine the properties such as run time complexity and the probability for success. Here we give a basic introduction to the area.

The binary symmetric channel. Let $\mathbf{u} = (u_0, u_1, \ldots, u_t, \ldots)$ be a sequence that is sent through a channel C, and let $\mathbf{z} = (z_0, z_1, \ldots, z_t, \ldots)$ be the corresponding sequence received on the other end of the channel. When bits are sent through a *Binary Symmetric Channel* (BSC), there is a probability p for that an error occur so that a transmitted 0 will be received as 1 or the other way around, that is $p = \Pr(u_t \neq z_t)$. Hence, the BSC is a noisy channel where p is the *crossover probability*, see Fig 5. The crossover probability is also often denoted

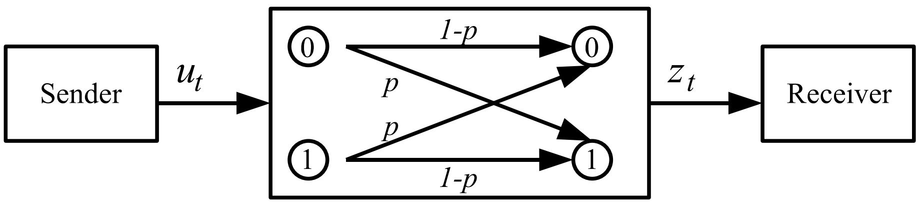

Fig. 5. The binary symmetric channel model

as $p = 0.5 \pm \epsilon$, where ϵ is called the *bias*. When $\epsilon > 0$ we say that there is a *correlation* $1-p$ between u_t and z_t.

Basic coding theory. Coding theory deals with the problem of how to reliably transfer information over a noisy channel. Assume we have a sender that needs to transmit some bits. Since the channel is noisy, errors may occur and the receiver needs to correct those errors. Thus, we need an *error correcting code.* In an $[n, k]$

block code $\mathcal{C}$, the bits we want to transmit are divided into blocks of k bits and encoded into blocks of $n > k$ bits. This constellation is called a codeword, and the code is defined by the set of 2^k codewords of length n. Thus, each codeword contains k bits of information (the actual information we want to transmit), and $n - k$ *redundancy bits*, which are used for error detection and correction. Each of the 2^k possible blocks of information corresponds to a codeword in $\mathcal{C}$. For example a very simple $[3, 1]$ code $\mathcal{C}$ can be defined by the two codewords $(0, 0, 0)$, $(1, 1, 1)$, which represent the bits 0 and 1. Since each codeword contains only k bits of information, the actual transmission rate is $R = \frac{k}{n}$ bits of information per transmitted bit. The code above has the transmission rate $R = 1/3$. Hence, the transmission is slowed down, but we are able to correct the errors that may occur as long as the number of errors is not too large.

The *Hamming distance* $H(\mathbf{x}, \mathbf{y})$ is defined as the number of bits in the vectors $\mathbf{x}$ and $\mathbf{y}$ that differ, for example $H(\,(1, 1, 1)\,, (0, 0, 0)\,) = 3$. Let d be the minimum Hamming distance between all codewords in a code. Let $\mathbf{u}$ be the transmitted codeword, and $\mathbf{z}$ be the received codeword. Using a maximum likelihood decoding algorithm, the receiver decodes the received codeword $\mathbf{z}$ to the codeword $\hat{\mathbf{u}}$ in $\mathcal{C}$ that has the lowest Hamming distance to $\mathbf{z}$. For example in the code above, the received codeword $(0, 0, 1)$ will be decoded to $(0, 0, 0)$ and not $(1, 1, 1)$, and we assume that a 0 was transmitted. We see that the code above is able to correct 1 error, with transmission rate $R = 1/3$. If more than $\lfloor d/2 \rfloor$ errors occur, the received codeword will be closer to a wrong codeword in C. Hence, to prevent that a received codeword is decoded wrongly and at the same time having fast communication, the objective of a good code is to have as great minimum Hamming distance d as possible with as high transmission rate as possible.

The *channel capacity* $C(p)$ of the BSC is a measurement for the maximum data rate that can be transmitted over the channel without errors. When the channel capacity is 0, it is not possible to send information over the channel at all. When the channel capacity is 1 no errors will occur, and we do not need coding. The channel capacity for the BSC is given by

$$C(p) = 1 + p \log_2 p + (1 - p) \log_2(1 - p), \tag{10}$$

where p is the crossover probability in the channel. We have that $C(0.5) = 0$, and $C(0) = C(1) = 1$, so the worst possible channel has $p = 0.5$. Let $P_{\mathrm{e}}(p)$ be the probability that a received codeword is not decoded to the correct codeword, that is $P_{\mathrm{e}}(p) = \Pr(\mathbf{u} \neq \hat{\mathbf{u}})$. Recall that $R = k/n$. If the information that is transmitted over the channel is random the *Shannon's Noisy Theorem* [23] tells us that for

$$R < C(p) \tag{11}$$

and for codeword length n large enough, there exist a code for any $P_{\mathrm{e}}(p) > 0$. Thus, as long as the code rate is lower than the channel capacity, it is possible to construct an error correction code where $P_{\mathrm{e}}(p)$ approaches zero when n (and k)

grows large. However, the theorem does not tell us how to construct such codes. In cryptography, this upper bound for the transmission rate is used to evaluate the success rates and runtime complexities of many types of attacks.

4.2 From Coding to Cryptanalysis

To be able to attack ciphers using coding theory, we must identify a coding system in the cipher that gives low noise, and a fast decoding algorithm.This is usually not easy, since the designers try (or at least should try) to design ciphers in such a way that the coding model can not be used. Attacks that use the coding theory model are called *correlation attacks*, since there is a correlation between the bits that are send through the channel and the bits that are received at the other end. The closer p is to 0.5, the harder it is to attack the cipher. Therefore, the strength of these attacks are most often evaluated from how low complexity the attacks have when p is as close to 0.5 as possible.

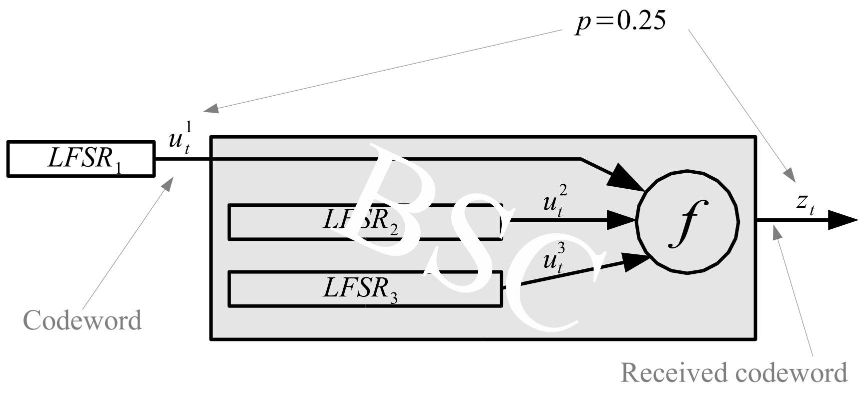

Fig. 6. The simple cipher viewed as a binary symmetric channel

4.3 Siegenthaler's Correlation Attack

The first and most simple correlation attack on stream ciphers was presented by Siegenthaler in [24]. The attack models most of the cipher as a black box, and is therefore quite general and works against many different kinds of ciphers. We will show how to attack the second cipher design in Figure 4 with the Geffe function as the Boolean function. The main point is that the attack is divided in to several stages, and we only have to do exhaustive search for the initialization state for one of the LFSRs at each stage instead of searching the whole key space.

First we must try to describe the cipher in context of the binary symmetric channel. Assume we have N bits of the keystream $\mathbf{z} = (z_0, z, \ldots, z_{N-1})$, and we want to find the unknown initialization state $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, \ldots, u_{l-1})$ for LFSR$_1$.

The Geffe function has a correlation property $\Pr(x_1 = f(x_1, x_2, x_3)) = 0.75$. Using this, the shift registers LFSR_2, LFSR_3 and the Geffe function are perceived as a binary symmetric channel with crossover probability $p = \Pr(z_t \neq u_t) = 1 - \Pr(x_1 = f(x_1, x_2, x_3)) = 0.25$ as showed in Figure 6. Let the key $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, \ldots, u_{l-1})$ be the information bits in the coding model, and let $\mathbf{u} = \mathbf{u}^{\mathrm{I}} \cdot G = (u_0, u_1, u_2, \ldots, u_{N-1})$ be the corresponding codeword that is generated by LFSR_1. The matrix G is the generator matrix for the code. Following the BSC model we let $\mathbf{z}$ be the received codeword. Thus, this is a $[N, l]$ code with code rate $R = \frac{l}{N}$. The problem is now reduced to a pure decoding problem, and knowing $\mathbf{z}$ we can recover $\mathbf{u}^{\mathrm{I}}$ using decoding techniques.

The attack algorithm is very simple. For all possible values of $\hat{\mathbf{u}^{\mathrm{I}}}$ we generate the $\mathrm{LFSR}_{\mathrm{u}}$ stream $\hat{\mathbf{u}}$ and test how many of the bits in $\mathbf{z}$ of length N that satisfy

$$z_t = \hat{u}_t. \tag{12}$$

The number of equal bits is called the *metric* for the guess $\hat{\mathbf{u}^{\mathrm{I}}}$. This is equivalent to letting the metric be number of equations that are satisfied in the following set of *parity check* equations, which is a simple combination of Equation (12) and the Equation set (7):

$$\begin{aligned}
z_0 &\approx \hat{u}_0 \\
&\vdots \\
z_{l-1} &\approx \hat{u}_{l-1} \\
z_l &\approx h_{0,l}\hat{u}_0 + h_{1,l}\hat{u}_1 + \cdots + h_{l-1,l}\hat{u}_{l-1} \\
&\vdots \\
z_{N-1} &\approx h_{0,N-1}\hat{u}_0 + h_{1,N-1}\hat{u}_1 + \cdots + h_{l-1,N-1}\hat{u}_{l-1}
\end{aligned} \tag{13}$$

We use '$\approx$' to denote that when $(\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{l-1})$ is the correct guess for initialization state, the equations hold with probability $1 - p$. In that case the metric will be approximately $(1 - p)N > \frac{N}{2}$. When the guess $\hat{\mathbf{u}}$ is wrong there will be no correlation between $\hat{\mathbf{u}}$ and $\mathbf{z}$, and the metric will be approximately $\frac{N}{2}$. Thus, after we have gone through all the 2^l possible values for $\hat{\mathbf{u}^{\mathrm{I}}}$, the guess with the highest metric is, with some probability, the correct initialization state $\mathbf{u}^{\mathrm{I}}$. Hence, the algorithm simply finds the $\hat{\mathbf{u}^{\mathrm{I}}}$ that produces the stream $\hat{\mathbf{u}}$ which is most equal to $\mathbf{z}$. This is according to the maximum likelihood decoding method, where we let $\mathbf{u}$ be the guess $\hat{\mathbf{u}}$ that minimizes the Hamming distance $H(\hat{\mathbf{u}}, \mathbf{z})$. The attack is repeated for all the rest of the shift registers.

Evaluation of Siegenthaler's attack. Since we have to make 2^{l_i} guesses for each LFSR_i and for each guess we must go through and check N_i bits to attack LFSR_i, the total total runtime complexity for the attack is $O\left(\sum_{i=1}^{n} 2^{l_i} N_i\right)$. This complexity is in most cases much lower than the total exhaustive search which has runtime complexity $O\left(\prod_{i=1}^{n} 2^{l_i}\right)$. Hence, this is a classical example of the divide and conquer technique, since we can divide the attack into several stages, where we in each stage only attack a small part of the cipher.

To find out how many bits we need for the attack to succeed we use coding theory. Using the Formula (11) from the Shannon's Noisy Theorem on the $[N, l]$ code as explained in Section 4.1, the attack will succeed with a certain probability $P_e(p) > 0$ if $N > \frac{l}{C(p)}$. Experimentation shows that the success rate for the attack will be approximately $P_e(p) = 0.5$ if $N = \frac{l}{C(p)}$, and will be close to 1 if

$$N \geq \frac{2l}{C(p)},$$

see [24].

Hence, the number of keystream bits needed to attack LFSR$_i$ is $N_i = \frac{2l_i}{C(p_i)}$, and the full complexity for the attack is $O\left(\sum_{i=1}^{n} \frac{l_i 2^{l_i}}{C(p_i)}\right)$. The full attack needs $N = \max_{0<i\leq n} \frac{2l_i}{C(p_i)}$ keystream bits and the memory complexity is also N.

For the cipher where the channel has crossover probability $p = 0.25$ and the length of the LFSR is $l = 40$, we need approximately $N = 2 \cdot 40/C(0.25) = 424$ keystream bits to succeed. The runtime for attacking the LFSR will be equivalent to approximately $N2^l = 424 \cdot 2^{40} \approx 2^{48.7}$ bit checks.

4.4 Fast Correlation Attacks

Although the basic correlation attack on ciphers with many LFSRs is much faster than exhaustive search, the runtime complexity is lower bounded by the lengths of the LFSRs. In addition to that, the attack does not work against ciphers where only one LFSR is filtered by a Boolean function, since guessing the LFSR is the same as exhaustive search of the whole key space. The objective of *fast correlation attacks* is to get a runtime complexity lower than the bound 2^l set by the lengths of the LFSRs.

A simple fast correlation attack. A natural extension of Siegenthaler's correlation attack is the simple fast correlation attack by Chepyzhov, Johansson and Smeets[3]. In this attack the divide and conquer technique is taken even further. By doing a simple transformation on the Equation set (13), it is possible to reduce the number of bits we need to guess on in each stage of the attack.

For a chosen B, the transformation is simply to find w equations in the Equation set (13) that when summed mod 2, the coefficients for $(u_{l-B}, u_{l-B+1}, \ldots, u_{l-1})$ in the result are zero. Hence, we do not need to guess on those bits, which speeds up the attack. More specific the transformation is to search for the indexes $i_0, i_1, \ldots, i_w$ in the equation set that give

$$(\mathbf{h}_{i_0} + \mathbf{h}_{i_1} + \cdots + \mathbf{h}_{i_{w-1}}) = (c_0, c_1, \ldots, c_{B-1}, \overbrace{0, \ldots, 0}^{l-B})^{\mathrm{T}},$$

where $\mathbf{h}_i = (h_{0,i}, h_{1,i}, \ldots, h_{l-1,i})^{\mathrm{T}}$ and $c_0, \ldots, c_{B-1}$ are arbitrary bits. Each such sum gives one new equation of the form

$$z_{i_0} + z_{i_1} + \cdots + z_{i_{w-1}} \approx c_0 u_0 + c_1 u_1 + \cdots + c_{B-1} u_{B-1}. \tag{14}$$

This is equivalent to finding w columns in the generator matrix G that xor to zero in the $l - B$ lowest bits. Finding m such equations, we get an equation set of the following form

$$\begin{aligned} z_{i_{0,0}} + \cdots + z_{i_{0,w-1}} &\approx c_{0,0}u_0 + c_{0,1}u_1 + \cdots + c_{0,B-1}u_{B-1} \\ z_{i_{1,0}} + \cdots + z_{i_{1,w-1}} &\approx c_{1,0}u_0 + c_{1,1}u_1 + \cdots + c_{1,B-1}u_{B-1} \\ &\vdots \\ z_{i_{m-1,0}} + \cdots + z_{i_{m-1,w-1}} &\approx c_{m-1,0}u_0 + c_{m-1,1}u_1 + \cdots + c_{m-1,B-1}u_{B-1} \end{aligned} \tag{15}$$

Let $y_j = z_{i_{j,0}} + z_{i_{j,1}} + \cdots + z_{i_{j,w-1}}$. We see that the Equation (13) set is reduced to

$$\begin{aligned} y_0 &\approx c_{0,0}u_0 + c_{0,1}u_1 + \cdots + c_{0,B-1}u_{B-1} \\ y_1 &\approx c_{1,0}u_0 + c_{1,1}u_1 + \cdots + c_{1,B-1}u_{B-1} \\ &\vdots \\ y_{m-1} &\approx c_{m-1,0}u_0 + c_{m-1,1}u_1 + \cdots + c_{m-1,B-1}u_{B-1}. \end{aligned} \tag{16}$$

From this point on, using $\mathbf{y}$ instead of $\mathbf{z}$ as input to the decoding algorithm, the attack is similar to Siegenthaler's attack. We test the equation set (16) on all the 2^B possible values for $\mathbf{s} = u_0, u_1, \ldots, u_{B-1}$. When this is done, we choose $u_0, \ldots, u_{B-1}$ to be the guess that satisfy the most equations.

Evaluation of the simple fast correlation attack. Since the fast correlation attack above can be looked upon as an extension of Siegenthaler's attack, it can be evaluated in a similar way. The main difference is that the channel is much worse, since the crossover probability must be calculated differently by using the Piling up lemma [14]. Let P_w be the channel noise, when w keystream bits are summed to get parity check equations. The lemma deals with the probability $\Pr(a_0 + a_1 + \cdots + a_{w-1} = b_0 + b_1 + \cdots + b_{w-1})$ for some correlated variables a_i and b_i. A simplified version of the lemma is given by

$$P_w = \frac{1}{2} - 2^{w-1}\left|\frac{1}{2} - p\right|^w,$$

where $P_w = \Pr(a_0 + a_1 + \cdots + a_{w-1} \neq b_0 + b_1 + \cdots + b_{w-1})$ when $p = \Pr(a_i \neq b_i)$. Thus, $1 - P_w$ is the probability that the parity check equation $y_t \approx c_{t,0}u_0 + c_{t,1}u_1 + \cdots + c_{t,B-1}u_{B-1}$ holds. Even though this channel has much lower channel capacity we gain in runtime complexity, since the codewords have fewer information bits and are easier to decode.

Using Shannon's noisy theorem and substituting l with B and p with P_w, the number of parity check equations needed for a successful attack with probability close to 1 is now given by

$$m \geq \frac{2B}{C(P_w)}. \tag{17}$$

The runtime complexity is

$$O(2^B m)$$

instead of $O(2^l N)$ for determining some of the initialization state bits, where $m > N$. However, since $2^B \ll 2^l$, this attack is in most cases much faster than Siegenthaler's attack.

In Siegenthaler's attack, the number of keystream bits are the same as the number of parity check equations, since each keystream bit gives one parity check equation. In this attack, the number of parity check equations are dependent how many sums of columns in G we can find in the pre-processing that sum to zero in the $l - B$ lowest bits. This number is given by the the $\binom{N}{w}$ combinations of w columns in G of length N that exist multiplied by the probability $1/2^{l-B}$ for that each of those combinations are zero in the $l - B$ lowest bit, see[10]. Thus, the number of parity check equations we expect to find is

$$E(m) = \frac{\binom{N}{w}}{2^{l-B}}. \tag{18}$$

To have a successful attack, the number of keystream bits N must large enough to satisfy $\frac{\binom{N}{w}}{2^{l-B}} > \frac{2B}{C(P_w)}$. Hence, it follows that the different parameters in the attack much be chosen carefully.

Convolutional attack. The fast correlation attack via convolutional codes[9,10] was actually presented before the attack above. However, mathematically the convolutional attack can be perceived as a further extension of the simple fast correlation attack. Since the Equation set (15) is cyclic we get that

$$\begin{aligned} y_{t,0} &\approx c_{0,0}u_t + c_{0,1}u_{t+1} + \cdots + c_{0,B-1}u_{t+B-1} \\ &\vdots \\ y_{t,m-1} &\approx c_{m-1,0}u_t + c_{m-1,1}u_{t+1} + \cdots + c_{m-1,B-1}u_{t+B-1} \end{aligned} \tag{19}$$

where $y_{t,j} = z_{t+i_{j,0}} + z_{t+i_{j,1}} + \cdots + z_{t+i_{j,w-1}}$. For each t, $0 \leq t < T$ for a chosen T, we let $\mathbf{s}_t = \hat{u}_t, \hat{u}_{t+1}, \ldots, \hat{u}_{t+B-1}$ go through all the 2^B possible state values. For each state we evaluate the Equation set (19) and let $\text{metric}_{\mathbf{s},t}$ be the number of equations that hold for the given state $\mathbf{s}_t$. The number of steps T should be around $10B$ [9]. Now we have a $2^B \times T$ *Trellis*, where each of the 2^B different states at time t has a $\text{metric}_{\mathbf{s},t}$ assigned to it. Using the *Viterbi* algorithm [5,25], we can now find the longest possible path (the path with the highest sum of metrics) trough the trellis, that is, we choose the path $(\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_{T-1})$ that maximizes the sum $\text{metric}_{\mathbf{s}_0,0} + \text{metric}_{\mathbf{s}_1,1} + \cdots + \text{metric}_{\mathbf{s}_{T-1},T-1}$. The path corresponds to the T first bits of the bit stream generated by the LFSR, and the attack is equivalent to the decoding of a $[m, 1, B]$ convolution code.

Evaluation. Let $p_e < T \cdot 2^{-B}$ and $p = \Pr(u_t \neq z_t)$. The attack has success with probability $1 - p_e$ if the inequality

$$p \leq \frac{1}{2} - \frac{1}{2}\left(\frac{4ln2}{m}\right)^{\frac{1}{2w}}$$

holds [10], and the runtime complexity for the attack is

$$O(2^B mT).$$

The attack seems from this formula to be slower than the simple fast correlation attack, but since this attack uses less parity check equations the speed is approximately the same. The downside of this attack is that it uses much memory, in order of $O(2^B T)$, to store the trellis. However, in the paper "Improved fast correlation attack using low rate codes" we show in this thesis that by using some algorithmic techniques and tuning the parameters properly, this attack becomes both memory and runtime efficient.

4.5 Iterative Fast Correlation Attacks

Another class of fast correlation attacks is the iterative attacks. The iteration attacks are generally not based on the block code model as above, and the focus is not on guessing bits. Instead the decoding is done in several rounds or iterations where the keystream is altered/corrected often using low weight parity check equations. Hopefully the bit stream converges to the bit stream produced by the LFSR in the cipher.

The first fast correlation attack. Even though the simple fast correlation attack presented by Chepyzhov, Johansson and Smeets is a natural extension of Siegenthaler's correlation attack, it was far from being the first fast correlation attack. The first fast correlation attack presented was quite different and was presented by Meier and Staffelbach in 1988 in [15]. This attack works when the generator polynomial $g(x)$ for the LFSR has low weight.

Two techniques are used to turn $g(x)$ into several parity check equations. Assume $g(x)$ has weight w, that is $g(x) = 1 + x^{i_1} + \cdots + x^{i_{w-2}} + x^{i_{w-1}}$, where $i_{w-1} = l$. This corresponds to the linear equation $u_t + u_{t+i_1} + u_{t+i_2} + \cdots + u_{t+i_{w-2}} + u_{t+i_{w-1}}$. Since $g(x)^2 = (1 + g_1x + g_2x^2 + \cdots + x^l)^2 = 1 + g_1x^2 + g_2x^4 + \cdots + x^{2l}$ in $F_2[x]$, we get that $u_t + u_{t+2i_1} + \cdots + u_{t+2i_{w-1}} = 0$. Thus, we get equations that hold over the bit stream $\mathbf{u}$ by repeatedly multiplying the indexes $i_1, \dots, i_{w-1}$ by 2. Hence, we can from one equation build up the following equation set

$$\begin{aligned} u_t + u_{t+i_1} + u_{t+i_2} + \cdots + u_{t+i_{w-1}} &= 0 \\ u_t + u_{t+2i_1} + u_{t+2i_2} + \cdots + u_{t+2i_{w-1}} &= 0 \\ u_t + u_{t+4i_1} + u_{t+4i_2} + \cdots + u_{t+4i_{w-1}} &= 0 \\ &\vdots \end{aligned} \tag{20}$$

By displacing the equations we can from each of the equation in (20) get w new parity check equations for the bit u_t, $i_{w-1} < t < N - i_{w-1}$.

$$\begin{aligned} \underline{u_t} + u_{t+i_1} + u_{t+i_2} + \cdots + u_{t+i_{w-1}} &= 0 \\ u_{t-i_1} + \underline{u_t} + u_{t+i_2-i_1} + \cdots + u_{t+i_{w-1}-i_1} &= 0 \\ &\vdots \\ u_{t-i_{w-1}} + u_{t+i_1-i_{w-1}} + u_{t+i_2-i_{w-1}} + \cdots + \underline{u_t} &= 0 \end{aligned}$$

Combining these two techniques the total number of parity check equations we get is approximately $m \approx w \log_l \frac{N}{2}$. Note that all the equations can not be used for all t, since some equations will exceed the end points 0 and N when t is low or high.

The attack is as follows. Assume we have a keystream $\mathbf{z}$ of length N, where $p = \Pr(u_t \neq z_t) \neq 0.5$. By iterative decoding we want to retrieve the LFSR bit stream $\mathbf{u}$ from $\mathbf{z}$.

A threshold p_{thr} is calculated, and each bit in $\mathbf{z}$ is given a probability p_t for being correct. The initial probability for all the bits are $p_t = 1 - p$. Then all the equations in the set that lie within the end points 0 and N are tested on z_t for all t. Each bit z_t is given a new probability p_t for being correct, calculated from the number of equations that hold, and the probabilities for the bits involved in the equations. When this is done, all the bits z_t that have a probabilities lower that the threshold p_{thr} are corrected to $z_t \leftarrow z_t \oplus 1$ and the probabilities for being correct are also recalculated to $p_t \leftarrow 1 - p_t$. Next, a new threshold p_{thr} is calculated and the whole process starts over. For each iteration we hopefully get a bit stream that is closer to $\mathbf{u}$ and the probabilities p_t for being correct get closer to 1. This is done until all the equations in the set hold, which means that we have retrieved the sequence $\mathbf{u}$ from $\mathbf{z}$. We refer to [15] for the mathematical details of this attack.

This attack only works when the generator polynomial has low weight, since when it has high weight, the equations give much less information about each bit. There are many variations of this attack, see among others [7,17], where the methods for calculating p_{thr} and p_t vary. If $g(x)$ has high weight, it is possible, as described in Section 3.2, to find multiples of $g(x)$ that have low weight, and use the multiples in the attack.

4.6 A Linear Consistency Attack on Irregularly Clocked Generators

To test if a bit stream $\mathbf{u}$ is produced by a given LFSR we can use a linear consistency test as proposed by Zeng, Yang and Rao in [26]. The test is to build up an equation set using $\mathbf{u}$ and the generator matrix G for the LFSR, as explained in Section 3.2, and try to solve it using the Gaussian reduction algorithm. If the set is solvable, the stream has passed the test and we assume that the stream was produced by the LFSR.

This can be used to attack the model where a LFSR$_\mathrm{u}$ of length l_u is clocked by a LFSR$_\mathrm{s}$ of length l_s. Assume we have a keystream $\mathbf{z}$ that is generated by the model in Section 3.3. From this sequence we want to determine the initialization states $\mathbf{s}^\mathrm{I}$ for LFSR$_\mathrm{s}$ and $\mathbf{u}^\mathrm{I}$ for LFSR$_\mathrm{u}$.

The attack is as follows. We make a guess $\hat{\mathbf{s}}^\mathrm{I}$ for the initialization state $\mathbf{s}^\mathrm{I}$ for LFSR$_\mathrm{s}$ and generate the corresponding clock control sequence $\hat{c}_t$. Using $\hat{c}_t$ we calculate $\hat{k}(t) = \sum_{i=0}^{t} \hat{c}_t$ for all t, $0 < t < l_\mathrm{u}$. If the guess $\hat{\mathbf{s}}^\mathrm{I}$ is correct we now know where the bits z_t would have been in $\mathbf{u}$ prior to the irregularly clocking, since $\hat{u}_{\hat{k}(t)} = z_t$. Let $k_t = k(t)$ and $l = l_\mathrm{u}$ for simplicity. Using the method from Section 3.2 we build the equation set

$$\begin{aligned}
\hat{u}_{k_0} &= h_{0,k_0}u_0 + h_{1,k_0}u_1 + \cdots + h_{l-1,k_0}u_{l-1}\\
\hat{u}_{k_1} &= h_{0,k_1}u_0 + h_{1,k_1}u_1 + \cdots + h_{l-1,k_1}u_{l-1}\\
&\ \vdots\\
\hat{u}_{k_{l-1}} &= h_{0,k_{l-1}}u_0 + h_{1,k_{l-1}}u_1 + \cdots + h_{l-1,k_{l-1}}u_{l-1},
\end{aligned}$$

where $u_0, u_1, \ldots, u_{l-1}$ are the unknown variables. Next we try to solve the set using the Gaussian reduction algorithm. If the set is solvable it will output $\mathbf{u}^\mathrm{I}$ and we know that the guess $\hat{\mathbf{s}}^\mathrm{I}$ was correct guess for $\mathbf{s}^\mathrm{I}$. If it is not solvable, the guess was wrong and we continue the search. Since the Gaussian algorithm has complexity (l_u^3) and we have to go through all possible guesses for $\mathbf{s}^\mathrm{I}$, the total runtime complexity for the attack is $O(2^{l_\mathrm{s}} l_\mathrm{u}^3)$.

In this thesis, in the paper "Improved Linear Consistency Attack on Irregularly Clocked Keystream Generators" we present a linear consistency attack that has runtime complexity $O(2^{l_\mathrm{s}})$. Hence, the runtime complexity of this attack is independent of the length of LFSR$_\mathrm{u}$.

5 A New Generator

One of the main reasons for using LFSRs as building blocks in stream ciphers is that they are easy to implement, especially in hardware. However, such ciphers can be vulnerable to attacks due to the linear properties of the LFSRs. To prevent attackers from using the weaknesses, new ciphers tends to combine the LFSRs with more and more complex components. In addition, even smaller embedded systems are getting relatively advanced with complex processors. This justifies the search for new and fast methods that use the complex capacities of the modern processors to generate nonlinear balanced sequences with maximum length.

A new method for generating maximum length sequences is to use *T-functions*. Let $\mathbf{y} = f(\mathbf{x})$ be a function from n input bits to n output bits. Let $\mathbf{x}$ (and $\mathbf{y}$) be defined by the bit vector $\mathbf{x} = (x_{n-1}, x_{n-2}, \ldots, x_0)$ or by the equivalent $\mathbf{x} = x_0 + 2x_1 + \cdots + 2^{n-1}x_{n-1}$. The function $f(\mathbf{x})$ is a *triangular function* (T-function) if bit y_j of the output $\mathbf{y}$ is only dependent on the bits $(x_j, x_{j-1}, \ldots, x_0)$ in the input.

5.1 The Klimov-Shamir T-function

In [12,11] Klimov and Shamir proposed a generator based on the T-function $f(\mathbf{x}) = \mathbf{x}^2 \vee \mathbf{C} + \mathbf{x}$ that can be used to produce maximum length sequences. The generator is word based and is given by

$$\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n,$$

where $\vee$ is the bitwise *or* operation, $\mathbf{x}_i = x_{i,0} + 2x_{i,1} + \cdots + 2^{n-1}x_{i,n-1}$ and $\mathbf{C} = C_0 + 2C_1 + \cdots + 2^{n-1}C_{n-1}$ for $x_{i,j}$ and C_j in F_2.

We can organize the sequence in an l times n matrix $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1})$ where l is the length of the sequence and n is the state length. Hence, the state at time t is the t'th row in $\mathbf{x}$. One of the major properties of the sequence is that column j in $\mathbf{x}$ has period 2^{j+1} where the first row is $j = 0$. Hence, the least significant bits have very low period and are insecure.

The generator uses both squaring and adding which are much more complex than the basic 'and','or', 'xor' and 'shift' operations often used in stream ciphers. Modern processors perform these operations quickly, so the generator is fast in software. However, some bits in the sequences produced by the generator have weaknesses. Therefore, they must be used with care. Only a few of the bits from internal state can be used, which slows down the bit rate of the generator.

Klimov and Shamir proposed a simple and experimental cipher in [11] to demonstrate the strength of the T-function. Let the internal state of the generator at time i be $(x_{i,n-1}, x_{i,n-2}, \ldots, x_{i,0})$. Let m be a chosen number for $0 < m < n$. Since the least significant bits have low period, the generator outputs only the m most significant bits $\mathbf{z}_i = (x_{i,n-1}, \ldots, x_{i,n-m})$ of $\mathbf{x}_t$ as keystream bits. The bits $(x_{i,n-m-1}, \ldots, x_{i,1}, x_{i,0})$ are kept secret. The secret key in this cipher is the $n - m$ least significant bits $(x_{0,n-m-1}, \ldots, x_{0,1}, x_{0,0})$ of the initialization state $\mathbf{x}_0$.

5.2 Cryptanalysis of the Klimov-Shamir T-function

There have only been a few attacks on the Klimov-Shamir T-function, and they are often focused on the low period of the least significant columns in $\mathbf{x}$. In [11] Klimov and Shamir presented an attack on the simple cipher described above. Let $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ be a sequence generated by the T-function with $\mathbf{C} = 5$ and let $m = n/2$. Assume that the m least significant bits in $\mathbf{x}_i$ are zero for some i, that is $\mathbf{x}_i = (x_{i,n-1}, x_{i,n-2}, \dots, x_{i,n-m}, 0, \dots, 0)$. Then $\mathbf{x}_i^2 = \mathbf{0} \bmod 2^n$ and we get that $\mathbf{x}_{i+1} = \mathbf{x}_i^2 \vee 5 + \mathbf{x}_i = 5 + \mathbf{x}_i$. Hence, $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ become equal in the m most significant bits. Further, if $\mathbf{x}_i$ is zero in the least m significant bits, then the same will be for $\mathbf{x}_{i+2^m}$. This can be used to find such i that has 0 in the secret m least significant bits of $\mathbf{x}_i$.

The attack is as follows. For all $i < 2^m$ test if $\mathbf{z}_i = \mathbf{z}_{i+1}$. If a such t is found, then test if $\mathbf{z}_{i+k2^m} = \mathbf{z}_{i+1+k2^m}$ for all $k < T$ for some threshold T. If the test does not hold, we continue the search. If the test holds for all k we stop the algorithm and assumes that $(\mathbf{x}_{i,n-m-1}, \dots, \mathbf{x}_{i,1}, \mathbf{x}_{i,0})$ is zero. We now know $\mathbf{x}_i$, and using an algorithm from [11] we can generate the stream backward from $\mathbf{x}_i$ and get $\mathbf{x}_0$.

References

1. Robert L. Benson. The Venona story. Published by National Security Agency at http://www.nsa.gov/publications/publi00039.cfm.
2. Elwyn R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill Book Co., 1968.
3. V. Chepyzhov, T. Johansson, and B. Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In *Fast Software Encryption, FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195. Springer-Verlag, 2001.
4. Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher snow. In *SAC '02: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, Lecture Notes in Computer Science, pages 47–61. Springer-Verlag, 2003.
5. G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268 – 278, March 1973.
6. P.R. Geffe. How to protect data with ciphers that are really hard to break. *Electronics*, -:99–101, 1973.
7. Jovan Golic, Mahmoud Salmasizadeh, Andrew Clark, Abdollah Khodkar, and Ed Dawson. Discrete optimisation and fast correlation attacks. In *Proceedings of Cryptography: Policy and Algorithms Conference (CPAC '95)*, volume 1029 of *Lecture Notes in Computer Science*, pages 186–200, 1995.
8. G.Vernam. Cipher printing telegraph system for secret wire and radio telegraphic communications. *Journal of American Institute of Electrical Engineers*, (45):109–115, 1926.
9. Thomas Johansson and Fredrik Jönsson. Fast correlation attacks on stream ciphers via convolutional codes. In *Advances in Cryptology-EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1999.
10. Thomas Johansson and Fredrik Jönsson. Theoretical analysis of a correlation attack based on convolutional codes. In *Proceedings of 2000 IEEE International Symposium on Information Theory*, IEEE Trans. Comput., page 212, 2000.
11. Alexander Klimov and Adi Shamir. Cryptographic applications of T-functions. In *SAC*, 2003.
12. Alexander Klimov and Adi Shamir. A new class of invertible mappings. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 470 – 483. Springer-Verlag, 2003.
13. J.L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15:122–127, Jan 1969.

14. M. Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology-EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1994.
15. W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In *Advances in Cryptology-EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 301–314. Springer-Verlag, 1988.
16. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
17. M. Mihaljevic and J. Golic. A fast iterative algorithm for a shift register initial state reconstruction given the noisy output sequence. In *Advances in Cryptology - AUSCRYPT'90*, volume 453 of *Lecture Notes in Computer Science*, pages 165–175. Springer-Verlag, 1990.
18. Håvard Molland. Improved linear consistency attack on irregularly clocked keystream generators. In *Fast Software Encryption, FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 109 – 126. Springer-Verlag, 2004.
19. Håvard Molland and Tor Helleseth. An improved correlation attack against irregular clocked and filtered keystream generators. In *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 373–389. Springer-Verlag, 2004.
20. Håvard Molland and Tor Helleseth. Linear properties in the Klimov-Shamir T-function. Submitted to IEEE Transactions on Information Theory. Accepted for presentation at ISIT-2005, 2005.
21. Håvard Molland, John Erik Mathiassen, and Tor Helleseth. Improved fast correlation attack using low rate codes. In *Cryptography and Coding, IMA 2003*, volume 2898 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2003.
22. Gregory G. Rose. A stream cipher based on linear feedback over GF(2^8). In *ACISP '98: Proceedings of the Third Australasian Conference on Information Security and Privacy*, Lecture Notes in Computer Science, pages 135–146. Springer-Verlag, 1998.
23. Claude E. Shannon. A mathematical theory of communication. *Bell*, 27:379–423, 623–656, 1948.
24. T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Trans. on Computers*, C-34:81–85, 1985.
25. Andrew J. Viterbi. Error bounds for convolutional codes and an asymtotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260 – 267, April 1967.
26. K. Zeng, C. Yang, and Y. Rao. On the linear consistency test (LCT) in cryptanalysis with applications. In *Advances in Cryptology-CRYPTO '89*, number 435 in Lecture Notes in Computer Science, pages 164–174. Springer-Verlag, 1990.

Improved Fast Correlation Attack Using Low Rate Codes

Håvard Molland, John Erik Mathiassen, Tor Helleseth

The Selmer Center**,
Dept. of Informatics,
University of Bergen
P.B. 7800 N-5020 BERGEN
Norway

Abstract. In this paper we present a new and improved correlation attack based on maximum likelihood (ML) decoding. Previously the code rate used for decoding has typically been around $r = 1/2^{14}$. Our algorithm has low computational complexity and is able to use code rates around $r = 1/2^{33}$. This way we get much more information about the key bits. Furthermore, the run time for a successful attack is reduced significantly and we need fewer keystream bits.

1 Introduction

Linear feedback shift registers, *LFSRs*, are popular building blocks for stream ciphers, since they are easy to implement, easy to analyze, and they have nice statistical properties. However, a linear shift register is not a cryptographic secure function in itself. Assuming we know the connection points in the LFSRs, we just need to know a few bits of the keystream to find the key bits, by using the linear properties in the streams to set up an equation set that is easily solved.

To make such a cipher system more secure, it is possible to combine n LFSRs with a nonlinear function f in such a way that linear complexity becomes very high. Fig. 1 describes an example for this model. The keystream $\mathbf{z} = (z_0, z_1, \ldots, z_t, \ldots, z_{N-1})$ is generated by $z_t = f(u_t^1, u_t^2, \ldots, u_t^n)$ and the linearity in the bit streams $\mathbf{u}^i = (u_0^i, u_1^i, \ldots, u_t^i, \ldots, u_{N-1}^i)$ from the n LFSRs is destroyed. The plain text $\mathbf{m}$ of length N is then encrypted to cipher text $\mathbf{c}$ by $c_t = z_t \oplus m_t$, $0 \leq t < N$.

There exist different types of attacks on systems based on this scheme. The type of attack we describe in this paper is the correlation attack. The attack uses the fact that there often exist some correlations between the bits in some of the shift register streams and the keystream $\mathbf{z}$. This can be formulated as the crossover probability $p = P(u_t \neq z_t)$, where u_t is the bit stream from a LFSR that has a correlation with $\mathbf{z}$. When $p \neq 0.5$, it is possible to do a correlation attack. Thus, the model is to decode a LFSR stream that has been sent through a binary symmetric channel (BSC) with crossover probability p.

** This work was supported by the Norwegian Research Council under Grant 146874/420.

The simplest correlation attack[8] choses the shift register LFSR$_i$ that has a correlation to the keystream bit z. Then the initialization bits $\hat{\mathbf{u}}^I$ for the LFSR are guessed and the bit stream $\hat{\mathbf{u}} = (\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{N-1})$ is generated. If for a chosen threshold p_{tr} there exists a correlation between the guessed bit stream $\hat{\mathbf{u}}$ and $\mathbf{z}$ such that $P(u_t \neq z_t) < p_{tr} < 0.5$ for $0 \leq t < N$, it is assumed that the correct initialization bits are found. This attack has a complexity of $O(2^{l_i} \cdot N)$ which is much better than $O(2^{l_1+l_2+\cdots+l_n})$, the complexity for guessing the initialization bits for all the LFSRs.

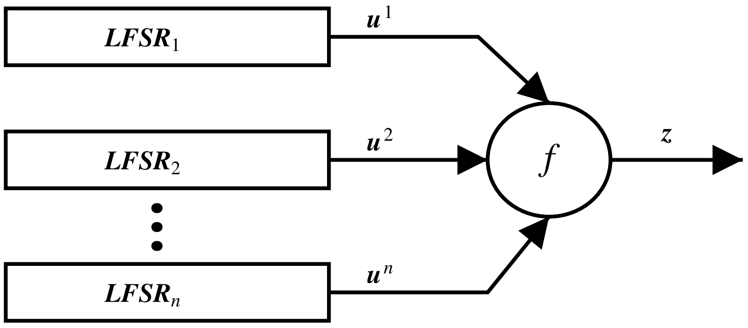

Fig. 1. An example of a stream cipher we are are able to attack using fast correlation attacks. The linear feedback shift registers LFSR$_i$ of length l_i, for $1 \leq i \leq N$, are sent through a nonlinear function f to generate the keystream $\mathbf{z}$

The complexity for guessing all the bits in a given LFSR$_i$ can be too high. To get around this, the fast correlation attack was developed[6,7] by Meier and Staffelbach. This attack uses parity check equations and reconstructs $\mathbf{u}$ from $\mathbf{z}$ using an iterative decoding algorithm. The attack works well when the polynomial that defines the LFSR$_i$ has few taps, but fails when the polynomial has many taps.

In [2] Johansson and Jönsson presented a better attack that works for LFSRs with many taps. Using a clever search algorithm, they find parity equations that are suitable for convolutional codes. The decoding is done using the Viterbi algorithm, which is maximum likelihood. This attack is briefly explained in Sect. 2.

In [9] David Wagner found a new algorithm to solve the generalized birthday problem. In this paper we present an algorithm based on the same idea that finds many equations suitable for correlation attacks. The problem with this algorithm is that it finds many but weak equations, and previous attacks would not be very effective since the code rate will be very low.

In this paper we present an improvement on the attacks based on ML decoding. While Johansson and Jönsson use few but strong equations, we go in the opposite direction and use many and weak equations. We present a new algorithm that is capable of performing an efficient ML decoding even when the code rate is very low. This gives us much more information about the secret initialization bits, and the run time complexity goes down considerably. For a crossover probability $p = 0.47$, polynomial of degree $l = 60$ and the number of known keystream bits $N = 100 \cdot 10^6$, our attack has complexity of order 2^{39}, while the previous convolutional code attack[2,4] has complexity of order 2^{48}. See Table 2 in Sect. 5 for more simulation results compared to previous attacks.

The paper will be organized as follows. First we will give a brief description of the systems we try to attack. In Sect. 2 we will describe the basic mathematics and some important previous attacks. In Sect. 3 we describe an efficient method for finding parity check equations, using the generalized birthday problem. In Sect. 4 we present a new algorithm that is capable of using the huge number of equations found by the method in Sect. 3.

2 Definitions and Previous Attacks

First we will define the basic mathematics for the correlation attacks in this paper.

2.1 The Generator Matrix

Let $g(x) = 1 + g_{l-1}x + g_{l-2}x^2 + \cdots + g_1x^{l-1} + x^l$ be the primitive feedback polynomial over $\mathbb{F}_2$ of degree l for a linear feedback register, LFSR, that generates the sequence $u = (u_0, u_1, \ldots, u_{N-1})$. The corresponding recurrence is $u_t = g_1u_{t-1} + g_2u_{t-2} + \cdots + g_{l-1}u_{t-l+1} + u_{t-l}$. Let α be defined by $g(\alpha) = 0$. From this we get the reduction rule $\alpha^l = g_1\alpha^{l-1} + g_2\alpha^{l-2} + \cdots + g_{l-1}\alpha + 1$. Then we can define the generator matrix for sequence u_t, $0 < t < N$ by the $l \times N$ matrix

$$G = [\alpha^0\alpha^1\alpha^2 \ldots \alpha^{N-1}]. \tag{1}$$

For each $i > l$, using the reduction rule, α^i can be written as $\alpha^i = h^i_{l-1}\alpha^{l-1} + \cdots + h^i_2\alpha^2 + h^i_1\alpha + h^i_0$. We see that every column $i \geq l$ is a combination of the first l columns. Any column i in G can be represented by

$$\mathbf{g}_i = [h^i_0, h^i_1, \ldots, h^i_{l-1}]^{\mathbf{T}}. \tag{2}$$

Thus the sequence u with length N and initialization bits $\mathbf{u}^I = (u_0, u_1, \ldots, u_{l-1})$, can be generated by

$$\mathbf{u} = \mathbf{u}^I G.$$

The shift register is now turned into a (N, l) block code.

Example 1. Let $g(x) = x^4 + x^3 + 1$. Using the reduction rule we get $\alpha^4 = \alpha^3 + 1$, $\alpha^5 = \alpha(\alpha^3 + 1) = \alpha^4 + \alpha = \alpha^3 + \alpha + 1$ and so on. We choose $N = 10$, and set $G = [\alpha^0\alpha^1 \ldots \alpha^9]$. The sequence $\mathbf{u}$ is generated by the 4×10 matrix G like this,

$$\mathbf{u} = \mathbf{u}^I G = [u_0, u_1, u_2, u_3] \begin{bmatrix} 1\,0\,0\,0\,1\,1\,1\,1\,0\,1 \\ 0\,1\,0\,0\,0\,1\,1\,1\,1\,0 \\ 0\,0\,1\,0\,0\,0\,1\,1\,1\,1 \\ 0\,0\,0\,1\,1\,1\,1\,0\,1\,0 \end{bmatrix}. \tag{3}$$

The reason that we use a generator matrix, is that we easily can see from G which initialization bits $(u_0, u_1, \ldots, u_{l-1})$ sum to u_i for every $0 \leq i < N$ by looking at column i. For example the bit u_9 (last column) in the example above is calculated by $u_9 = u_0 + u_2$, and it is independent of the initialization bits u_1 and u_3.

2.2 Equations

In [2] Johansson and Jönsson presented the following method for finding equations that are usable for decoding.

Let u be a sequence generated by the generator polynomial $g(x)$ with degree l. If we can find w columns in the generator matrix G that summarize to zero in the $l - B$ last bits,

$$(\mathbf{g}_{i_0} + \mathbf{g}_{i_1} + \ldots + \mathbf{g}_{i_{w-1}})^{\mathbf{T}} = (c_0, c_1, \ldots, c_{B-1}, \underbrace{0, 0, \ldots, 0}_{l-B}), \tag{4}$$

for a given B, $0 < B \leq l$, and $l \leq i_0, i_1, \ldots, i_{w-1} < N$, we get an equation of the form

$$c_0 u_0 + c_1 u_1 + \cdots + c_{B-1} u_{B-1} = u_{i_0} + u_{i_1} + \cdots + u_{i_{w-1}}. \tag{5}$$

This can be seen by noting that column i in G shows which of the initialization bits $\mathbf{u}^I = (u_0, u_1, \ldots, u_{l-1})$ that summarize to the bit u_i in the sequence $\mathbf{u}$. When two columns i and j in G sum to zero in the last $l - B$ entries $(u_B, u_{B+1}, \ldots, u_{l-1})$, the sum $u_i + u_j$ is independent of those bits. Then we can concentrate on finding just the B first bit of $\mathbf{u}^I$. The equation (5) is cyclic and can therefore be written as

$$c_0 u_t + c_1 u_{t+1} + \cdots + c_{B-1} u_{t+B-1} = u_{t+i_0} + u_{t+i_1} + \cdots + u_{t+i_{w-1}}, \tag{6}$$

for $0 \leq t < N - i_{w-1}$.

Example 2. Let $w = 2$, and $B = 1$. If we examine the matrix G in equation (3), we see that $(\mathbf{g}_6 + \mathbf{g}_8)^{\mathbf{T}} = (1, 1, 1, 1) + (0, 1, 1, 1) = (1, 0, 0, 0)$. From this we get $c_0 = 1$, $i_0 = 6$ and $i_1 = 8$ and the equation is $u_0 = u_6 + u_8$. Because of the cyclic structure we finally get $u_t = u_{t+6} + u_{t+8}$. This equation will hold for every sequence that is generated with $g(x) = x^4 + x^3 + 1$ as feedback polynomial.

In section 3 we will go further into how the actual search for columns that sum to zero in the last $l - B$ bits can be done efficiently.

2.3 Principle for Decoding

In [1] Chepyzhov, Johansson and Smeets presented a simple maximum likelihood algorithm that uses equations found in Sect. 2.2 for decoding. We will now briefly describe this algorithm. First we take equation (5) and make the right side of the equation point to the corresponding keystream bits $\mathbf{z}$ instead of $\mathbf{u}$. From this we get the following equation,

$$c_0u_0 + c_1u_1 + \cdots + c_{B-1}u_{B-1} \approx z_{i_0} + z_{i_1} + \cdots + z_{i_{w-1}}. \tag{7}$$

Let m be the number of equations found by the method in Sect. 2.2. Then we get the equation set

$$\begin{aligned} c_{0,0}u_0 + c_{0,1}u_1 + \ldots + c_{0,B-1}u_{B-1} &\approx z_{i_{0,0}} + z_{i_{0,1}} + \cdots + z_{i_{0,w-1}} \\ c_{1,0}u_0 + c_{1,1}u_1 + \ldots + c_{1,B-1}u_{B-1} &\approx z_{i_{1,0}} + z_{i_{1,1}} + \cdots + z_{i_{1,w-1}}. \\ &\vdots \\ c_{m-1,0}u_0 + c_{m-1,1}u_1 + \ldots + c_{m-1,B-1}u_{B-1} &\approx z_{i_{m-1,1}} + z_{i_{m-1,2}} + \cdots + z_{i_{m-1,w-1}} \end{aligned} \tag{8}$$

We use $'\approx'$ to notify that the equations only hold with a certain probability.

Here $(u_0, u_1, \ldots, u_{B-1})$ are the unknown secret bits we want to find and $\mathbf{z}$ is the keystream. Remember that u_t and z_t are equal with a probability $1-p$ where $p = P(u_t \neq z_t)$. Thus, each equation in (8) will hold with a probability

$$P_w = \frac{1}{2} + 2^{w-1}(\frac{1}{2} - p)^w, \tag{9}$$

using the Piling up lemma[5]. Replace the bits $(u_0, u_1, \ldots, u_{B-1})$ in the set (8) with a guess $\hat{\mathbf{U}} = (\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{B-1})$. If $(\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{B-1}) \neq (u_0, u_1, \ldots, u_{B-1})$, (that is, if one or more of the guessed bits are wrong) each equation will hold with a probability $P = 0.5$. If the guess is right, each equation will hold with a probability $P_w > 0.5$. We see that the (N, l) block-code is reduced to a (m, B) block-code, and the decoding problem is to decode message blocks of length B that are sent as codewords of length m through a channel with crossover probability $1 - P_w$.

The decoding can be done the following way. For all the 2^B possible guesses for $\hat{\mathbf{U}} = (\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{B-1})$, test $\hat{\mathbf{U}}$ with all the equations in the set (8), and give the guess one point for every equation in the set that holds. Afterward, assume that $(u_0, u_1, \ldots, u_{B-1}) = \hat{\mathbf{U}}$ for the guess of $\hat{\mathbf{U}}$ that has the highest score. In this way we get the first B bits of the secret initialization bits $(u_0, u_1, \ldots, u_{l-1})$. The procedure can be repeated to find the rest of the bits $(u_B, u_{B+1}, \ldots, u_{l-1})$.

The complexity for this algorithm is

$$O(2^B \cdot m) \tag{10}$$

since we have to test m equations on the 2^B different guesses of $\hat{\mathbf{U}}$.

2.4 Fast Correlation via Convolutional Codes

In [2] Johansson and Jönsson showed how the equation set (8) can be used to decode the keystream $\mathbf{z}$ via convolutional codes. The problem is formulated as decoding of a $(m, 1, B)$ convolutional code, and the decoding is done using the Viterbi algorithm. This algorithm is optimal, but has relatively high usage of memory. In convolutional codes the coding is done over T bits. Using the fact that the equations are cyclic, the algorithm in Sect. 2.3 is used for calculating the metrics for each state $\hat{\mathbf{U}}$ at time t, $0 \leq t < T$. The algorithm in Sect. 2.3 is actually a special case of the fast correlation attack via convolutional code with $T = 1$. When the metrics are calculated, we try to find the longest possible path through the states $0 \leq t < T$. We see that the problem is transformed into finding the longest path trough a $2^B \times T$ trellis. The Viterbi algorithm is optimal for solving this problem. We refer to [2] for details about the convolutional attacks.

2.5 Theoretical Analysis and Complexity

In [4] Johansson and Jönsson, presented a theoretical estimate of the success rate for fast correlation attacks via convolutional codes.

For a given bit stream of length N generated by a shift register with feedback polynomial $g(x)$, the expected number of equations of type (5) is

$$E(m) = \frac{\binom{N-T-l}{w}}{2^{l-B}} \approx \frac{\binom{N}{w}}{2^{l-B}} \tag{11}$$

Let $p_e < l \cdot 2^{-B}$ and $p = P(z_t \neq u_t)$. Then the convolutional attack described in Sect. 2.4 has a success with probability $1 - p_e$ if

$$p \leq \frac{1}{2} - \frac{1}{2}\left(\frac{8\ln 2}{m}\right)^{\frac{1}{2w}}. \tag{12}$$

The probability p is set by the stream cipher. The closer p is to 0.5, the more equations are needed to fulfill (12). One way to find more equations is to increment w, the number of bits on the right hand side of the equations. If we do this, each equation we find gets weaker, see equation (9). However, although each equation is weaker, we find so many of them that they together give more information about the unknown key bits $\mathbf{u}^I$. The problem with this is, as shown below, that the complexity of the attack also increases when we use many more equations. In Sect. 4 we will describe a new method to solve this problem.

The complexity of the convolutional attack in [2,4] is $O(2^B \cdot m \cdot T)$, since we decode over T bits. This can be rewritten using equation (11) and noting that $m = 2^B \frac{\binom{N}{w}}{2^l}$. Let $o = \frac{\binom{N}{w}}{2^l}$. In this way we see that the complexity can be formulated as

$$O(2^{2B} \cdot o \cdot T), \tag{13}$$

The complexity for the simple attack in [1] is $O(2^{2B} \cdot o)$. Using this formulation, we see that if we use all the equations for given w, N, B and l, the run time complexity increases with a factor 4, when we increment B by one.

3 Methods for Finding Equations

In this section we will describe a fast method for finding many equations. The method is in some ways similar to the solution of the generalized birthday problem that Wagner presented in [9].

We have an equation of the form (5) if we find w columns in the generator matrix G of length N that sum to zero in the last $l-B$ positions. For $w=2$ we sort the N columns from the generator matrix. Equal columns will then be located next to each other. The complexity for this method is $O(N \log_2 N)$.

3.1 A Basic Method for Finding Equations with $w > 2$

We will now describe a simple and well known algorithm for finding equations when $w > 2$.

First we sort the columns in the $l \times N$ generator matrix G according to the values in last $l-B$ bits. Then we run through all possible sums of $w-1$ columns, and search for columns in G that are equal to the sums in the last $l-B$ bits. The sum of these w columns is then zero in the $l-B$ last bits. The complexity of this algorithm will be $O(N^{w-1} \log_2 N)$.

This method is straightforward and seems good since we find all the equations. The problem is when $l-B$ becomes big, since it is less likely that the sum of the combination of $w-1$ columns matches a single column. The number of possible different values in the $l-B$ last bits are 2^{l-B}. If we pick a random combination of $w-1$ columns we will have a probability less than $P_m = N/2^{l-B}$ of getting a match from the sorted generator matrix. If $N = 2^{20}$, $B = 15$ and $l = 40$ then $P_m = 2^{-5}$, so on average each 32'th sum combination will give an equation. If we increase the degree of the feedback polynomial to $l = 60$, the probability of finding an equation for given $w-1$ columns will be reduced to $P_m = 2^{-25}$. Since an equation with $w = 4$ is a very weak equation, we need millions of equations in most cases.

Table 1. The table shows the percentage of the total number of equations we need for a successful convolutional attack when $N = 2^{21}$, $l = 60$, $w = 4$ and $B = 20$

p	v
0,41	0.00068
0,43	0,0051
0,45	0,075%
0,47	4.5%

The method above finds all the equations, but in fact we do not need all the equations for the attack to succeed. From (12) we get the equation

$$m_s = \frac{8 \ln 2}{(1-2p)^{2w}},$$

where m_s is the number of equations needed for success for a given crossover probability p. Then $v = \frac{m_s}{m}$ will give us the rate of the total number of equations m needed for a successful attack. Table 1 shows different rates needed for different attacks. The fact that we do not need all the equations indicates that we may use a fast method to find a subset of them.

3.2 A Method for Finding All the Equations with $w = 4$

The method described here works in certain situations where the parameters are small. The algorithm works as follows. In the first step we go through all the possible sums of pairs $\mathbf{g}_{j_0}$, $\mathbf{g}_{j_1}$, of columns in G. These sums are stored in a matrix G_2 and the indexes of the two columns in G are also stored. In the second step we sort the matrix G_2 according to the last $l-B$ bits. Then we search for the columns in G_2 that are equal in the last $l-B$ bits. Let $\mathbf{f}_{j_*} = \mathbf{g}_{i_{*,0}} + \mathbf{g}_{i_{*,1}}$ be the sums of pairs found in G. In this way we get weight 4 equations of the form:

$$(\mathbf{f}_{j_0} + \mathbf{f}_{j_1})^{\mathrm{T}} = (\mathbf{g}_{i_{0,0}} + \mathbf{g}_{i_{0,1}} + \mathbf{g}_{i_{1,0}} + \mathbf{g}_{i_{1,1}})^{\mathrm{T}} = (c_0, c_1, \ldots, c_{B-1}, \underbrace{0, \ldots, 0}_{l-B}) \qquad (14)$$

where the $\mathbf{f}_j$'s are columns in G_2 and the $\mathbf{g}_j$'s are columns in G.

By this method we will find all the equations 3 times. The reason for this is illustrated by the equation

$$\mathbf{g}_{i_{0,0}} + \mathbf{g}_{i_{0,1}} + \mathbf{g}_{i_{1,0}} + \mathbf{g}_{i_{1,1}} = 0 \Longleftrightarrow \mathbf{g}_{i_{0,0}} + \mathbf{g}_{i_{0,1}} = \mathbf{g}_{i_{1,0}} + \mathbf{g}_{i_{1,1}}.$$

Two other pairs giving the same equation is $\mathbf{g}_{i_{0,0}} + \mathbf{g}_{i_{1,1}} = \mathbf{g}_{i_{0,1}} + \mathbf{g}_{i_{1,0}}$ and $\mathbf{g}_{i_{0,0}} + \mathbf{g}_{i_{1,0}} = \mathbf{g}_{i_{0,1}} + \mathbf{g}_{i_{1,1}}$. This collisions are avoided if the pairing in the second step has a restriction. All the indexes on the left side of the equation must all be less or greater than the indexes on the right side. In this way $\frac{2}{3}$ of the equations will be thrown away, but the remaining $\frac{1}{3}$ will represent all the equations. This method will be impractical if N is big, since G_2 will have a length of $N_2 = \binom{N}{2}$.

3.3 A Fast Method for Finding a Subset of the Equations with $w = 4$

Here we will solve the problem concerning memory requirement in the algorithm presented above. Using this algorithm we are able to find all the equations, but the number of possible sums in step one is far too many. If we can reduce the size of G_2, without reducing the number of equations significantly, we have succeeded.

Algorithm 1 The algorithm for finding a subset of all the equations with $w = 4$

```
Input: $G$, $N$, $B_2$, $B_4 < B_2$, $l$.
Step 1:
sort the $l \times N$ matrix $G$ according to the last $l - B_2$ bits.
For $0 \leq i_0, i_1 < N$ find all pairs of columns $\mathbf{g}_{i_0}$ and $\mathbf{g}_{i_1}$ that sums to
    $\mathbf{f}^{\mathrm{T}} = (\mathbf{g}_{i_0} + \mathbf{g}_{i_1})^{\mathrm{T}} = (d_0, d_1, \ldots, d_{B_2-1}, \underbrace{0, \ldots, 0}_{l-B_2})$
    Add $\mathbf{f}$ and the indexes $i_0, i_1$to the $G_2$ matrix.

Step 2:
sort $l \times N_2$ matrix $G_2$ according to the last $l - B_4$ bits.
For $0 \leq j_0, j_1 < N_2$ find all pairs of columns $\mathbf{f}_{j_0}$ and $\mathbf{f}_{j_1}$ in $G_2$ that sums to
    $(\mathbf{f}_{j_0} + \mathbf{f}_{j_1})^{\mathrm{T}} = (c_0, c_1, \ldots, c_{B_4-1}, \underbrace{0, \ldots, 0}_{l-B_4}) = (\mathbf{g}_{j_{0,0}} + \mathbf{g}_{j_{0,1}} + \mathbf{g}_{j_{1,0}} + \mathbf{g}_{j_{1,1}})^{\mathrm{T}}$
where $j_{*,0}$ and $j_{*,1}$ are the indexes from $G$ associated to $\mathbf{f}_{j_*}$.
    Add $c_0, c_1, \ldots, c_{B-1}$ and the indexes $j_{0,0}, j_{0,1}, j_{1,0}, j_{1,1}$ to $F$.

Return: $F$
```

The algorithm is divided into two steps. In step one we find a subset of all the sums, where the pairing in step 2 only involves elements in that subset. The sum of two columns that are unequal in the last bits will never be zero. Therefore, we may look for sums of pairs in step 1 where we require a certain value in the last $l - B_2$ positions. Without loss of generality we require zeroes in the last $l - B_2$ positions in G_2.

Let $B_4 < B_2 < l$. First we sort the columns in G according to the last $l - B_2$ positions. Then we go through the matrix and find the pairs of columns $\mathbf{f} = \mathbf{g}_{i_0} + \mathbf{g}_{i_1}$that sum to zero in the last $l - B_2$ positions and store them in matrix G_2. The original positions of the columns in the sum are also stored. The size of G_2 is thereby reduced by a factor of 2^{l-B_2}. In the second step we repeat the algorithm using B_4 on G_2. We sort the matrix G_2 according to the last $l - B_4$ bits, in order to find pairs of columns from G_2, where the sum is zero in the last $l - B_4$ bits. In this way we get weight 4 equations of the form (5). The pseudo code for this is shown in Algorithm 1.

Algorithm 1 is a method which may keep the memory requirements sufficiently low. From (11) we get the size N_2 of G_2,

$$N_2 = \frac{\binom{N}{2}}{2^{l-B_2}} \approx \frac{N^2}{2^{l-B_2+1}}.$$

It is possible to run this algorithm several times to find even more equations. Instead of keeping the last $l - B_2$ bits zero in the first step, we may repeat this algorithm requiring these bits having the fixed values $(d_{B_2}, d_{B_2+1}, \ldots, d_l) \neq \mathbf{0}$. We may choose to only vary the first two bits, and run the algorithm 2^2 times. Thus we get four times as many equations compared to running it only once. The cost is that we have to sort the matrices G and G_2

Using Algorithm 1 some of the equations we get will be equal, called collisions. If we use algorithm 1 repeatedly while changing r bits (we repeat 2^r times), this bound is

$$p(collision) < 2^{(B_2+r)-l} + 2^{2(B_2+r-l)} < 2 \cdot 2^{(B_2+r-l)} = 2^{(B_2+r+1-l)}$$

since $B_2 + r - l < 0$. If we do not use repetitions we set $r = 0$. In practical attacks, this probability will be very low, and the simulations show that this has little impact on the decoding.

4 Fast Decoding using Quick Metric

In Sect. 3 we presented a fast method for finding a huge number of equations. These equations can give us a lot of information about the initialization bits. However, since there are so many of them, we get two new problems. It will take too much memory to store all the equations, and the complexity will be too high when we use them to calculate the metrics during decoding. Thus, we need an efficient method for storing the equations, and an efficient method for using them.

The complexity for calculating the metrics by the method in Sect. 2.3, is $O(2^B \cdot m)$, where m is the number of equations and B is the message block size of the code. If m is very high, the decoding problem can be to complex. We reduce the decoding complexity to $O(2^{2B} + m)$ by the following two methods referred to as *Quick Metric*.

4.1 A New and Efficient Method for Storing the Equations

Let $m \gg 2^B$ be the number of equations found using the method described in Sect. 3 with $B = B_4$. We get an equation set like (8). The main observation here is that although there are m different equations, there exist only 2^B different versions of the left side of the equations. This means that many equations will share the same left sides defined by $(c_0, c_1, \ldots, c_{B-1})$ when $m \gg 2^B$. We can now use *counting sort* to store the equations. Let E be an integer array of size 2^B. When an equation of the form (5) is found, we set

$$e = c_0 + 2c_1 + 2^2c_2 + \cdots + 2^{B-1}c_{B-1}. \tag{15}$$

Then we count the equation by setting $E(e) \leftarrow E(e) + 1$.

Algorithm 2 The algorithm for storing equations (first step)

```
Input: $G$, $N$, $T$, $B$,$w$ and $z$.
For every $i_0, i_1, \ldots, i_{w-1}$, $T \leq i_0, i_1, \ldots, i_{w-1} < N - T$,
    If the columns $\mathbf{g}_{i_0}, \mathbf{g}_{i_1}, \ldots, \mathbf{g}_{i_{w-1}}$ in $G$ summarize to
        $(\mathbf{g}_{i_0} + \mathbf{g}_{i_1} + \cdots + \mathbf{g}_{i_{w-1}})^{\mathrm{T}} = (c_0, c_1, \ldots, c_{B-1}, \underbrace{1, 0, \ldots, 0}_{l-B})$
    Let $e$ be the integer value of the bits $(c_0, c_1, \ldots, c_{B-1})$.
    $E(e) \leftarrow E(e) + 1$
    For every $t$, $1 \leq t \leq T$,
        $Sum(e,t) \leftarrow Sum(e,t) + (z_{t+i_0} + z_{t+i_1} + \cdots + z_{t+i_{w-1}} \bmod 2)$
Return:The integer arrays $Sum$ and $E$
```

At this point we have stored the left side of the equation. To store the right side, we use another integer array, $sum()$, of size 2^B. Then we calculate the binary sum $s = (z_{i_0} + z_{i_1} + \cdots + z_{i_{w-1}}) \bmod 2$ for the given $(i_0, i_1, \ldots, i_{w-1})$. Finally we set $Sum(e) \leftarrow Sum(e) + s$.

When the search for equations is finished, $E(e)$ is the number of the equations of type e that was found, and $Sum(e)$ is the number of equations of type e that sum to 1 on the right hand side for a given keystream $\mathbf{z}$.

Algorithm 2 shows a pseudo code for this idea. Here the idea is expanded so that it works with decoding via convolutional codes as presented in [2,4]. We assume that the search methods in Algorithm 1 are used to find the w columns that sum to zero in the last B bits. When decoding is done via convolutional codes, the equations are cycled T times when we decode over T bits. This means that we have to calculate $Sum(e)$ for every $0 \leq t < T$, since the right side of (7) is not cyclic itself. From this we get the 2-dimensional array $Sum(e,t)$. One little detail to make this work with convolutional codes, is that the bit c_B in the sum of the columns has to be 1. However, this has no impact on the complexity for the algorithm.

4.2 A New and Efficient Method for Calculating the Metrics

Assume we have done the search for equations according to Sect. 3.3 and Algorithm 2. After this preprocessing step, we have the two arrays E and sum. Let $m_e = E(e)$ be the number of equations found of type e. Now we can test m_e equations on a guess $\hat{\mathbf{U}}$ in just one step instead of m_e .

Make a guess $\hat{\mathbf{U}}$ for $\mathbf{u}^I$. For every equation type e, do as follows: If the sum $c_0\hat{u}_0 + c_1\hat{u}_1 + \cdots + c_{B-1}\hat{u}_{B-1}$ corresponding to the equation type e (see equation 15) is 1, the number of the $m_e = E(e)$ equations that hold is $sum(e)$. The metric for the guess $\hat{\mathbf{U}}$ is incremented with $sum(e)$. If $c_0\hat{u}_0 + c_1\hat{u}_1 + \cdots + c_{B-1}\hat{u}_{B-1} = 0$,

Algorithm 3 Quick Metric algorithm (second step)

```
Input: state $\widehat{\mathbf{U}}$, time $t$, and the tables $Sum$ and $E$ .
$metric_{\widehat{\mathbf{U}}} \leftarrow 0$
For every $e$, $0 \leq e < 2^B$
    If equation $e$ over state $\widehat{\mathbf{U}}$ sums to 1,
        $metric_{\widehat{\mathbf{U}}} \leftarrow metric_{\widehat{\mathbf{U}}} + Sum(e,t)$
    Else
        $metric_{\widehat{\mathbf{U}}} \leftarrow metric_{\widehat{\mathbf{U}}} + (E(e) - Sum(e,t))$

Return: $metric_{\widehat{\mathbf{U}}}$
```

the number of the m_e equations that hold is $m_e - sum(e)$, and the metric is incremented with $m_e - sum(e)$. Algorithm 3 shows the pseudo code for this idea.

Now we have calculated the metric for one guess in just 2^B steps instead of $m > 2^B$ steps. The complexity for this part of the attack is actually independent of the amount of equations that are used, and the complexity for calculating the metrics for all the 2^B guesses is $O(2^{2B})$. The reason that the overall complexity is $O(2^{2B} + m)$, is that we have to go through all m equations once in the preprocessing, each time we want to analyze a new keystream z. Using the search algorithm in Sect. 3, we can do some processing independently from z. However, in the end we have to go through m equations and save the $z_{i_0} + z_{i_1} + \cdots + z_{i_{w-1}}$ in the array sum for each equation. This part of the search algorithm is almost linear in m.

4.3 Complexity and Properties

When we use Quick Metric, the decoding is done in two steps. The first step is the building of the equation count matrix E. The second step is decoding using the Viterbi algorithm with complexity $O(T \cdot 2^{2B})$, because of Quick Metric. The building of matrix E can be divided into 3 parts. First the sorting of G of length N, then the sorting of G_2 of length N_2. Finally we have to go through the sorted G_2 and save all the equations in E. Thus, the total complexity for the first step is $O(N \cdot \log_2 N + N_2 \cdot \log_2 N_2 + T \cdot m)$. Since m has to be very high for our attack, the complexity is most often dominated by $T \cdot m$, and the overall complexity for the first step is $O(T \cdot m)$.

It will vary which of the two steps that will dominate the complexity. Thus, the total run time complexity for both step is given by

$$O(T \cdot m + T \cdot 2^{2B}).$$

To have a success rate close to 1, the number of equations m and the convolutional memory B should satisfy equation (12) where p and l is are set by the cipher system. T must be high enough so that the algorithm converge to the right path in the trellis. $T \approx l$ is enough for most cases. The complexity for the attack in [4,2] is $O(2^{2B} \cdot o \cdot T)$, where $o = \frac{\binom{N}{w}}{2^l}$.

The first observation is that when we use Quick Metric, the computational complexity for the Viterbi algorithm is independent from the number of equations m that is used for decoding. The main difference from the attacks in [4,2] is that we just have to go through all m equations once in the first step. In [4,2] they have to go through all the m equations for every time they test one of the 2^B states. Thus, our algorithm has a big advantage when we choose to use more than 2^B equations.

A drawback for our algorithm is that we have to do the first step every time we want to decode a new stream generated by the same system. In [4,2], they just have to do the preprocessing once for each cipher system. Therefor we have to keep the complexity in the first step as low as possible. There is actual a trade off between the two steps. When the first step takes long time, the second step takes less time, and the other way around. This means that we have to choose the parameters N, B and m carefully to get the best possible attack.

The next observation is that our algorithm is stronger for lower B, since we can use many more equations. That means that we can attack a system using a lower B than is possible with the attacks in [4,2]. Thus, the run time for given B, w and m goes down considerably since B has a huge impact on the complexity.

5 Simulations

The evaluation of the attacks needs some explanation. The interesting parameters of the cipher systems we attack, are the polynomial degree l and the crossover probability p. Finally we are given a keystream of length N. We want for a given high l to be able to decode a keystream $\mathbf{z}$ where the crossover probability $p = (u_i \neq z_i)$ is as near 0.5 as possible. Of course we want to use few keystream bits and low run time complexity.

To be able to compare the different attacks, we compute the complexity for decoding as the total number of times we have to test an equation on a guessed state. The complexity for the pre-computation is computed as the number of table lookups that have to be done during the search for equations. When we use Quick Metric we have 2 steps, so the overall complexity is given by the sum of the two steps.

See Table 2 for the simulation results. It is important to notice that we have programmed and tested all the attacks in the table, and the results for p come from these tests, not the theoretical estimate (12). For this purpose we used a 2.26 GHz Pentium IV with 1 gigabyte memory running on Linux. The algorithm is fast in practice and even the biggest attack ($p = 0.47$) was done in just a few hours including the search for equations.

Table 2. The tables show our attack compared to previous attacks. The generator polynomial degree l for the LFSR is 60 for all the simulations. We set $T = 60$. The * is a theoretical estimate using the success rate equation (12)

Improved convolutional code attack

B	p	N	w	Total decoding runtime
14	**0.43**	$\mathbf{15 \cdot 10^6}$	4	$\mathbf{2^{35}}$
10	**0.43**	$\mathbf{100 \cdot 10^6}$	4	$\mathbf{2^{31}}$
16	**0.47**	$\mathbf{100 \cdot 10^6}$	4	$\mathbf{2^{39}}$
11	**0.43**	$\mathbf{40 \cdot 10^6}$	4	$\mathbf{2^{30}}$

Previous convolutional code attack[2]

B	p	N	w	Decoding runtime
20	**0.43**	$\mathbf{100 \cdot 10^6}$	2	$\mathbf{2^{38}}$
18	**0.37**	$\mathbf{600 \cdot 10^3}$	3	$\mathbf{2^{37}}$
25*	**0.47**	$\mathbf{100 \cdot 10^6}$	2	$\mathbf{2^{48}}$

Previous attack through reconstruction of linear polynomials [3]

B	p	N	w	Rounds n	Decoding runtime
25	**0.43**	$\mathbf{40 \cdot 10^6}$	2	4	$\mathbf{2^{41.5}}$

From the table we see that our attack is best when p is close to 0.5. For $p = 0.47$ the run time of our attack is dominated by the pre-computation step which is $m \cdot T \approx 2^{39}$. The parameters for this attack is $B_2 = 34$, $B = B_4 = 16$ and $m = 2^{33}$ which gives the code rate $r = 2^{-33}$. If we use the method in [2,4], the estimated run time is 2^{48} parity checks.

Another attack from Johansson and Jönsson is the the fast correlation attack through reconstruction of linear polynomials[3]. This attack has lower complexity than fast correlation via convolutional codes and it uses less memory. We can apply Quick Metric on the reconstruction algorithm, but unfortunately this will not give a better result than using it on the convolutional code attack. The reason for this is that in each round in the algorithm we would have to repeat the search for equations. To keep B sufficient low, we would have to use many rounds. Thus, the computational complexity for this would become too high.

However, when we use Quick Metric on the convolutional attack, the attack achieves in most cases a much lower run time complexity than the attack in [3]. This is shown by the two attacks in Table 2 using $N = 40 \cdot 10^6$.

6 Conclusion

We have presented a new method for calculating the metrics in fast correlation attacks. This method enable us to handle the huge number of parity check equations we get when we use $w = 4$ and the method in Sect. 3. Earlier it has only been possible to handle convolutional code rates down to around $r = 1/2^{14}$. Using our method we have decoded convolutional codes with rates down to $1/2^{32}$ in just a few hours. Because of this we have done attacks on cipher systems with higher crossover probability p than before.

An open problem is the search for equations with $w = 3$. We use $w = 4$ since there exists a fast method for finding those equations. However, the equations with $w = 4$ are weak, and this gives the first step high complexity. A good solution would be to use $w = 3$, with a fast search algorithm.

References

1. V. Chepyzhov, T. Johansson, and B. Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In *Fast Software Encryption, FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195. Springer-Verlag, 2001.
2. Thomas Johansson and Fredrik Jönsson. Fast correlation attacks on stream ciphers via convolutional codes. In *Advances in Cryptology-EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1999.
3. Thomas Johansson and Fredrik Jönsson. Fast correlation attacks through reconstruction of linear polynomials. In *Advances in Cryptology-CRYPTO'2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 300–315. Springer-Verlag, 2000.
4. Thomas Johansson and Fredrik Jönsson. Theoretical analysis of a correlation attack based on convolutional codes. In *Proceedings of 2000 IEEE International Symposium on Information Theory*, IEEE Trans. Comput., page 212, 2000.
5. M. Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology-EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1994.
6. W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In *Advances in Cryptology-EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 301–314. Springer-Verlag, 1988.
7. Willi Meier and O. staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1:159–176, 1989.
8. T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Trans. on Computers*, C-34:81–85, 1985.
9. David Wagner. A generalized birthday problem. In *Advances in cryptology-CRYPTO' 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303, 2002.

Improved Linear Consistency Attack on Irregularly Clocked Keystream Generators

Håvard Molland

The Selmer Center[**],
Dept. of Informatics,
University of Bergen
P.B. 7800 N-5020 BERGEN
Norway

Abstract. In this paper we propose a new attack on a general model for irregularly clocked keystream generators. The model consists of two feedback shift registers of lengths l_1 and l_2, where the first shift register produces a clock control sequence for the second. This model can be used to describe among others the shrinking generator, the step-1/step-2 generator and the stop and go generator. We prove that the maximum complexity for attacking such a model is only $O(2^{l_1})$.

Keywords: Stream ciphers, irregularly clocked generators, linear consistency test

1 Introduction

The objective in stream ciphers is to expand a short key into a long keystream **z** that is difficult to distinguish from a truly random bit stream. It should not be possible to reconstruct the short key from **z**. The message is then encrypted by mod-2 additions with the keystream.

In this paper we analyze additive stream ciphers where the keystream is produced by an irregularly clocked *linear feedback shift register* (LFSR). This model produces bit streams with high linear complexity, which is an important criteria for pseudo random sequences.

The cipher model we attack is composed of two $LFSRs$, $LFSR_{\mathrm{s}}$ of length l_{s} and $LFSR_{\mathrm{u}}$ of length l_{u}. $LFSR_{\mathrm{s}}$ produces a bit stream **s** and $LFSR_{\mathrm{u}}$ produces a bit stream **u**. The bit stream **s** is sent through a function $D()$. Finally $D()$ outputs the clock control sequence of integers, **c**, which is used to clock $LFSR_{\mathrm{u}}$. See Fig. 1 for an illustration, and Sec. 2.1 for a full description of the model. The effect of the irregularly clocking is that **u** is irregularly decimated. The result from the decimation is the keystream **z**. Thus, the positions of the bits in the original stream **u** are altered and the linearity of the stream are destroyed. This gives the keystream **z** high linear complexity.

[**] This work was supported by the Norwegian Research Council under Grant 146874/420.

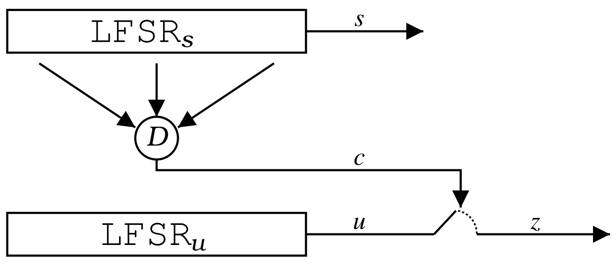

Fig. 1. The general model for irregularly clocked keystream generators

There have been several previous attacks on this scheme. One popular method is to use the constrained Levenshtein distance (CLD) (also called edit distance), which is the number of deletions, insertions, or substitutions required to transform one sequence into another. In [2,3] they find the optimal edit distance and present efficient algorithms for its computation.

Another technique is to use the linear consistency test (LCT), see Handbook of Cryptography (HAC) [5] and [9]. Here the l_s clock control initialization bits are guessed and used to restore the positions the keystream bits had in $\mathbf{u}$. This gives the guess $\mathbf{u}^* = (\ldots, *, z_i, \ldots, z_j, \ldots, *, \ldots, z_k, \ldots, *, \ldots)$, where z_i, z_j, z_k are some keystream bits and the stars are the deleted bits. They now perform the LCT on $\mathbf{u}^*$, using the Gaussian algorithm on an equation set with l_u unknowns derived from $LFSR_u$ and $\mathbf{u}^*$. If the equation set is consistent, the guess is outputted as the correct initialization bits for $LFSR_s$. The Gaussian algorithm will use about $\frac{2}{3}l_u^3$ calculations and the total complexity for the attack is $O(2^{l_s} \cdot l_u^3)$.

In [1,10] and the recent paper [11] they guess only a few of the clock control bits before they reject/accept the guess, using the Gaussian algorithm. If the guessed bits pass the test, they do an exhaustive search on the remaining key space.

It is hard to estimate the running time for the attacks in [1,10,2,3]. The attack in [11] is estimated to have an upper bound complexity $O(L^3 \cdot 2^{L\lambda})$, where $\lambda = \log A/(1 + \log A)$, $L = l_s + l_u$ and A is the number of different clocking numbers from the $D()$ function.

Most of the previous LCT attacks have in common that they try to find the initialization bits for both $LFSR_s$ and $LFSR_u$ at once. We have a much more simple and algorithmic approach to the problem. The resulting algorithm is deterministic and has a lower and easily estimated running time which is independent from the number of clock control behaviors A, and the length l_u of $LFSR_u$. We will show that our attack has lower computational complexity than the previous LCT attacks.

We also do a test similar to the LCT, but our test is much more efficient since we are not using the Gaussian algorithm to reject or accept the initialization bits for $LFSR_s$. Our rejection test has constant complexity $O(K)$, where K is only

2 parity check operations in average. Thus, the total complexity for the attack becomes $O(2^{l_\mathrm{s}})$.

The basic idea for the test is simple. From the generator polynomial $g_\mathrm{u}(x)$ for $LFSR_\mathrm{u}$ we derive a low weight cyclic equation that will hold for all bitstreams generated by $LFSR_\mathrm{u}$. In Appendix C.3 we describe a modified version of Wagners general birthday algorithm [8] that finds the low weight cyclic equation. For each guess of $\mathbf{c}$ we generate the guess $\mathbf{u}^*$ for $\mathbf{u}$. Then we try the cyclic equation at a given number of entries in the $\mathbf{u}^*$ stream. If the equation hold every time, we can conclude that the bits are generated by $LFSR_\mathrm{u}$, and it is most likely that we have the correct guess for $\mathbf{c}$. If the guess is wrong we have to test the equation at in average 2 entries before the guess is rejected. A naive implementation of this algorithm will, as shown in Section 3.2, have complexity $O(2^{l_\mathrm{s}} \cdot N)$ where N is the length of the guess $\mathbf{u}^*$. The reason for this is that we have to calculate a new $\mathbf{u}^*$ for each guess for $\mathbf{c}$.

The real advantage in this paper is the new algorithm we present in Section 3.4. The algorithm is iterative and except for the first iteration it calculates each guess $\mathbf{u}^*$ using just a few operations. The idea is to go through the guesses for $\mathbf{c}$ cyclically. This way we can reuse most of $\mathbf{u}^*$ from one guess to another. In worst case our attack needs 2^{l_s} iterations to succeed, and we have the complexity $O(2^{l_\mathrm{s}})$. Thus, by using the cyclic properties of feedback shift registers, we have got rid of the l_u^3 factor they have in the LCT attacks in [5,9,1,11]. In Section 4 we present some simulations of the algorithm.

2 A General Model for Irregularly Clocked Generators

2.1 Description

We will first give a general description of irregularly clocked generators.

Let $g_\mathrm{u}(x)$ be the feedback polynomial for the shift register $LFSR_\mathrm{u}$ of length l_u, and let $g_\mathrm{s}(x)$ be the feedback polynomial for a shift register $LFSR_\mathrm{s}$ of length l_s. $LFSR_\mathrm{u}$ is called the *data generator*, and $LFSR_\mathrm{s}$ is called the *clock control generator.*

From $g_\mathrm{s}(x)$ we can calculate a clock control sequence $\mathbf{c}$ in the following way. Let $c_t = D(s_v, s_{v+1}, \dots, s_{v+l_\mathrm{s}-1}) \in \{a_1, a_2, \dots, a_A\}, a_j \geq 0$ be a function where the input $(s_v, s_{v+1}, \dots, s_{v+l_\mathrm{s}-1})$ is the inner state of $LFSR_\mathrm{s}$ after v feedback shifts and A is the number of values that c_t can take. Let p_j be the probability $p_j = \mathrm{Prob}(c_t = a_j)$. The way $LFSR_\mathrm{s}$ is clocked is defined by the specific generator. Often $LFSR_\mathrm{s}$ and c_t are synchronized, which means that $v = t$.

$LFSR_\mathrm{u}$ produces the stream $\mathbf{u} = (u_0, u_1, \dots)$ The clock c_t decides how many times $LFSR_\mathrm{u}$ is clocked before the output bit from $LFSR_\mathrm{u}$ is taken as keystream bit z_t. Thus, the keystream z_t is produced by $z_t = u_{k(t)}$, where $k(t)$ is the total sum of the clock at time t, that is $k(t) \leftarrow k(t-1) + c_t$.

Let $\mathbf{u} = (u_0, u_1, \ldots, u_{N-1})$ be the bit stream produced by the shift register $LFSR_{\mathrm{u}}$. The resulting sequence will then be $z_t = u_{k(t)}$, $1 < t < M$. This gives the following definition for the clocking of $LFSR_{\mathrm{u}}$.

Definition 1. *Given bit stream* $\mathbf{u}$ *and clock control sequence* $\mathbf{c}$*, let* $\mathbf{z} = Q(\mathbf{c}, \mathbf{u})$ *be the function that generates* $\mathbf{z}$ *of length* M *by*

$$Q(\mathbf{c}, \mathbf{u}) : \, z_t \leftarrow u_{k(t)}, \, 0 \leq t < M$$

where $k(t) = \sum_{j=0}^{t} c_j - S,\ S \in \{0, 1\}$

The parameter S only is for synchronization, and most often $S = 1$. Finally we let $\mathbf{s}^{\mathrm{I}} = (s_0, s_1, \ldots, s_{l_{\mathrm{s}}-1})$ and $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, \ldots, u_{l_{\mathrm{u}}-1})$ be the initialization states for $LFSR_{\mathrm{s}}$ and $LFSR_{\mathrm{u}}$. Together, $\mathbf{s}^{\mathrm{I}}$ and $\mathbf{u}^{\mathrm{I}}$ defines *the secret key* for the given cipher system.

If $a_j \geq 1,\ \ 1 \leq j \leq A$, the function $Q(\mathbf{c}, \mathbf{u})$ can be looked on as a deletion channel with input $\mathbf{u}$ and output $\mathbf{z}$. The deletion rate is

$$P_{\mathrm{d}} = 1 - \frac{1}{\sum_{j=1}^{A} p_j a_j}. \tag{1}$$

Thus, given a stream $\mathbf{z}$ of length M, the expected length N of the stream $\mathbf{u}$ is

$$\mathrm{E}(N) = \frac{M}{(1 - P_{\mathrm{d}})} = M \sum_{j=1}^{A} p_j a_j. \tag{2}$$

2.2 Some Examples for Clock Control Generators

The Step-1/Step-2 Generator. The clocking function is defined by $Q(\mathbf{c}, \mathbf{u}) : z_t \leftarrow u_{k(t)}, 0 \leq t < M$, and $D(s_t) = 1 + s_t$. We see that the number of outputs is $A = 2$, with probabilities $p_j = 1/2,\ 1 \leq j \leq 2$. This gives $P_{\mathrm{d}} = 1 - \frac{1}{\frac{1}{2} + 2\frac{1}{2}} = \frac{1}{3}$, and $\mathrm{E}(N) = \frac{3}{2}M$. Since this generator is simple, we will use it in the examples in this paper.

Example 1. Assume we have an irregularly clock control stream cipher as defined in Section 2.1, with $g_{\mathrm{s}}(x) = x^3 + x^2 + 1$. We let $\mathbf{s}^{\mathrm{I}} = (s_0, s_1, s_2) = (1, 0, 1)$ and we get $\mathbf{c}$ by $c_t = D(s_t)$:

$$\mathbf{c} = (2, 1, 2, 1, 1, 2, 2, 2, 1, 2, 1, 1, 2, 2, 2, 1, 2, 1, 1, 2, \ldots).$$

Let $g_{\mathrm{u}}(x) = x^4 + x^3 + 1$ and $LFSR_{\mathrm{u}}$ be initialized with $\mathbf{u}^{\mathrm{I}} = (1, 1, 0, 0)$. We get the following bit stream

$$\mathbf{u} = (1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1). \tag{3}$$

Using $\mathbf{c}$ on $\mathbf{u}$, the bits are discarded in this way,

$$\begin{aligned}\mathbf{u}^* = (&*, 1, 0, *, 1, 0, 0, *, 1, *, 1, *, 0, 1, *,\\ &1, 1, 0, *, 1, *, 0, *, 1, 1, *, 1, 0, 1)\end{aligned} \tag{4}$$

Finally the output bit from the cipher will be

$$\mathbf{z} = Q(\mathbf{c}, \mathbf{u}) = (1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1). \tag{5}$$

The LILI-128 Clock Control Generator. The clock control generator which is one of the building blocks in the LILI-128 cipher [7] is similar to the *step-1/step-2* generator but $\mathbf{c}$ has a larger range. The generator is defined by $Q(\mathbf{c}, \mathbf{u})$: $z_t \leftarrow u_{k(t)}, 0 \leq t < M$, and $c_t = D(s_{t+i_1}, s_{t+i_2}) = 1 + s_{t+i_1} + 2s_{t+i_2}$. This gives $A = 4$, $p_j = \frac{1}{4}$, $1 \leq j \leq 4$, and $P_{\mathrm{d}} = 1 - \frac{1}{\sum_{j=1}^{4} \frac{1}{4} j} = \frac{3}{5}$, and the length of $\mathbf{u}$ is expected to be $N = \frac{5}{2}M$.

The Shrinking Generator. In the shrinking generator, the output bit u_k from $LFSR_{\mathrm{u}}$ is outputted as keystream bit z_t if the output s_k from $LFSR_{\mathrm{s}}$ equals one. If $s_k = 0$ then u_k is discarded.

To be able to attack the generator with our algorithm we must have the clock control sequence $\mathbf{c}$. The clock control sequence for the shrinking generator can be generated as follows. Let $y - 1$ be the number of consecutive zeros from $s_v = 1$, that is $(s_v, s_{v+1}, \ldots, s_{v+l_{\mathrm{s}}-1}) = (\underbrace{1, 0, \ldots, 0}_{y}, *, \ldots, *)$. Then the clocking function is defined as $D(s_v, s_{v+1}, \ldots, s_{v+l_{\mathrm{s}}-1}) = y$. It follows from the definition of the shrinking generator that $LFSR_{\mathrm{s}}$ and $LFSR_{\mathrm{u}}$ are synchronized, so $LFSR_{\mathrm{s}}$ must be clocked c_t times before the next bit is outputted. Thus, the clock control sequence is $c_t = D(s_{k(t-1)}, s_{k(t-1)+1}, \ldots, s_{k(t-1)+l_{\mathrm{s}}-1})$, where $k(t) \leftarrow k(t-1) + c_t$ for each iteration and $k(-1) = 0$. $Q(\mathbf{c}, \mathbf{u})$ is the same as for the generators above. If we analyze the clock control sequence, $c_t \in \{1, 2, \ldots, \ldots, l_{\mathrm{s}} - 1\}$, where $p_j = 1/2^j$, when l_{s} is a large number ($l_{\mathrm{s}} > 10$). This gives $A = l_{\mathrm{s}} - 1$, $P_{\mathrm{d}} = 1 - \frac{1}{\sum_{j=1}^{l_{\mathrm{s}}} \frac{1}{2^j} j} \approx 0.5$ and $\mathrm{E}(N) = 2M$, as intuitively expected.

3 A New Attack on Irregularly Clocked Generators

The idea behind the attack is to guess the clock control sequence $\mathbf{c}$, and reconstruct the original positions the keystream bits in $\mathbf{z}$ had in $\mathbf{u}$ using the reversed function $Q^*(\mathbf{c}, \mathbf{z})$ defined below. From this we get a sequence $\hat{\mathbf{u}}$ looking similar to (4). When this is done, we test if $\hat{\mathbf{u}}$ is a sequence that could have be generated by $LFSR_{\mathrm{u}}$ using some linear equations we know hold over any sequences generated by $LFSR_{\mathrm{u}}$. If the test holds, we assume we have made the correct guess for $\mathbf{c}$. Knowing the correct $\mathbf{c}$, we can use the Gaussian algorithm as described in [9] to find the initialization bits for $\mathbf{u}$.

3.1 The Basics

First we state a definition.

Definition 2. *Given the clock control sequence* $\mathbf{c}$ *and keystream* $\mathbf{z}$*, let the function* $\mathbf{u}^* = Q^*(\mathbf{c}, \mathbf{z})$ *be the (not complete) reverse of* Q*, defined as*

$$Q^*(\mathbf{c}, \mathbf{z}) : u^*_{k(t)} \leftarrow z_t, \; 0 \leq t < M,$$

where $k(t) = \sum_{j=0}^{t} c_j - S$*, and* $u^*_k = *$ *for the entries* k *in* $\mathbf{u}^*$ *where* u^*_k *is deleted. When this occurs we say that* u^*_k *is not defined.*

The length of $\mathbf{u}^*$ will be $N^* = \sum_{j=0}^{M-1} c_j$. Note that the only difference between this definition and Definition 1, is that $\mathbf{u}$ and $\mathbf{z}$ have changed sides. Thus, $Q^*(\mathbf{c}, \mathbf{z})$ is a reverse of the $Q(\mathbf{c}, \mathbf{u})$. But since some bits are deleted, the reverse is not complete and we get the stream $\mathbf{u}^*$. As seen in Example 1, we can reverse the keystream (5) back to (4) but not completely back to the original stream (3), since the deleted bits are not known.

The probability for a bit u^*_k being defined is $\text{Prob}(u^*_k) = 1 - P_\text{d}$. This happens when $k = k(t)$ holds for for some t, $0 \leq t < M$. It follows that the sum $\delta = u^*_k + u^*_{k+j_1} + \cdots + u^*_{k+j_{w-1}}$will be defined if and only if all of the bits in the sum are defined. Thus, the sum δ will be defined for given k in $\mathbf{u}^*$ with probability

$$P_\text{def} = \text{Prob}(u^*_k, u^*_{k+j_1}, \ldots, u^*_{k+j_{w-1}}) = (1 - P_\text{d})^w = \left(\frac{1}{\sum_{j=1}^{A} p_j a_j} \right)^w . \qquad (6)$$

3.2 Naive Attack

Using definition 2, we first present a naive high complexity attack. In the next section we present a more advanced and low complexity version of the attack.

Let $\mathbf{s}^\text{I}$ be the initial state for $LFSR_\text{s}$, and let $L^v(\mathbf{s}^\text{I})$ be the inner state after v feedback shifts. Without loss of generality we assume $S = 1$ and that $LFSR_\text{s}$ is clocked once for each output c_t. Thus, $v = t$ and $c_t = D(L^t(\mathbf{s}^\text{I}))$ is the output from the clock generator after t feedback shifts.

We are given a keystream $\mathbf{z}$ of length M which is generated with $\mathbf{z} = Q(\mathbf{c}, \mathbf{u})$. Assume we have found an equation $u_k + u_{k+j_1} + \cdots + u_{k+j_{w-1}} = 0$ that holds over $\mathbf{u}$. First we guess the initial state $LFSR_\text{s}$ and generates the corresponding guess $\hat{\mathbf{c}}$ for $\mathbf{c}$ using the $D()$ function. Using definition 2 we can calculate $\mathbf{u}^* = Q^*(\hat{\mathbf{c}}, \mathbf{z})$. Then we try to find m (typically $m = l_\text{s} + 10$, we add 10 to prevent false alarms) entries in $\mathbf{u}^*$ where the equation is defined. If the equation holds for every entry it is defined, we assume we have found the correct guess for $\mathbf{s}^\text{I}$. If not, we make a new guess and do the test again. The pseudo code for this algorithm is given below.

Input The keystream $\mathbf{z}$ of length M

1. Preprocessing: Find an equation of low weight that holds over the stream $\mathbf{u}$ of length N.
2. For all possible guesses $\hat{\mathbf{s}}^{\mathrm{I}}$ do the following:
3. Generate the clock control sequence $\hat{\mathbf{c}}$ of length M by $c_t = D(L^t(\hat{\mathbf{s}}^{\mathrm{I}}))$.
4. Generate $\hat{\mathbf{u}}^*$ of average length $N = \frac{M}{(1-P_{\mathrm{d}})}$ using $\hat{\mathbf{u}}^* = Q^*(\hat{\mathbf{c}}, \mathbf{z})$.
5. Find m entries $(k_1, k_2, \ldots, k_m)$ in the stream $\hat{\mathbf{u}}^*$ where the equation is defined.
6. If the equation holds for all the m entries over $\hat{\mathbf{u}}^*$, then stop the search and output the guess $\hat{\mathbf{s}}^{\mathrm{I}}$ as the key for $LFSR_{\mathrm{s}}$.

The problem with this algorithm is that for each guess for $\hat{\mathbf{s}}^{\mathrm{I}}$, we have to generate a new clock control stream of length M and generate $\hat{\mathbf{u}}^* = Q^*(\hat{\mathbf{c}}, \mathbf{z})$ of length N. In larger examples, N and M will be large numbers, say around 10^6. Since the complexity is $O(N \cdot 2^{l_{\mathrm{s}}})$, the run time for this algorithm will in many cases be worse than the algorithm in [9]. In the next session we present an idea that fixes this problem.

3.3 Final Idea

The problem in the previous section was that we had to generate M bits of the clock control stream for each guess for $\mathbf{s}^{\mathrm{I}}$. This can be avoided if we go through the guesses in a more natural way. We start by an initial guess $\hat{\mathbf{s}}^{\mathrm{I}} = (0, 0, \ldots, 1)$, and let the i'th guess be the internal state of the $LFSR_{\mathrm{s}}$ after i feedback shifts.

Let $\mathbf{c}^i = (c_0^i, c_1^i, \ldots, c_{M-1}^i)$ be the i'th guess for the clock control sequence defined by $c_t^i = D(L^{i+t}(1, 0, \ldots, 0))$, $0 \le t < M$. Let $\mathbf{u}^i = Q^*(\mathbf{c}^i, \mathbf{z})$ be the corresponding guess for $\mathbf{u}^*$ of length $N_i = \sum_{t=0}^{M-1} c_t^i$. We can now give an iterative method for generating $\mathbf{u}^{i+1}$ from $\mathbf{u}^i$.

Lemma 1. *We can transform $\mathbf{u}^i$ into $\mathbf{u}^{i+1} = Q^*(\mathbf{c}^{i+1}, \mathbf{z})$ using the following method: Delete the first c_0^i entries $(*, \ldots, *, z_0)$ in $\mathbf{u}^i$, append the $c_{M-1}^{i+1} = c_M^i$ entries $(*, \ldots, *, z_M)$ at the end, and replace z_t with z_{t-1} for $1 \le t \le M$.*

Proof. See Appendix B.

Lemma 1 gives us a fast method for generating all possible guesses for $\mathbf{u}$ given a keystream $\mathbf{z}$. See Table 1 for an intuitive example of how the lemma works.
Next we prove a theorem that allows us to reuse the equation set defined for $\mathbf{u}^i$.

Theorem 1. *If the sum*

$$\beta_{\mathbf{u}^i,k} = u_k + u_{k+k_1} + \cdots + u_{k+k_{w-1}} = z_t + z_{t+j_1} + \cdots + z_{t+j_{w-1}} = \gamma_{\mathbf{z},t}$$

is defined over $\mathbf{u}^i$, then the sum

$$\beta_{\mathbf{u}^{i+1},k-c_0^i} = z_{t-1} + z_{t+j_1-1} + \cdots + z_{t+j_{w-1}-1} = \gamma_{\mathbf{z},t-1}$$

is defined over $\mathbf{u}^{i+1}$.

Proof. See Appendix B.

The main result from this theorem is that the equation set that is defined over $\mathbf{u}^i$ will still be defined over $\mathbf{u}^{i+1}$ if we shift the equations c_0^i entries to the left over $\mathbf{u}^{i+1}$. This means that we can just shift the equations 1 entry to the left over $\mathbf{z}$, and we will have an sum that is defined for the guess $\hat{\mathbf{s}}^{\mathrm{I}} = D(L^{i+1}(1, 0, \ldots, 0)$. Thus, the theorem indicates that we can go around a lot of computations if we let the i'th guess for the inner state of $LFSR_{\mathrm{s}}$ be $L^i(1, 0, \ldots, 0)$.

Table 1. Example of a walk through of the key. The bits in bold font show how the pattern of defined bits in $\mathbf{u}^i$ shifts to the left, while the actual key bits stay relatively put. Also notice how the entries z_i in the patterns are replaced with z_{i-1} after one iteration. For example the sub stream $z_7, z_8, z_9, *, z_{10} \rightarrow z_6, z_7, z_8, *, z_9$. This means that if the sum $z_7 + z_8 + z_9 + z_{10}$ is defined for $\mathbf{c}^i$, then $z_6 + z_7 + z_8 + z_9$ will be defined for $\mathbf{c}^{i+1}$

Guessed clock sequence $\mathbf{c}^i$	Resulting 'known' bits of $\mathbf{u}^i = Q^*(\mathbf{c}^i, \mathbf{z})$.
$(2, 1, 1, 2, 2, 2, 1, 2, 1, 1, 2)$	$(*, z_0, z_1, z_2, *, z_3, *, z_4, *, z_5, z_6, *, \mathbf{z_7}, \mathbf{z_8}, \mathbf{z_9}, *, \mathbf{z_{10}})$
$(1, 1, 2, 2, 2, 1, 2, 1, 1, 2, 2)$	$(z_0, z_1, *, z_2, *, z_3, *, z_4, z_5, *, \mathbf{z_6}, \mathbf{z_7}, \mathbf{z_8}, *, \mathbf{z_9}, *, z_{10})$
$(1, 2, 2, 2, 1, 2, 1, 1, 2, 2, 2)$	$(z_0, *, z_1, *z_2, *, z_3, z_4, *, \mathbf{z_5}, \mathbf{z_6}, \mathbf{z_7}, *, \mathbf{z_8}, *, z_9, *, z_{10})$
$(2, 2, 2, 1, 2, 1, 1, 2, 2, 2, 1)$	$(*, z_0, *, z_1, *z_2, z_3, *, \mathbf{z_4}, \mathbf{z_5}, \mathbf{z_6}, *, \mathbf{z_7}, *, z_8, *, z_9, z_{10})$
$(2, 2, 1, 2, 1, 1, 2, 2, 2, 1, 2)$	$(*, z_0, *z_1, z_2, *, \mathbf{z_3}, \mathbf{z_4}, \mathbf{z_5}, *, \mathbf{z_6}, *, z_7, *, z_8, z_9, *, z_{10})$

3.4 The Complete Attack

We will now present a new algorithm that makes use of the observations above. We start by analyzing $LFSR_{\mathrm{u}}$ (See Appendix C.3) to find an equation λ

$$\lambda : u_k + u_{k+j_1} + \cdots + u_{k+j_{w-1}} = 0$$

that holds over all $\mathbf{u}$ generated by $LFSR_{\mathrm{u}}$ for any $k \geq 0$. Let the first guess for the initialization state for $\mathbf{s}$ be $\hat{\mathbf{s}}^{\mathrm{I}} = (1, 0, 0, \ldots, 0)$, generate $\mathbf{c}^0$ by $c_t^0 = D(L^t(1, 0, \ldots, 0))$, $t < M$, and $\mathbf{u}^0 = Q^*(\mathbf{c}, \mathbf{z})$. Next we try to find m places $(k_1, k_2, \ldots, k_m)$ in $\mathbf{u}^0$ where the equation λ is defined. From this we get the equation set

$$\begin{aligned}
u_{k_1}^0 + u_{k_1+j_1}^0 + \cdots + u_{k_1+j_{w-1}}^0 &= 0 \\
u_{k_2}^0 + u_{k_2+j_1}^0 + \cdots + u_{k_2+j_{w-1}}^0 &= 0 \\
\vdots \qquad\qquad & \quad \iddots \\
u_{k_m}^0 + u_{k_m+j_1}^0 + \cdots + u_{k_m+j_{w-1}}^0 &= 0
\end{aligned}$$

Since every $u_{k_x+j_y}$ in this equation set is defined in $\mathbf{u}^0$, we can replace $u_{k_x+j_y}$ with the corresponding bit z_t in the keystream $\mathbf{z}$. Thus, $\mathbf{u}^0$ is a sequence of pointers to $\mathbf{z}$ and we can write the equations over $\mathbf{z}$ as the equation set Ω :

$$\begin{aligned} z_{t_{1,1}} + z_{t_{1,2}} + \cdots + z_{t_{1,w}} &= 0 \\ z_{t_{2,1}} + z_{t_{2,2}} + \cdots + z_{t_{2,w}} &= 0 \\ \vdots \qquad\qquad & \ \vdots \\ z_{t_{m,1}} + z_{t_{m,2}} + \cdots + z_{t_{m,w}} &= 0 \end{aligned} . \tag{7}$$

We are now finished with the precomputation.

Next, we test the equation set to see if all the equations hold. If not, we iterate using the algorithm below which outputs the correct $\mathbf{s}^{\mathrm{I}}$. Knowing $\mathbf{s}^{\mathrm{I}}$ it is easy to calculate $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, \ldots, u_{l_{\mathrm{u}}-1})$ using the Gaussian algorithm once on an equation set derived from $\mathbf{s}^{\mathrm{I}}$ and $LFSR_{\mathrm{u}}$.

Input The keystream $\mathbf{z}$ of length M, the equation λ, the equation set Ω, the pointer sequence $\mathbf{u}^0$, the states $L^0(1,0,\ldots,0)$ and $L^M(1,0,\ldots,0)$, Set $i \leftarrow 0$

1. Calculate $c_0^i = D(L^i(1,0,\ldots,0))$, and $c_{M-1}^{i+1} = c_M^i = D(L^{M+i}(1,0,\ldots,0))$.
2. Use lemma 1 to generate $\mathbf{u}^{i+1} = Q^*(\mathbf{c}^{i+1}, \mathbf{z})$ and lower all indexes in the equation set Ω by one. Theorem 1 guarantees that the equations are defined over $\mathbf{u}^{i+1}$.
3. If the first equation in Ω gets a negative index, then remove the equation from Ω. Find a new index at the end of $\mathbf{u}^{i+1}$ where λ is defined, and add the new equation over $\mathbf{z}$ to Ω.
4. If the current equation set Ω holds, stop the algorithm and output $\mathbf{s}^{\mathrm{I}} = L^{i+1}(1,0,\ldots,0)$ as the initialization state for $LFSR_{\mathrm{s}}$.
5. If δ does not hold, we set $i \leftarrow i+1$ and go to step 1.

Note 1. To reach the desired complexity ($2^{l_{\mathrm{s}}}$) a few details on the implementation of the algorithm are needed. These details are given in Appendix A.

All changes during the iterations are done on $\mathbf{u}^i$ and the equation set Ω. Thus, each guess $L^i(1,0,\ldots,0)$ for $\mathbf{s}^{\mathrm{I}}$ result in an unique equation set Ω. The $\mathbf{z}$ stream is never altered.

Example 2. We continue on the generator in Example 1. We have found the equation $u_k + u_{k+6} + u_{k+8} = 0$, which corresponds to the multiple $h(x) = 1 + x^6 + x^8$. We have $\mathbf{z}$ of length 19, and want to find $\mathbf{s}^{\mathrm{I}}$. The length of $\mathbf{u}^i$ will be $N \approx \frac{3}{2}19 = 28.5$. We set the first guess to $\mathbf{s}_0^{\mathrm{I}} = (1,0,0)$. From this we generate the clock control sequence using the function $c_t^0 = D(L^t(1,0,0))$, $0 \le t \le M-1$, and we get

$$\mathbf{c}^0 = (2,1,1,2,2,2,1,2,1,1,1,2,2,2,1,2,1,1,2,2,2).$$

Then we spread out the $\mathbf{z}$ stream corresponding to $\mathbf{c}$, that is $\mathbf{u}^0 = Q^*(\mathbf{c}^0, \mathbf{z})$. From this we get the sequence

$$\begin{aligned}\mathbf{u}^0 = (&*, z_0, z_1, z_2, *, z_3, *, z_4, *, z_5, z_6, *, z_7, z_8, z_9, *, z_{10},\\ &*, z_{11}, *, z_{12}, z_{13}, *, z_{14}, z_{15}, z_{16}, *, z_{17}, *, z_{18}).\end{aligned}$$

We search through $\mathbf{u}^0$ to find 4 entries where the equation $u_k + u_{k+6} + u_{k+8} = 0$ is defined. Since all the defined entries in $\mathbf{u}^0$ points to bits in the $\mathbf{z}$ stream, we get the following set of equations Ω over $\mathbf{z}$:

$$\begin{aligned} z_0 + z_4 + z_5 &= 0\\ z_6 + z_{10} + z_{11} &= 0\\ z_7 + z_{11} + z_{12} &= 0\\ z_{13} + z_{17} + z_{18} &= 0\end{aligned}$$

We test the equations to see if all the equations hold. If the set does not hold, we continue as follows. We shift the $LFSR_\text{s}$ once, and it will have $\mathbf{s}_1^\text{I} = L^1(1,0,0) = (0,0,1)$ as inner state. We calculate $c_{M-1}^1 = c_M^0 = D(L^M(1,0,0))$. Then we use Lemma 1 to calculate $\mathbf{u}^1$ from $\mathbf{u}^0$. That is, we delete the $c_0^0 = 2$ entries $(*, z_0)$, append the $(c_{18}^1 = 2)$ entries $(*, z_{19})$ at the end, and at last replace the pointer z_t with z_{t-1} for $1 \leq t \leq M$. We get this guess for $\mathbf{u}$:

$$\begin{aligned}\mathbf{u}^1 = (&z_0, z_1, *, z_2, *, z_3, *, z_4, z_5, *, z_6, z_7, z_8, *, z_9,\\ &*, z_{10}, *, z_{11}, z_{12}, *, z_{13}, z_{14}, z_{15}, *, z_{16}, *, z_{17}, *, z_{18}).\end{aligned}$$

If an equation is defined for the entry t in $\mathbf{z}$ for the guess $\mathbf{s}_0^\text{I}$, it will now be defined for the entry $t-1$ in $\mathbf{z}$ for the guess $\mathbf{s}_1^\text{I}$ as guaranteed by Theorem 1. From this Ω becomes:

$$\begin{aligned} z_{-1} + z_3 + z_4 &= 0\\ z_5 + z_9 + z_{10} &= 0\\ z_6 + z_{10} + z_{11} &= 0\\ z_{12} + z_{16} + z_{17} &= 0\end{aligned}$$

We remove the first equation from Ω since it has a negative index, and find a new index at the end of $\mathbf{u}^1$ where λ is defined. We find the equation $z_{13} + z_{17} + z_{18} = 0$ and add it to Ω. We test the equations to see if all the equations hold. If the set does not hold, we continue the algorithm.

3.5 Complexity and Properties

Precomputation If the generator polynomial $g_{\mathrm{u}}(x)$ for $LFSR_{\mathrm{u}}$ has sufficient low weight, say ≤ 10, we can use it directly in our algorithm with $w = \mathrm{weight}(g_{\mathrm{u}})$ and $h(x) = g_{\mathrm{u}}(x)$. In such a case we do not need much precomputation. The only precomputation is to generate $\mathbf{u}^0$ of length N, where the length of N is calculated below.

If $g_{\mathrm{u}}(x)$ has too high weight we use a modified version of Wagners algorithm for the generalized birthday problem [8] to find a multiple $h(x)$=$a(x)g(x)$ of weight $w = 2^r$ and degree l_{h}. The multiple $h(x)$ gives a new recursion of low weight. The fast search algorithm is described in Appendix C.3. See Table 2 for some multiples found by the algorithm.

Table 2. The table shows some weight 4 multiples of different polynomials found using the algorithm in Appendix C.3. The algorithm used 1 hour and 15 minutes to find the multiple of the degree 80 polynomial, mostly due to heavy use of hard disk memory. The search for the multiple of the degree 60 polynomial took 14 seconds

$g(x)$	$h(x) = a(x)g(x)$
$x^{40} + x^{38} + x^{35} + x^{32} + x^{28} + x^{26} + x^{22} + x^{20} + x^{17} + x^{16} + x^{14} + x^{13} + x^{11} + x^{10} + x^{9} + x^{8} + x^{6} + x^{5} + x^{4} + x^{3} + 1$	$x^{24275} + x^{6116} + x^{1752} + 1$
$x^{60} + x^{58} + x^{56} + x^{52} + x^{51} + x^{50} + x^{49} + x^{48} + x^{47} + x^{46} + x^{44} + x^{41} + x^{40} + x^{39} + x^{36} + x^{29} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{21} + x^{20} + x^{19} + x^{16} + x^{15} + x^{11} + x^{10} + x^{9} + x^{4} + x^{2} + x + 1$	$x^{2464041} + x^{1580916} + x^{131400} + 1$
$x^{80} + x^{79} + x^{78} + x^{76} + x^{75} + x^{69} + x^{68} + x^{57} + x^{56} + x^{55} + x^{54} + x^{52} + x^{49} + x^{46} + x^{45} + x^{44} + x^{42} + x^{37} + x^{36} + x^{35} + x^{32} + x^{31} + x^{30} + x^{28} + x^{27} + x^{26} + x^{24} + x^{23} + x^{21} + x^{20} + x^{19} + x^{13} + x^{12} + x^{10} + x^{8} + x^{6} + x^{4} + x^{3} + 1$	$x^{312578783} + x^{309946371} + x^{210261449} + 1$

When we have found a polynomial $h(x) = 1 + x^{j_1} + \cdots + x^{j_{w-1}}$ with $j_{w-1} = l_{\mathrm{h}}$, the corresponding equation λ over $\mathbf{u}$ is $u_k + u_{k+j_1} + \cdots + u_{k+j_{w-1}} = 0$. We want to find m places in the stream $\mathbf{u}$ where λ is defined. From equation (6) we have that an equation of weight w is defined at an random entry in $\mathbf{u}$ with a probability $P_{\mathrm{def}} = (1 - P_{\mathrm{d}})^w$. Thus, we must test around $m/(1 - P_{\mathrm{d}})^w$ entries to find m equations over $\mathbf{z}$. To be able to do this $\mathbf{u}$ must have length

$$N > l_{\mathrm{h}} + \frac{m}{(1 - P_{\mathrm{d}})^w}. \tag{8}$$

To avoid false keys, we choose $m > l_s$. From the expectation (2) of N we have $E(M) = N(1 - P_{\mathrm{d}}) = (1 - P_{\mathrm{d}})l_{\mathrm{h}} + \frac{m}{(1-P_{\mathrm{d}})^{w-1}}$, and we have proved the following proposition:

Proposition 1. *Let an equation over* $\mathbf{u}$ *be defined by* $h(x)$ *of weight* w *and degree* l_{h}. *To get an equation set* Ω *of* $m > l_s$ *equations over* $\mathbf{z}$, *the length of the* $\mathbf{z}$ *stream must be*

$$M > (1 - P_{\mathrm{d}})l_{\mathrm{h}} + \frac{m}{(1 - P_{\mathrm{d}})^{w-1}}. \tag{9}$$

where $m \approx l_{\mathrm{s}} + 10$.

We see that the keystream length M is dependent of the degree l_{h} of $h(x)$ of weight $w = 2^r$. The degree l_{h} is then again highly dependent on the search algorithm we use to find $h(x)$. When we use the search algorithm in Appendix C.3 with the proposed parameters we show in the appendix that l_{h} will be in order of $l_{\mathrm{u}} \approx T_{\mathrm{mem}}(l_{\mathrm{u}}, r) = 2^{\frac{r+l}{r+1}}$.

Decoding. If this algorithm is implemented properly (Appendix A) it will have worst case complexity $O(2^{l_{\mathrm{s}}})$ with a very little constant factor. In average the number of iterations will be in the order of $2^{l_{\mathrm{s}}-1}$.

At each iteration i we shift the sliding window c_0^i to the right over $\mathbf{u}^i$. Then we shift the equation set 1 to the left over $\mathbf{z}$, and test it. If we have the wrong guess for $\mathbf{s}^{\mathrm{I}}$, each equation in the set will hold with a probability $\frac{1}{2}$. When we reach an equation that does not hold we know that the guess for $\mathbf{s}^{\mathrm{I}}$ is wrong and we break off the test. Thus, the average number of equations we have to evaluate per guess is $\frac{\lim_{m\to\infty}\sum_{j=1}^{m} j\cdot 2^{l_{\mathrm{s}}}/2^j}{2^{l_{\mathrm{s}}}} = \lim_{m\to\infty}\sum_{i=1}^{m} i/2^i = 2$. This gives an average constant factor of 2 parity check tests for each of the $2^{l_{\mathrm{s}}}$ guesses. Thus, the complexity is $O(2 \cdot 2^{l_{\mathrm{s}}}) = O(2^{l_{\mathrm{s}}})$

Each time an equation gets a negative index, we must delete it and search for a new equation at the end of $\mathbf{u}^i$. We expect to search through $1/(1-p_{\mathrm{d}})^{w-1}$ entries in $\mathbf{u}^i$ to find a new equation. This is done every $\frac{M - l_{\mathrm{u}}(1-p_{\mathrm{d}})}{m}$ iteration in average, and will have little impact on the decoding complexity.

When we after i iterations have found the initialization bits for $LFSR_{\mathrm{s}}$, we use the Gaussian algorithm on the linear equation set derived from $LFSR_{\mathrm{u}}$ and $\mathbf{u}^i$ to find the initialization bits for $LFSR_{\mathrm{u}}$. This has complexity $O(l_{\mathrm{u}}^3)$ and will have little effect on the overall complexity of the algorithm.

4 Simulations

We have done the attack on 4 small cipher systems, defined with clock control generator polynomials of degree 25 and 26, and the data generator polynomials of degree 40 and 60 from Table 2. The clock function $D()$ is the LILI-clock function as described in Section 2.2. Note that we only attack the irregularly clocking building block in LILI and not the complete LILI-128 cipher. In LILI-128 the stream is filtered through a Boolean function, and this is beyond the scope of this paper.

Degree l_s of $g_\mathrm{s}(x)$	Degree l_u of $g_\mathrm{u}(x)$	Degree l_h of $h(x)$	Number of Iterations	Decoding time	Length M of $\mathbf{z}$
25	40	24275	2^{25}	9 sec.	10000
26	40	24275	2^{26}	18 sec	10000
25	60	2464041	2^{25}	9 sec	1000000
26	60	2464041	2^{26}	18 sec	1000000

Table 4. The attacks are done in C code on a 2.2 GHz Pentium IV running under Linux. Note how the running time is exactly the same for $l_\mathrm{u} = 40$ and $l_\mathrm{u} = 60$. We have set the number of equations to $m = 35$. The polynomials $g(x)$ and $h(x)$ are from Table 2

We have used Proposition 1 and Equation (8) to calculate the length M of $\mathbf{z}$ and length N of $\mathbf{u}$ (rounded up to nearest thousand and hundred thousand). The number of parity check equations over $\mathbf{z}$ is set to $m = 35 \approx l_s + 10$. Recall that the number of parity check equations does not effect the complexity. Table 4 shows how the running time of the attack is unchanged when the degree of $g_\mathrm{u}(x)$ gets larger. The impact from a larger l_u is that we need longer keystream.

Normally we would stop the search when we have found the correct key. But then the running time would be highly dependent on where the key is in the key space. To avoid this we have gone trough the whole key space to be able to compare the different attacks in the table. In a real attack the average running time would be half the running times in Table 4. To compare with previous LCT attacks, the Gaussian factor $\frac{2}{3}l_\mathrm{u}^3$ would be around 144000 for $l_\mathrm{u} = 60$, and around 42666 for $l_\mathrm{s} = 40$. In our attack the constant factor is only 2 in average. Thus, the same attacks presented in Table 4 would take several hours or even days using the previous LCT algorithm.

5 Conclusion

We have presented a new linear consistency attack with lower complexity than previous on a general model for irregularly clocked stream ciphers. We have tested the attack in software and confirmed that the attack has a very low running time that follows the expected complexity $O(2^{l_s})$. Thus, the run time complexity is independent of the degree l_u of $LFSR_u$.

Further on, if we modify the algorithm, it will work on systems where noise is added on keystream $\mathbf{z}$. Using much higher m and giving each guess $\mathbf{s}^I$ a metric, we can perform an correlation attack with complexity $O(m \cdot 2^{l_s})$ on such systems. Initial tests seem very promising and we will come back to this matter in future work.

Acknowledgment

I would like to thank my supervisor prof. Tor Helleseth and Dr. Matthew Parker for helpful discussions and for reading and helping me in improving this paper.

References

1. Jovan Dj. Golic. Cryptanalysis of three mutually clock-controlled stop/go shift registers. *IEEE Transactions in information Science*, 46(3):525–533, 2000.
2. Jovan Dj. Golic and Miodrag J. Mihaljevic. A generalized correlation attack on a class of stream ciphers based on the levenshtein distance. *Journal of Cryptology*, 3:201–212, 1991.
3. Jovan Dj. Golic and Slobodan V. Petrovic. A generalized correlation attack with a probabilistic constrained edit distance. In *Advances in Cryptology - EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 472–476, 1993.
4. Thomas Johansson and Fredrik Jönsson. Fast correlation attacks on stream ciphers via convolutional codes. In *Advances in Cryptology-EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1999.
5. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
6. Håvard Molland, John Erik Mathiassen, and Tor Helleseth. Improved fast correlation attack using low rate codes. In *Cryptography and Coding, IMA 2003*, volume 2898 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2003.
7. L. Simpson, E. Dawson, J. Golic, and W. Millan. LILI keystream generator. In *SAC'2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 231–236. Springer-Verlag, 2002. Available at http://www.isrc.qut.edu.au/lili.
8. David Wagner. A generalized birthday problem. In *Advances in cryptology-CRYPTO' 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303, 2002.
9. K. Zeng, C. Yang, and Y. Rao. On the linear consistency test (LCT) in cryptanalysis with applications. In *Advances in Cryptology-CRYPTO '89*, number 435 in Lecture Notes in Computer Science, pages 164–174. Springer-Verlag, 1990.
10. E. Zenner, M. Krause, and S. Lucks. Improved cryptanalysis of the self-shrinking generator. In *ACISP '01*, volume 2119 of *Lecture Notes in Computer Science*, pages 21–35, 2001.
11. Erik Zenner. On the efficiency of the clock control guessing attack. In *ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, 2002.

Appendix

A Implementation Details

To reach the desired complexity $O(2^{l_s})$, the implementation of the algorithm needs some tricky details:

1. In Lemma 1 we get $\mathbf{u}^{i+1}$ by among other things deleting the c_0^i first bits of u^i. This is done using the sliding window technique, which means that we move the viewing to the right instead of shifting the whole sequence to the left. This way the shifting can be done in just one operation. To avoid heavy use of memory, we slide the window over an array of fixed length N, so that the entries that become free at the beginning of the array are reused. Thus, the left and right of the sliding window after i iterations will be

$$(left, right) = (i \bmod N, i + N_i \bmod N),$$

where $N > N_i$, for all i, $0 \leq i < 2^{l_s}$
2. In lemma 1 every reference z_{t+1} in $\mathbf{u}$ is replaced with z_t for every $0 \leq t \leq M$, which would take M operations. If we skip the replacements we note that after i iterations the entry z_t in $\mathbf{u}$ will become z_{t+i}. It is also important to notice that when we write $\mathbf{u} = (\ldots, z_0, \ldots, z_t, \ldots, z_M, \ldots)$, the entries $z_0, \ldots, z_t, \ldots, z_M$ are pointers from $\mathbf{u}$ to $\mathbf{z}$. They are not the actual key bits. Thus, in the implementation we do not replace z_t with z_{t-1}. However, when we after i iterations in the search for equations find an equation $u_k^i + u_{k+j_1}^i + \cdots + u_{k+j_{w-1}}^i = 0$ that is defined, we replace the corresponding $z_{t_1} + z_{t_2} + \cdots + z_{t_w}$ with $z_{t_1-i} + z_{t_2-i} + \cdots + z_{t_w-i}$, to compensate.
3. We do not have to keep the whole clock control sequence $\mathbf{c}^i$ in memory. We only need the two clocks, c_0^i and c_{M-1}^{i+1}, since they are used by lemma 1 to generate $\mathbf{u}^{i+1}$.

B Proofs of Lemma 1 and Theorem 1

B.1 Proof of Lemma 1

Proof. Let $\mathbf{c}^i$ be the clocking integer sequence for a given i, $0 \leq i < 2^{l_s}$. We see that $c_t^{i+1} = c_{t+1}^i$, $0 \leq t < M-1$, which means that pattern of the defined bits in $\mathbf{u}^{i+1}$ are the same as the pattern in $\mathbf{u}^i$ shifted c_0^i to the left. From this we deduce the following for given $1 \leq t \leq M$ and $k = \sum_{j=0}^{t-1} c_j^i - 1$: If $u_k^i = z_t$ for given k then $u_{k-c_0^i}^{i+1} = z_{t-1}$. If we delete the first c_0^i bits of $\mathbf{u}^i$ and get $\mathbf{u}'$ we will have that if $u'_k = z_t$ for given k, then $u_k^{i+1} = z_{t-1}$ for $k = \sum_{j=0}^{t-1} c_j^{i+1} - 1$. If we now replace every z_t in $\mathbf{u}'$ with z_{t-1} for $0 < t < M$ and get $\mathbf{u}''$ we see that $u''_k = u_k^{i+1}$, $0 \leq k < N_i - c_0^i$. To finally transform $\mathbf{u}''$ into $\mathbf{u}^{i+1}$ we just have to append the c_{M-1}^{i+1} entries $(*, \ldots, z_{M-1})$ at the end of $\mathbf{u}''$.

B.2 Proof of Theorem 1

Proof. Let

$$\mathbf{u}^i = (\ldots, \underbrace{z_0}_{c_0^i-1}, \ldots, *, \ldots, \underbrace{z_t}_{k}, \ldots, \underbrace{z_{t+j_1}}_{k+k_1}, \ldots, *, \ldots, \underbrace{z_{t+j_{w-1}}}_{k+k_{w-1}}, \ldots, \underbrace{z_{M-1}}_{N_i-1})$$

be the stream of length N_i we get using $\mathbf{u}^i = Q^*(\mathbf{c}^i, \mathbf{z})$. The notation means that $u^i_{c^i_0-1} = z_0$, $u^i_k = z_t$, and $u^i_{N_i-1} = z_{M-1}$. We see that the sum $\beta_{\mathbf{u}_i,k} = u^i_k + u^i_{k+k_1} + \cdots + u^i_{k+k_{w-1}}$ is defined over $\mathbf{u}^i$. The corresponding sum over $\mathbf{z}$ will be $\gamma_{\mathbf{z},t} = z_t + z_{t+j_1} + \cdots + z_{t+j_{w-1}}$. Then the clock control sequence we get from $c^{i+1}_t = L^{i+1+t}(1, 0, \ldots, 0)$ will be

$$\mathbf{c}^{i+1} = (c^i_1, \ldots, c^i_{M-1}, c^{i+1}_{M-1}) = (c^{i+1}_0, \ldots, c^{i+1}_{M-1}).$$

The main observation here is the following: We transform $\mathbf{u}^i$ into $\mathbf{u}^{i+1}$ by deleting the first c^i_0 entries $\underbrace{(*, \ldots, z_0)}_{c^i_0}$ in $\mathbf{u}^i$, appending $\underbrace{(*, \ldots, *, z_M)}_{c^{i+1}_M}$ at the end, and then replacing z_t with z_{t-1} for $1 \leq t \leq M$, as explained in lemma 1. From this we get the sequence

$$\mathbf{u}^{i+1} = (\ldots, \underbrace{z_0}_{c^{i+1}_0-1}, \ldots, *, \ldots, \underbrace{z_{t-1}}_{k-c^i_0}, \ldots, \underbrace{z_{t+j_1-1}}_{k+k_1-c^i_0}, \ldots, *, \ldots, \underbrace{z_{t+j_{w-1}-1}}_{k+k_{w-1}-c^i_0}, \ldots, \underbrace{z_{M-1}}_{N_{i+1}-1}). \tag{10}$$

We can easily see from (10) that the sum $\beta_{\mathbf{u}^{i+1},k-c^i_0} = u_{k-c^i_0} + u_{k+k_1-c^i_0} + \cdots + c_{k+k_{w-1}-c^i_0}$ is defined since every entry in the sum is defined. The corresponding sum over $\mathbf{z}$ is $\gamma_{\mathbf{z},t-1} = z_{t-1} + z_{t+j_1-1} + \cdots + z_{t+j_{w-1}-1}$.

C Searching for Parity Check Equations

C.1 The Generator Matrix

Let $g(x) = 1 + g_{l-1}x + g_{l-2}x^2 + \cdots + g_1x^{l-1} + x^l$, $g_i \in F_2$, $g_l = g_0 = 1$ be the primitive feedback polynomial of degree l for a shift register that generates the sequence $\mathbf{u} = (u_0, u_1, \ldots, u_{N-1})$. The corresponding recurrence is $u_{t+l} = g_1u_{t+l-1} + g_2u_{t+l-2} + \cdots + g_lu_t$. Let α be defined by $g(\alpha) = 0$. From this we get the reduction rule $\alpha^l = g_1\alpha^{l-1} + g_2\alpha^{l-2} + \cdots + g_{l-1}\alpha + 1$. Then we can define the generator matrix for sequence u_t,$0 < t < N$ by the $l \times N$ matrix

$$G = [\alpha^0\alpha^1\alpha^2 \ldots \alpha^{N-1}]. \tag{11}$$

For each $i > l$, using the reduction rule, α^i can be written as $\alpha^i = h^i_{l-1}\alpha^{l-1} + \cdots + h^i_2\alpha^2 + h^i_1\alpha + h^i_0$. We see that every column $i \geq l$ is a combination of the first l columns, and any column i in G can be represented by

$$\mathbf{g}_i = [h^i_0, h^i_1, \ldots, h^i_{l-1}]^{\mathbf{T}}. \tag{12}$$

Now the sequence $\mathbf{u}$ with length N and initialization state $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, \ldots, u_{l-1})$, can be generated by

$$\mathbf{u} = \mathbf{u}^{\mathrm{I}}G.$$

The shift register is now turned in to a (N, l) block code.

C.2 Equations

Let **u** be a sequence generated by the generator polynomial $g(x)$ with degree l. It is well known that, if we can find $w-1>2$ columns in the generator matrix G, that sum to zero,

$$(\mathbf{g}_0 + \mathbf{g}_{j_1} + \ldots + \mathbf{g}_{j_{w-1}})^{\mathbf{T}} = (0, 0, \ldots, 0), \tag{13}$$

for $l \leq j_1, \ldots, j_{w-1} < N$, we get an equation of the form

$$u_t + u_{t+j_1} + \cdots + u_{t+j_{w-1}} = 0. \tag{14}$$

The equation (13) can be formulated as $1 + \alpha^{j_1} + \cdots + \alpha^{j_{w-1}} = 0$. Thus, if (13) holds, the equation $\alpha^t(1 + \alpha^{j_1} + \cdots + \alpha^{j_{w-1}}) = \alpha^t + \alpha^{j_1+t} + \cdots + \alpha^{j_{w-1}+t} = 0$ also holds for $0 \leq t < N - j_{w-1}$. From this we can conclude that the equation is cyclic and can be written as

$$u_t + u_{t+j_1} + \cdots + u_{t+j_{w-1}} = 0, \tag{15}$$

for $0 \leq t < N - j_{w-1}$.

We can also use the indexes $j_1, j_2, \ldots, j_{w-1}$ to formulate the polynomial $h(x) = 1 + x^{j_1} + \cdots + x^{j_{w-1}}$. If $j_0, j_1, \ldots, j_{w-1}$ is found using the method above, we will have the relationship $h(x) = g(x)a(x)$ for a polynomial $a(x)$. Thus, $h(x)$ is a multiple of $g(x)$.

C.3 Fast Method for Finding an Multiple Weight $w = 2^r$

A previous and naive search algorithm for finding multiple $h(x)$ of weight w and degree $<n$ is as follows. It corresponds to searching for w columns in G that sum to zero mod 2.

First sort the generator matrix G corresponding to the $l-1$ lowest bits, that is, we ignore the first bit in the columns. From G we have that $\mathbf{g}_0 = (1, 0, \ldots, 0)^{\mathrm{T}}$. Next for every choice for the $w-2$ columns $j_2, j_2, \ldots, j_{w-1}$ in G search for the column $\mathbf{g}_{j_{w-1}}$ in G that gives $(1, 0, \ldots, 0)^{\mathrm{T}} = \mathbf{g}_{j_1} + \mathbf{g}_{j_2} + \cdots + \mathbf{g}_{j_{w-1}}$. This algorithm is not very efficient and has the complexity $O(n^{w-2}\log_2 n)$. By using hashing techniques we can get down to $O(n^{w-2})$. However, we can do better if we use the iterative method explained next. The algorithm is a modification of the generalized birthday algorithm in [8] and the search for equation algorithm in [6].

First we sort the $n \times l$ generator matrix $G_1 = G$ in respect to the $l - B_1$ lowest entries in the columns, for a proper number B_1. The columns that are equal in the lowest $l - B_1$ bits, will now lie beside each other. If we sum them, the sum will be zero in the lowest $l - B_1$ bits. Next we go through the matrix and sum all the columns that are equal in the $l - B_1$ lowest entries, and store the sums in a new matrix G_2. If we find m_1 sums, the matrix G_2 will have size $m_1 \times B_1$, since the m_1 sums we find will have 0 in the $l - B_1$ lowest entries. For each column i in G_2 we also store the indexes of the two columns from G that where summed to column i. Next we sort G_2 in respect to the $B_1 - B_2$ lowest bits, and do the same procedure over again and get a new matrix G_3 of size $m_2 \times B_2$.

We repeat the procedure until we in round r set B_r to be zero. After the r'th round we will hopefully have found 2^r columns in G that sum to the zero column. According to Section C.2 we will now have found a multiple of $g(x)$. This algorithm is much faster than the naive algorithm, but it "misses" a lot possible multiples and needs bigger matrix G.

Now we will present some new properties for this algorithm. The first round of the algorithm is similar to the well known search algorithm in [4] for finding equations of the type $c_0u_0 + c_1u_1 + \cdots + c_{B-1}u_{B-1} = u_i + u_j$. From this paper we have that the expected number of equations m_1 is given by $m_1 = n(n-1)/2^{l-B_1}$. When n is large we can approximate m_1 by

$$E(m_1) = \frac{n^2}{2^{l-B_1+1}}. \tag{16}$$

Since the algorithm is iterative we can use (16) over again for the next round and we have $E(m_2) = \frac{m_1^2}{2^{B_1-B_2+1}} = \frac{N^4}{2^{2l-B_1-B_2+3}}$. Generally for each round i we will have

$$E(m_i) = \frac{m_{r-1}^2}{2^{B_{i-1}-B_i+1}} \tag{17}$$

for $B_0 = l$ and $m_0 = N$.

The iterative search algorithm has complexity $O(\sum_{i=0}^{r-1} m_i \log_2 m_i)$ since we have to sort the matrices $G_1, G_2, \ldots, G_r$. Thus, the memory limits the algorithm, not the run time complexity. Given an polynomial $g(x)$ of degree l, we will now present a bound for needed memory for finding a multiple $h(x)$ of weight $w = 2^r$.

Assume that we have a computer with T_{mem} memory units and that one column in G_1 takes up one memory unit. It will be natural to use a column G of the maximum size $T_{\text{mem}} \times l$. To use the memory most efficiently, we will try find around $m_i = T_{\text{mem}}$ sums in each round i, that is $G_i = T_{\text{mem}} \cdot B_{i-1}$. Thus, we can set $N = m_1 = \cdots = m_{r-1} = T_{\text{mem}}$. We just need find one multiple, so $m_r = 1$. Setting these restriction we can now give an easy expression for how much memory that is needed to find a multiple of weight $w = 2^r$ of $g(x)$ of degree l.

Theorem 2. *Given a primitive polynomial $g(x)$ of degree l, and $r+1$ divides $r+l$, the expected amount of memory needed to find a weight $w = 2^r$ multiple $h(x)$ of $g(x)$ using the iterative search algorithm is*

$$T_{\text{mem}}(l_{\text{u}}, r) = 2^{\frac{r+l}{r+1}}, \tag{18}$$

with $B_i = i + l - i\frac{r+l}{r+1}$, $1 \le i \le r-1$, $B_r = 0$.

Proof. From equation (17) we have these formulas for $m_1, \ldots, m_r$:

$$\begin{aligned} m_1 &= \tfrac{n^2}{2^{l-B_1+1}}, \\ m_2 &= \tfrac{m_1^2}{2^{B_1-B_2+1}}, \\ &\vdots \\ m_r &= \tfrac{m_{r-1}^2}{2^{B_{r-1}-B_r+1}}. \end{aligned}$$

We require that $m_1 = m_2 = \cdots = m_{r-1} = n = T_{\text{mem}}$, and $m_r = 1$. We solve $n = m_1 = \frac{n^2}{2^{l-B_1+1}}$, in respect to B_1 and get

$$B_1 = 1 + l - \log_2 n. \tag{19}$$

We use equation (17) and solve $n = \frac{n^2}{2^{B_{i-1}+B_i+1}}$ in respect to B_i and get

$$B_i = B_{i-1} + 1 - \log_2 n. \tag{20}$$

Using (20) together with B_1, we get this expression for B_i :

$$B_i = i + l - i\log_2 n. \tag{21}$$

Next we solve $m_r = 1$, that is $\frac{n^2}{2^{B_{r-1}-B_r+1}} = 1$. By solving in respect to n, putting in (21) for $i = r - 1$ and setting $B_r = 0$, we get $n = 2^{\frac{r+l}{r+1}}$. The algorithm requires that all the B_i's are integers. This will only be the case when we can set $n = 2^x$, for some x. If we want the expression to be exact, we get the requirement that $\frac{r+l}{r+1}$ must be an integer. Thus, $r + 1$ must divide $r + l$.

The theorem does not give a guarantee for finding an equation, it just say that we are expected to find one. Thus, in practical searches we may use around twice as many bits to assure success.

An Improved Correlation Attack Against Irregularly Clocked and Filtered Keystream Generators

Håvard Molland and Tor Helleseth

The Selmer Center[**]
Institute for Informatics,
University of Bergen,
Norway

Abstract. In this paper we propose a new key recovery attack on irregularly clocked keystream generators where the stream is filtered by a nonlinear Boolean function. We show that the attack is much more efficient than expected from previous analytic methods, and we believe it improves all previous attacks on the cipher model.

Keywords: Correlation attack, Stream cipher, Boolean functions, irregularly clocked shift registers.

1 Introduction

In this paper we present a new key recovery correlation attack on ciphers based on an irregularly clocked linear feedback shift register (*LFSR*) filtered by a Boolean function. The cipher model we attack is composed of two components, the clock control generator and the data generator and is shown in Fig. 1.

- *The data generator sub system* consists of $\mathrm{LFSR_u}$ of length l_u and the nonlinear multivariate Boolean function f, where the internal state of $\mathrm{LFSR_u}$ is filtered by the f function. The output from f is the bit stream $\mathbf{v}$ which has high linear complexity.
- *The clock control sub system* consists of $\mathrm{LFSR_s}$ of length l_s where the output from $\mathrm{LFSR_s}$ is sent through the clock function $D()$. The output from $D()$ is the clock control sequence of integers, $\mathbf{c}$, which is used to clock $\mathrm{LFSR_u}$.

The effect of the irregularly clocking is that $\mathbf{v}$ is irregularly decimated and the positions of the bits in the stream are altered. The result from this decimation is the keystream $\mathbf{z}$. The secret key in this cipher is the $(l_\mathrm{u}+l_\mathrm{s})$ initialization bits for $\mathrm{LFSR_u}$ and $\mathrm{LFSR_s}$ $(\mathbf{I}_\mathrm{u}, \mathbf{I}_\mathrm{s})$.

[**] This work was supported by the Norwegian Research Council.

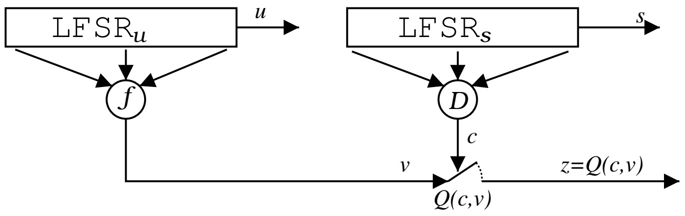

Fig. 1. The general cipher model we attack in this article

To attack this encryption scheme we need to know the positions the keystream bits $\mathbf{z}$ had in the stream $\mathbf{v}$ before $\mathbf{v}$ was irregularly decimated. The previous effective algorithms are not specially designed to attack irregularly clocked and filtered generators. However, there exist effective attacks on the data generator sub system[6,1,10,3,4]. To deal with the irregularly clocking, one of two techniques are often used:

1. **Do the attack on the data generator 2^{l_s} times**[7]. The attack is done one time for each guess for the 2^{l_s} possible initialization states for LFSR$_s$. If the attack on the sub system has complexity $O(K)$ the full attack will have complexity $O(K \cdot 2^{l_s})$.
2. **Ignore the clock control generator**[3,14,4]. If the attack on the data generator subsystem needs M keystream bits, we can use the fact[14] that we know the original $\mathbf{v}$ position of every $2^{l_s} - 1$ bit in the keystream $\mathbf{z}$. Thus, we can only use every $2^{l_s} - 1$ keystream bit in the attack, which means that we need $(2^{l_s} - 1) \cdot M$ keystream bits to succeed.

None of these techniques are optimal. The first one leads to large runtime complexity, the second leads to the need for a large number of keystream bits.

Our attack is not designed to attack the data generator subsystem only, but is especially aimed at irregularly clocked and filtered keystream generators as one system. First we guess the initialization state $\mathbf{I}_s$ for LFSR$_s$. From this we can reconstruct the positions the bits in $\mathbf{z}$ had in $\mathbf{v}$. Using the iteration algorithm from [11], this reconstruction is done using just a couple of operations per guess, exploiting the cyclic redundancies in LFSR$_s$. This method is fully explained in Section 4.3. This method gives the guess $\mathbf{v}^* = (\ldots, *, z_i, \ldots, z_j, \ldots, *, \ldots, z_k, \ldots, *, \ldots)$, where z_i, z_j, z_k are some keystream bits and the stars are the deleted bits. Then we test $\mathbf{v}^*$ to see if it is likely that the stream is generated by the data generator subsystem LFSR$_u$ and f. Hence, we only use a distinguisher test on the the $\mathbf{v}^*$ stream to decide if the guess for $\mathbf{I}_s$ is correct. This is easier than to actually decode the $\mathbf{v}^*$ stream to find $\mathbf{I}_u$, and then decide if we have found the correct $\mathbf{I}_s$. When $\mathbf{I}_s$ is determined, we can use one of the previous attacks on the data generator sub system to determine $\mathbf{I}_u$.

The distinguisher test is to evaluate a large number m of low weight parity check equations on the bit stream $\mathbf{v}^*$. All equations are derived from one multiple $h(x)$ of weight 4 of the generator polynomial $g_{\mathrm{u}}(x)$. Surprisingly this test works much better than expected from previous evaluation methods. In previous correlation attacks, the Piling up lemma[9] is often used to calculate the correlation[1,7,6] which the algorithm must decode. Since our algorithm only uses a distinguisher on $\mathbf{v}^*$ we can use a correlation property of the function f which gives much higher correlation between $\mathbf{v}^*$ and the keystream $\mathbf{z}$. Hence, we need fewer parity check equations. This correlation property exists even if the function is correlation immune in the normal sense of the term.

Our attack has complexity $O(2^{l_{\mathrm{s}}} \cdot m)$, independently of the length of $\mathrm{LFSR}_{\mathrm{u}}$. A cipher based on the model we attack in this paper is LILI-128. To attack the LILI-128 cipher our algorithm needs about 2^{23} parity check equations. In LILI-128, $l_{\mathrm{s}} = 39$. Thus, the runtime for our attack is $2^{39+23} \approx 2^{62}$ parity checks, with virtually no precomputation. We have implemented and tested the attack, and it works on computers having under 300 MB of RAM, and needs only around 68 Mbyte of keystream data. The precomputation has low runtime complexity and is negligible. When $\mathbf{I}_{\mathrm{s}}$ is found, we can use one of the previous algorithms to attack the data generator sub system.

A comparable previous correlation attack by Johansson and Jönsson is presented in [7]. The runtime for the attack is equivalent to 2^{71} parity checks and the pre-computation is 2^{79} table lookups. The keystream length is approximately 2^{30}. This attack uses the first technique to handle the irregularly clocking.

Recently new algebraic attacks have been proposed by Courtois and Meier[3,4]. This attack uses the second technique to handle the irregularly clocking in LILI-128. Although the attack has an impressive runtime complexity $2^{31} \cdot C$ (an optimistic estimation for some unknown constant C), the attack needs about 2^{60} keystream bits to succeed, which is unpractical.

There is also a time-memory trade-off attack against LILI-128 by Markku-Juhani Olavi Saarinen[14]. This attack needs approximately $2^{51.4}$ bits of computer memory and 2^{46} keystream bits. The runtime complexity is claimed to be 2^{48} DES operations, which is not easy to compare with our runtime complexity. However, the high use of computer memory and keystream bits also makes this attack unpractical.

2 A Correlation Property of Nonlinear Functions

Let $V = F_2^n$ and let f be a balanced Boolean function from V to F_2. We start by analyzing the Boolean function $f(\mathbf{x})$ for a correlation property that we will use in the attack. A similar property is analyzed in [18] where they look at the nonhomomorphicity of functions. In this paper we identify the probability

$$p = P(f(\mathbf{x}_1) + f(\mathbf{x}_2) + f(\mathbf{x}_3) + f(\mathbf{x}_4) = 0 \mid \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4 = \mathbf{0}) \qquad (1)$$

which is crucial for the success rate of our attack.

2.1 The Correlation Property

Let $q = 2^n$ and let $\mathbf{a}\cdot\mathbf{b} = \sum_{i=1}^{n} a_i b_i$ denote the inner product of $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$. Define the Walsh coefficients of f by

$$\hat{f}(\mathbf{a}) = \sum_{\mathbf{x}\in V} (-1)^{f(\mathbf{x})+\mathbf{a}\cdot\mathbf{x}}.$$

Lemma 1. *Let f be a function from $V = F_2^n$ to F_2 and let $\mathbf{x}_i \in F_2^n$ for $i = 1, 2, 3, 4$. Let $q = 2^n$ and let N denote the number of solutions of*

$$\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4 = \mathbf{0} \tag{2}$$

$$f(\mathbf{x}_1) + f(\mathbf{x}_2) + f(\mathbf{x}_3) + f(\mathbf{x}_4) = 0. \tag{3}$$

Then

$$N = \frac{q^3}{2} + \frac{1}{2q}\sum_{\mathbf{a}\in V} \hat{f}(\mathbf{a})^4. \tag{4}$$

Proof. Each term in the sum below gives a contribution $2q$ for each solution of the system of equations, and zero otherwise. Therefore, we have

$$\begin{aligned}
2qN &= \sum_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3,\mathbf{x}_4\in V} \Big(\sum_{\mathbf{a}\in V}(-1)^{\mathbf{a}\cdot(\mathbf{x}_1+\mathbf{x}_2+\mathbf{x}_3+\mathbf{x}_4)}\Big)\Big(\sum_{y=0}^{1}(-1)^{y(f(\mathbf{x}_1)+f(\mathbf{x}_2)+f(\mathbf{x}_3)+f(\mathbf{x}_4))}\Big) \\
&= \sum_{\mathbf{a}\in V}\sum_{y=0}^{1}\sum_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3,\mathbf{x}_4\in V} (-1)^{yf(\mathbf{x}_1)+\cdots+yf(\mathbf{x}_4)+\mathbf{a}\cdot\mathbf{x}_1+\cdots+\mathbf{a}\cdot\mathbf{x}_4} \\
&= \sum_{\mathbf{a}\in V}\sum_{y=0}^{1}\Big(\sum_{\mathbf{x}\in V}(-1)^{yf(\mathbf{x})+\mathbf{a}\cdot\mathbf{x}}\Big)^4 \\
&= q^4 + \sum_{\mathbf{a}\in V}\hat{f}(\mathbf{a})^4,
\end{aligned}$$

where the first term comes from the case $y = 0$ and $\mathbf{a} = \mathbf{0}$, and the last term from the case $y = 1$.

Corollary 1. *If $f(\mathbf{x})$ is a balanced function then the number of solutions N of the system of equations above is,*

$$N \geq \frac{q^3}{2} + \frac{q^3}{2(q-1)}.$$

Proof. Since $f(\mathbf{x})$ is balanced we obtain $\hat{f}(\mathbf{0}) = \sum_{\mathbf{x}\in V}(-1)^{f(\mathbf{x})} = 0$. It follows from Parseval's identity that the average value of $\hat{f}(\mathbf{a})^2$ is $\frac{q^2}{q-1}$. Hence, it follows from the Cauchy-Schwartz inequality that $\sum_{\mathbf{a}\in V}\hat{f}(\mathbf{a})^4 \geq (q-1)\frac{q^4}{(q-1)^2}$, which substituted in the lemma above gives the result.

Corollary 2. *The expected number of solutions N of the system of equations above is,*

$$E(N) = \frac{q^3}{2} + \frac{3q^2 - 2q}{2}.$$

Proof. An average estimate of N can be found as follows. When there exist two equal vectors $x_{i_1} = x_{i_2}$ in Equation (2), the two other vectors x_{i_3}, x_{i_4} will also be equal. When this occurs it follows that the Equation (3) will sum to zero. This gives the unbalance that causes the high correlation. Equation (2) implies $\mathbf{x}_4 = \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3$ Then there are $q(q-1)(q-2)$ triples in $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ where all the $\mathbf{x}_i$'s are distinct and there are therefore $3q^2 - 2q$ triples with one or two pairs $\mathbf{x}_{i_1} = \mathbf{x}_{i_2}$. Using this fact and substituting Equation (2) into Equation (3), we can write

$$\begin{aligned}
2N &= \sum_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3 \in V} \sum_{y=0}^{1} (-1)^{y(f(\mathbf{x}_1)+f(\mathbf{x}_2)+f(\mathbf{x}_3)+f(\mathbf{x}_1+\mathbf{x}_2+\mathbf{x}_3))} \\
&= q^3 + \sum_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3 \in V} (-1)^{f(\mathbf{x}_1)+f(\mathbf{x}_2)+f(\mathbf{x}_3)+f(\mathbf{x}_1+\mathbf{x}_2+\mathbf{x}_3)} \\
&= q^3 + (3q^2 - 2q) + \sum_{\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3 \text{ distinct } \in V} (-1)^{f(\mathbf{x}_1)+f(\mathbf{x}_2)+f(\mathbf{x}_3)+f(\mathbf{x}_1+\mathbf{x}_2+\mathbf{x}_3)}.
\end{aligned}$$

Since for an arbitrary function f we can expect that $f(\mathbf{x}_1)$, $f(\mathbf{x}_2)$, $f(\mathbf{x}_3)$, and $f(\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3)$ take on all binary quadruples approximately equally often when $\mathbf{x_1} \neq \mathbf{x_2} \neq \mathbf{x_3} \neq \mathbf{x_1}$, we expect in the average the last term to be 0. This implies the result.

Corollary 3. *Let f be an arbitrary balanced function, and let p denote the probability*

$$p = Prob(f(\mathbf{x}_1) + f(\mathbf{x}_2) + f(\mathbf{x}_3) + f(\mathbf{x}_4) = 0 \mid \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4 = \mathbf{0}),$$

then p is expected to be $E(p) = \frac{1}{2} + \frac{3q-2}{2q^2}$ and its minimum is $p_{min} \geq \frac{1}{2} + \frac{1}{2(q-1)}$.

Proof. Since Equation (2) has q^3 solutions, it follows from Corollary 1 that the expected probability is equal to $E(p) = \frac{E(N)}{q^3} = \frac{1}{2} + \frac{3q-2}{2q^2}$. Further from Corollary 2 we obtain that the minimum is $p_{min} \geq (\frac{q^3}{2} + \frac{q^3}{2(q-1)})/q^3 = \frac{1}{2} + \frac{1}{2(q-1)}$.

Corollary 4. *Given a specific balanced function f, the probability*

$$p = Prob(f(\mathbf{x}_1) + f(\mathbf{x}_2) + f(\mathbf{x}_3) + f(\mathbf{x}_4) = 0 \mid \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 + \mathbf{x}_4 = \mathbf{0}),$$

is $p = \frac{1}{2} + \frac{\sum_{\mathbf{a} \in V} \hat{f}(\mathbf{a})^4}{2q^4}$

Proof. Using the N from Lemma 1 we get $p = \frac{N}{q^3} = \frac{1}{2} + \frac{\sum_{\mathbf{a} \in V} \hat{f}(\mathbf{a})^4}{2q^4}$

It is straightforward to extend Lemma 1 to compute the number of common solutions of the two equations

$$\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_w = \mathbf{0}$$
$$f(\mathbf{x}_1) + f(\mathbf{x}_2) + \cdots + f(\mathbf{x}_w) = 0.$$

and show that the corresponding probability

$$\mathrm{Prob}(f(\mathbf{x}_1) + f(\mathbf{x}_2) + \cdots + f(\mathbf{x}_{w-1}) = 0 \mid \mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_{w-1} = \mathbf{0}),$$

equals $p = \frac{1}{2} + \frac{\sum_{\mathbf{a} \in V} \hat{f}(\mathbf{a})^w}{2q^w}$, which reduces to the result of Corollary 4 when $w = 4$.

In the case $w = 3$, we can calculate the expected value of a balanced Boolean function, with a given $f(\mathbf{0})$, to be $E(p) = \frac{1}{2} + \frac{3q-2}{2q^2}(-1)^{f(\mathbf{0})}$. This implies that the bias is the same for the case $w = 3$ as for $w = 4$. Similar arguments for equations with $w \geq 5$ show that these equations give too low correlation, which would lead to a high runtime complexity for our attack. It turns out that for $w = 3$ the attack needs much more keystream bits to succeed, see the Sections 4.1 and 5.2. Since the correlation bias is exactly the same for $w = 3$ and $w = 4$ it is optimal to use $w = 4$.

2.2 Analysis of Some Functions

In Table 1 we have analyzed some functions using Corollary 4. This correlation is surprisingly high. Let $p_{\mathrm{app}} = 0.53125$ be the best linear approximation to the LILI-128 function. Due to the design of the previous attacks[6,7,10] the channel noise has been independent of the stream $\mathbf{u}$ generated by LFSR$_\mathrm{u}$. Thus, the Piling up lemma [9], $p_{\mathrm{pil}} = \frac{1}{2} + 2^{w-1}(\frac{1}{2} - p_{\mathrm{app}})^w$, is used to evaluate the crossover correlation $1 - p_{\mathrm{pil}}$ which the algorithms must be able to decode. Using the Piling up lemma for weight $w = 4$ equations, the correlation p_{pil} for LILI-128 will be $p_{\mathrm{pil}} = 0.50000763$. From Table 1 we have the correlation $p = 0.501862$. The

Table 1. The probability $P(f(\mathbf{x}_1)+f(\mathbf{x}_2)+f(\mathbf{x}_3)+f(\mathbf{x}_4) = 0 \mid \mathbf{x}_1+\mathbf{x}_2+\mathbf{x}_3+\mathbf{x}_4 = \mathbf{0})$ calculated for some given functions. $E(p)$ is the expected correlation for given $q = 2^n$ and p is the actual correlation for the given function

Function	Number of inputs bits n	Best linear approximation.	$E(p)$	p
Geffe function	2	0.75	0.671875	0.625
LILI-128	10	0.53125	0.501464	0.501862
LILI-II	12	0.51367	0.500366	0.500190

reason for the higher correlation, is that our attack only uses a distinguisher on the data generator sub system, and not a complete decoder. Hence, in our key recovery attack on the clock control system, we can use Corollary 4 from Section 2.1 to calculate the correlation. To test the corollary we generated 2000 random and balanced Boolean tables for $n = 10$, and calculated the average correlation. The result was that the average p was 0.501466 which is close to the theoretical expected $E(p) = 0.5001464$.

3 A General Model

Here we define a general model for irregularly clocked and filtered stream ciphers, and some well known properties for the model.

3.1 General Model

Let $g_{\mathrm{u}}(x)$ and $g_{\mathrm{s}}(x)$ be the feedback polynomials for the shift registers LFSR$_{\mathrm{u}}$ of length l_{u} and LFSR$_{\mathrm{s}}$ of length l_{s}. We let $\mathbf{I}_{\mathrm{s}} = (s_0, s_1, \ldots, s_{l_{\mathrm{s}}-1})$ and $\mathbf{I}_{\mathrm{u}} = (u_0, u_1, \ldots, u_{l_{\mathrm{u}}-1})$ be the initialization states for LFSR$_{\mathrm{s}}$ and LFSR$_{\mathrm{u}}$. The initialization states $(\mathbf{I}_{\mathrm{s}}, \mathbf{I}_{\mathrm{u}})$ define *the secret key* for the given cipher system.

From $g_{\mathrm{s}}(x)$ we can calculate a clock control sequence $\mathbf{c}$ in the following way. Let $c_t = D(L_{\mathrm{s}}^t(\mathbf{I}_{\mathrm{s}})) \in \{a_1, a_2, \ldots, a_A\}$, $a_j \geq 0$, be a function where the input $L_{\mathrm{s}}^t(\mathbf{I}_{\mathrm{s}})$ is the inner state of LFSR$_{\mathrm{s}}$ after t feedback shifts and A is the number of values that c_t can take. Let p_j be the probability $p_j = \mathrm{Prob}(c_t = a_j)$.

LFSR$_{\mathrm{u}}$ produces the stream $\mathbf{u} = (u_0, u_1, \ldots)$ which is filtered by f. The output from f is $v_k = f(u_{k+i_0}, u_{k+i_1}, \ldots, u_{k+i_{n-1}})$, or the equivalent $v_k = f(L_{\mathrm{u}}^k(\mathbf{I}_{\mathrm{u}}))$. The clock c_t decides how many times LFSR$_{\mathrm{u}}$ is clocked before the output bit v_k is taken as keystream bit z_t. Thus, the keystream z_t is produced by $z_t = v_{k(t)}$, where $k(t)$ is the total sum of the clock at time t, that is $k(t) \leftarrow k(t-1) + c_t$. This gives the following definition for the clocking of LFSR$_{\mathrm{u}}$.

Definition 1. *Given bit stream* $\mathbf{v}$ *and clock control sequence* $\mathbf{c}$*, let* $\mathbf{z} = Q(\mathbf{c}, \mathbf{v})$ *be the function that generates* $\mathbf{z}$ *of length* M *by*

$$Q(\mathbf{c}, \mathbf{v}) : \; z_t \leftarrow v_{k(t)}, \; 0 \leq t < M$$

where $k(t) = \sum_{j=0}^{t} c_j - 1$.

If $a_j \geq 1, \;\; 1 \leq j \leq A$, the function $Q(\mathbf{c}, \mathbf{v})$ can be considered as a deletion channel with input $\mathbf{v}$ and output $\mathbf{z}$. The deletion rate is

$$P_{\mathrm{d}} = 1 - \frac{1}{\sum_{j=1}^{A} p_j a_j}. \tag{5}$$

The $D()$ function described above can in this model be among others the shrinking generator, the step-1/step-2 generator and the stop and go generator. Next we define the (not complete) reverse of Definition 1.

Definition 2. *Given the clock control sequence* $\mathbf{c}$ *and keystream* $\mathbf{z}$*, let the function* $\mathbf{v}^* = Q^*(\mathbf{c},\mathbf{z})$ *be the (not complete) reverse of* Q*, defined as*

$$Q^*(\mathbf{c},\mathbf{z}) : \, v^*_{k(t)} \leftarrow z_t, \, 0 \le t < M,$$

where $k(t) = \sum_{j=0}^{t} c_j - 1$*, and* $v_k = *$ *for the entries* k *in* $\mathbf{v}^*$ *where* v^*_k *is deleted. When this occurs we say that* v^*_k *is not defined.*

The length of $\mathbf{v}^*$ will be $N^* = \sum_{j=0}^{M-1} c_j$. Given a stream $\mathbf{z}$ of length M, the expected length N of the stream $\mathbf{v}$ is

$$\mathrm{E}(N) = \frac{M}{(1-P_{\mathrm{d}})} = M \sum_{j=1}^{A} p_j a_j. \tag{6}$$

Note that the only difference between this definition and Definition 1, is that $\mathbf{v}$ and $\mathbf{z}$ have switched sides. Thus, $Q^*(\mathbf{c},\mathbf{z})$ is a reverse of $Q(\mathbf{c},\mathbf{v})$. However, since some bits are deleted, the reverse is not complete and we get the stream $\mathbf{v}^*$.

The probability for a bit v^*_k being defined is $\mathrm{Prob}(v^*_k) = 1 - P_{\mathrm{d}}$. This happens when $k = k(t)$ holds for some t, $0 \le t < M$. It follows that the sum $v^*_k + v^*_{k+j_1} + \cdots + v^*_{k+j_{w-1}}$ will be defined if and only if all of the bits in the sum are defined. Thus, the sum will be defined for given k in $\mathbf{v}^*$ with probability

$$P_{\mathrm{def}} = (1 - P_{\mathrm{d}})^w. \tag{7}$$

4 The Attack

4.1 Equations of Weight 4

To succeed with our attack we need to find exactly one weight 4 equation

$$\lambda_{\mathrm{u}} : \, u_k + u_{k+j_1} + u_{k+j_2} + u_{k+j_3} = 0 \tag{8}$$

that holds over all $\mathbf{u}$ generated by LFSR$_{\mathrm{u}}$ for $k \ge 0$. This corresponds to finding a multiple $h(x) = a(x)g_{\mathrm{u}}(x)$ of weight 4. There exist several algorithms for finding such a multiple, see among others [13,2,5,17,12].

In this paper we use the fast search algorithm in [12,11], which is a modified version of the David Wagner's generalized birthday algorithm[17]. If the stream $\mathbf{u}$ has length N, this algorithm has runtime complexity $O(N \log N)$ and memory complexity $O(N)$, where N is of order $2^{l_{\mathrm{u}}/3}$. The algorithm is effective in practice, and we have succeeded in finding multiples of the generator polynomial of high degree, see Section 6.3 for an example. We refer to Appendix C in [11] for the details for this search algorithm.

Next, we let the input vector $\mathbf{x}_k$ to the Boolean function $f(\mathbf{x})$ be

$$\mathbf{x}_k = (u_{k+i_0}, u_{k+i_1}, \ldots, u_{k+i_{n-1}}), \tag{9}$$

where $(i_0, i_1, \ldots, i_{n-1})$ defines the tapping positions from the internal state $L_{\mathrm{u}}^k(\mathbf{I}_{\mathrm{u}})$ of LFSR$_{\mathrm{u}}$ after k feedback shifts. Substituting the vector (9) into the Equation (8) we have that $\mathbf{x}_k + \mathbf{x}_{k+j_1} + \mathbf{x}_{k+j_2} + \mathbf{x}_{k+j_3} = \mathbf{0}$ always holds for $k \geq 0$. Since $v_k = f(\mathbf{x}_k)$ we have from Corollary 4 that the equation

$$\lambda_{\mathrm{v}} : \; v_k + v_{k+j_1} + v_{k+j_2} + v_{k+j_3} \approx 0, \tag{10}$$

will hold for $k \geq 0$ with probability $p = \frac{1}{2} + \frac{\sum_{\mathbf{a} \in V} \hat{f}(\mathbf{a})^4}{2q^4}$.

Remark 1. In [8] the multiple of $g_{\mathrm{u}}(x)$ of weight $w = 3$ is exploited to define an iterative decoding attack on regularly clocked LFSRs filtered by Boolean functions. The constrained system

$$\begin{aligned} \sum_{a=0}^{w-1} \mathbf{x}_{k+j_a} &= \mathbf{0} \\ z_{k+j_a} &= f(\mathbf{x}_{k+j_a}),\ 0 \leq a < w \end{aligned} \tag{11}$$

is analyzed. This system is similar to the one we use in this paper, but it is used differently. Since there are limited solutions to this system, the *a posteriori* probabilities for each of the *input bits* $(u_{k+j_a+i_0}, u_{k+j_a+i_1}, \ldots, u_{k+j_a+i_{n-1}})$ in $\mathbf{x}_{k+j_a}$ can be calculated. Then these probabilities are put into a Gallager like probabilistic decoding algorithm(SOJA) which outputs $\mathbf{I}_{\mathrm{u}}$. However the correlation property in Corollary 4 is neither identified or exploited in [8].

4.2 Naive Algorithm

Let $\hat{\mathbf{I}}_{\mathrm{s}}$ be a guess for the initialization state $\mathbf{I}_{\mathrm{s}}$. Given the keystream $\mathbf{z}$ of length M, we generate $\hat{c}_t = D(L_{\mathrm{s}}^t(\hat{\mathbf{I}}_{\mathrm{s}}))$, $0 \leq t \leq M$ and $\hat{\mathbf{v}}^* = Q^*(\hat{\mathbf{c}}, \mathbf{z})$ of length $N \approx \sum_{t=0}^{M-1} \hat{c}_t^i$. Then we test if $\hat{\mathbf{v}}^*$ is likely to have been generated by LFSR$_{\mathrm{u}}$ using the following method.

Find m entries in $\hat{\mathbf{v}}^*$ where the equation is defined. From this we get a set of m equations. We test the m equations, and let the metric for the guess be the number of equations that hold. When we have the correct guess for $\mathbf{I}_{\mathrm{s}}$ we expect $p \cdot m$ of the equations to hold, where p is calculated using Corollary 4. Thus, this is a maximum likelihood decoding algorithm.

The runtime complexity for the attack will be of order $2^{l_{\mathrm{s}}} \cdot (m + N)$, since we have to generate the bit stream $\hat{\mathbf{v}}^*$ of length N for each of the $2^{l_{\mathrm{s}}}$ guesses. In a real attack, N will be a large number and the naive algorithm will have very high runtime complexity.

4.3 Some Observations

If we use the technique in the previous section the attack has the runtime of order $2^{l_{\mathrm{s}}} \cdot (m + N)$. In [11, Sec. 3.3] two important observations were made that reduce the complexity down to $2^{l_s} \cdot m$. Since $N \gg m$, these observations will speed up the attack considerably. We start with an initial guess $\mathbf{I}_s^0 = (1, 0, \ldots, 0)$ and let the i'th guess be the internal state of LFSR$_s$ after i feedback shifts, that is $\mathbf{I}_s^i = L_s^i(\mathbf{I}_s^0)$.

Let $\mathbf{c}^i = (c_0^i, c_1^i, \ldots, c_{M-1}^i)$ be the i'th guess for the clock control sequence defined by $c_t^i = D(L_s^{i+t}(1, 0, \ldots, 0))$, $0 \le t < M$. Let $\mathbf{v}^i = Q^*(\mathbf{c}^i, \mathbf{z})$ be the corresponding guess for $\mathbf{v}^*$ of length $N_i = \sum_{t=0}^{M-1} c_t^i$. We can now give an iterative method for generating $\mathbf{v}^{i+1}$ from $\mathbf{v}^i$.

Lemma 2. *We can transform $\mathbf{v}^i$ into $\mathbf{v}^{i+1} = Q^*(\mathbf{c}^{i+1}, \mathbf{z})$ using the following method: Delete the first c_0^i entries $(*, \ldots, *, z_0)$ in $\mathbf{v}^i$, append the $c_{M-1}^{i+1} = c_M^i$ entries $(*, \ldots, *, z_M)$ at the end, and replace z_t with z_{t-1} for $1 \le t \le M$.*

Proof. See Appendix B.1 in [11].

Lemma 2 shows that we can generate each $\mathbf{v}^i$ using just a few operations instead of N operations, when implemented properly (See Appendix A.1 for the implementation details). This gives a fast method for generating all possible guesses for $\mathbf{v}^*$ given a keystream $\mathbf{z}$. However, using this lemma we still have to search for m entries in $\mathbf{v}^*$ where the equations are defined. Since on average we must search through $1/P_{\text{def}}$ entries in $\mathbf{v}^*$ per equation, we want to avoid this search. In the next theorem we show how this can be done. The theorem proves that we can reuse the equation set for $\mathbf{v}^i$ in $\mathbf{v}^{i+1}$.

Theorem 1. *If the sum*

$$v_k + v_{k+k_1} + \cdots + v_{k+k_{w-1}} = z_t + z_{t+j_1} + \cdots + z_{t+j_{w-1}} = \gamma_{\mathbf{z},t}$$

is defined over $\mathbf{v}^i$, then the sum

$$v_{k-c_0^i} + \cdots + v_{k+k_{w-1}-c_0^i} = z_{t-1} + z_{t+j_1-1} + \cdots + z_{t+j_{w-1}-1} = \gamma_{\mathbf{z},t-1}$$

is defined over $\mathbf{v}^{i+1}$.

Proof. See Appendix B.2 in [11].

The main result from this theorem is that the equation set defined over $\mathbf{v}^i$ will be defined over $\mathbf{v}^{i+1}$ when we shift the equations c_0^i entries to the left over $\mathbf{v}^{i+1}$. This means that we can just shift the equations one entry to the left over $\mathbf{z}$, and we will have a sum that is defined for the guess $\hat{\mathbf{I}}_s = D(L_s^{i+1}(1, 0, \ldots, 0)$. Thus, the theorem shows that we can avoid a lot of computations if we let the i'th guess for the inner state of LFSR$_s$ be $L_s^i(1, 0, \ldots, 0)$.

Remark 2. To use the lemma and theorem above we do not put the *actual bit values* z_t and restore them to the position $k(t)$ in $\mathbf{v}^*$ given by $Q^*(\mathbf{c}, \mathbf{z})$. Instead we store the *index* z_t (the pointer to the position t in $\mathbf{z}$) in $v_{k(t)}$. This means that $v_{k(t)}^*$ holds the position t, which the keystream bit z_t have in $\mathbf{z}$. However, when we evaluate an equation we use the indexes to put in the actual bit values.

4.4 An Efficient Algorithm

Assume we have found an equation $\lambda_{\mathrm{v}} : v_k + v_{k+j_1} + v_{k+j_2} + v_{k+j_3} \approx 0$. The equation holds over $\mathbf{v}$ with probability p calculated using Corollary 4. Let the first guess for the initialization state for $\mathbf{s}$ be $\mathbf{I}_{\mathrm{s}}^0 = (1, 0, 0, \ldots, 0)$, generate $\mathbf{c}^0$ by $c_t^0 = D(L_{\mathrm{s}}^t(1, 0, \ldots, 0))$, $t < M$, and $\mathbf{v}^0 = Q^*(\mathbf{c}^0, \mathbf{z})$. Next we try to find m entries $(k_1, k_2, \ldots, k_m)$ in $\mathbf{v}^0$ where the equation λ_{v} is defined. From this we get the equation set

$$\begin{array}{rl} v_{k_1}^0 + v_{k_1+j_1}^0 + v_{k_1+j_2}^0 + v_{k_1+j_3}^0 & \approx 0 \\ v_{k_2}^0 + v_{k_2+j_1}^0 + v_{k_2+j_2}^0 + v_{k_2+j_3}^0 & \approx 0 \\ \vdots \qquad\qquad & \vdots \\ v_{k_m}^0 + v_{k_m+j_1}^0 + v_{k_m+j_2}^0 + v_{k_m+j_3}^0 & \approx 0. \end{array} \tag{12}$$

Since every $v_{k_x+j_y}$ in this equation set is defined in $\mathbf{v}^0$ and $z_t = v_{k(t)}$, we can replace $v_{k_x+j_y}$ with the corresponding bit z_{t_x} from the keystream $\mathbf{z}$. Thus, $\mathbf{v}^0$ is a sequence of pointers to $\mathbf{z}$ and we can write the equations over $\mathbf{z}$ as the equation set Ω :

$$\begin{array}{rl} z_{t_{1,1}} + z_{t_{1,2}} + z_{t_{1,3}} + z_{t_{1,4}} & \approx 0 \\ z_{t_{2,1}} + z_{t_{2,2}} + z_{t_{2,3}} + z_{t_{2,4}} & \approx 0 \\ \vdots \qquad\qquad & \vdots \\ z_{t_{m,1}} + z_{t_{m,2}} + z_{t_{m,3}} + z_{t_{m,4}} & \approx 0. \end{array} \tag{13}$$

We are now finished with the precomputation. Let $metric_{\mathrm{best}}$ be the number of equations in Ω that hold. We iterate as follows:

Input The keystream $\mathbf{z}$ of length M, the equation λ, the equation set Ω, the index sequence $\mathbf{v}^0$, the states $L^0(1, 0, \ldots, 0)$ and $L^M(1, 0, \ldots, 0)$, and let $i \leftarrow 0$.

1. Calculate $c_{M-1}^{i+1} = c_M^i = D(L_{\mathrm{s}}^{M+i}(1, 0, \ldots, 0))$.
2. Use Lemma 2 to generate $\mathbf{v}^{i+1} = Q^*(\mathbf{c}^{i+1}, \mathbf{z})$ and lower all indexes in the equation set Ω by one. Theorem 1 guarantees that the equations are defined over $\mathbf{v}^{i+1}$.
3. If the first equation in Ω gets a negative index, then remove the equation from Ω. Find a new index at the end of $\mathbf{v}^{i+1}$ where λ is defined, and add the new equation over $\mathbf{z}$ to Ω.
4. Calculate *metric* as the number of equations in Ω that hold.
5. If $metric_{\mathrm{best}} > metric$, set $metric_{\mathrm{best}} \leftarrow metric$ and $\mathbf{I}_{\mathrm{s}}^i = L_{\mathrm{s}}^i(10, 0, \ldots, 0)$.
6. Set $i \leftarrow i + 1$ and go to step 1.
7. Output $\mathbf{I}_{\mathrm{s}}^i$ as the initialization state for LFSR$_{\mathrm{s}}$.

Remark 3. The algorithm is presented this way to make it readable and to show the basic idea. To reach the complexity $O(2^{l_{\mathrm{s}}} \cdot m)$ a few technical details on the implementation of the algorithm are needed. These details are given in Appendix A.

5 Theoretical Properties

5.1 Success Formula

We can let an (unusual) encoder be defined by removing the Boolean function from the cipher. Then we can use coding theory to evaluate the attack. Let the initialization state $\mathbf{I}_\mathrm{s}$ for LFSR$_\mathrm{s}$ define the information bits in such a system.

Let $\mathbf{y} = (y_0, y_1, \dots, y_{M-1})$ be the (not filtered) irregularly clocked stream from LFSR$_\mathrm{u}$, that is $\mathbf{y} = Q(\mathbf{c}, \mathbf{u})$ and $c_t = D(L_\mathrm{s}^t(\mathbf{I}_\mathrm{s}))$. Then the bitstream $\mathbf{y}$ defines the codeword that is sent over a noisy channel. Let the keystream $\mathbf{z} = Q(\mathbf{c}, \mathbf{v})$ (the filtered version of $\mathbf{y}$) be the received codeword.

Assume we have the wrong guess for $\mathbf{I}_\mathrm{s}$, then approximately $m/2$ of the equations in the set (13) will hold. Now assume we have have guessed the correct $\mathbf{I}_\mathrm{s}$. According to the observation in Section 2.1 the equations in the set (13) will hold with probability $p = \frac{1}{2} + \sum_{\mathbf{a} \in V} \hat{f}(\mathbf{a})^4/2q^4$, independently of the initialization bits $\mathbf{I}_\mathrm{u}$.

Let p define the channel 'noise'. The uncertainty is defined by $H(p) = -p \log_2 p - (1-p)\log_2(1-p)$, and the channel capacity is given by $C(p) = 1 - H(p)$. We can approximate $C(p)$ with $C(p) \approx 2(p - \frac{1}{2})^2/\ln 2$. Following Shannon's noisy coding theorem we can set up this bound for success.

Proposition 1. *The attack will succeed with probability* $> \frac{1}{2}$ *if the number of parity check equations* m *is*

$$m > m_0 = \frac{l_\mathrm{s}}{C(p)} \approx \frac{0.347 l_\mathrm{s}}{(p - \frac{1}{2})^2}$$

where $p \approx \frac{1}{2} + \sum_{y \in V} \hat{f}(y)^4/2q^4$ *and* $q = 2^n$*, where* n *is the number of input bits in* $f(\mathbf{x})$*.*

When m is close to $2 \cdot m_0$ we expect the probability for success to be close to 1, see [15]. The simulations of our algorithm show that if we set $m = 2.1 \cdot m_0$ the success rate is approximately 99%.

5.2 Keystream Length

If the generator polynomial $g_\mathrm{u}(x)$ has weight $w > 4$, we must find a multiple $h(x)$ of $g_\mathrm{u}(x)$ of weight 4 and a degree l_h. We need at least the $\mathbf{v}$ stream to be of length l_h. In addition, to find m entries in $\mathbf{v}$ where the equation is defined $\mathbf{v}$ must at least have length

$$N > l_\mathrm{h} + m/P_\mathrm{def}. \tag{14}$$

From the expectation (6) of N we get $E(M) = N(1-P_\mathrm{d}) = (1-P_\mathrm{d})l_\mathrm{h} + m/(1-P_\mathrm{d})^3$, which proves the following proposition:

Proposition 2. *Let an equation over* $\mathbf{v}$ *be defined by* $h(x)$ *of weight* 4 *and degree* l_{h}. *To obtain an equation set* Ω *of* m *equations over* $\mathbf{z}$, *the length of the* $\mathbf{z}$ *stream must be*

$$M > (1 - P_{\mathrm{d}})l_{\mathrm{h}} + m/(1 - P_{\mathrm{d}})^3. \tag{15}$$

The keystream length M depends on the number of equations m, the deletion rate P_{d} and the degree l_{h} of $h(x)$. The degree l_{h} is then again highly dependent on the search algorithm we use to find $h(x)$. When we use the search algorithm in [11,17] the degree l_{h} of $g_{\mathrm{h}}(x)$ will be of order $l_{\mathrm{h}} = 2^{(2+l_{\mathrm{u}})/3}$, which is close to the theoretical expected degree $2^{l_{\mathrm{u}}/(w-1)}$ [5] for $w = 4$.

5.3 Runtime Complexity

The runtime complexity for our attack is

$$O(2^{l_{\mathrm{s}}} \cdot m) = O(\frac{2^{l_{\mathrm{s}}} \cdot l_{\mathrm{s}}}{(p - \frac{1}{2})^2}) \tag{16}$$

parity check tests, where p is calculated using Corollary 4. Note that the runtime is independent of the length l_{u} of LFSR$_{\mathrm{u}}$.

5.4 Memory Complexity

If we implement the attack directly as described in Sections 4.3 and 4.4 the algorithm will need around $32N + 4 * 32m$ bits of computer memory. The reason for the $32N$ term is that $\mathbf{v}^i = z_0, *, *, z_1, z_2, \ldots, *, z_{M-1}$ of length N is a sequence of pointers of 32 bits. In appendix A.2 we show how we can store $\mathbf{v}^i$ using N memory bits without affecting the runtime complexity. The total amount of memory *bytes* needed is then

$$\frac{N}{8} + 16m \tag{17}$$

6 Simulations of the Attack

The LILI-128 cipher[16] is based on the general model we attack in this paper. To be able to compare our attack with previous attacks, we have tested it on this cipher.

6.1 The LILI-128 cipher

In the LILI cipher the clock control generator is defined by

$$g_{\mathrm{s}}(x) = x^{39} + x^{35} + x^{33} + x^{31} + x^{17} + x^{15} + x^{14} + x^2 + 1,$$

Table 2. We have tested the attack on the LILI-128 Boolean function with $p = 0.501862$. Note that the runtime for finding $\mathbf{I}_\mathrm{s}$ is independent of the length l_u of LFSR$_\mathrm{u}$, and the length M of the keystream. The attack on a full LFSR$_\mathrm{u}$ of length 89 and reduced LFSR$_\mathrm{s}$ of length 11 took 12 seconds

l_s	l_u	Keystream length M	Successes out of 100	m	Runtime	$2^{l_s} \cdot m$
11	60	$2^{24,1}$	59	m_0	6 sec.	2^{31}
11	60	$2^{25,1}$	100	$2.2 \cdot m_0$	13 sec.	2^{32}
11	40	$2^{24,0}$	51	m_0	6 sec.	2^{31}
11	40	$2^{25,0}$	100	$2.2 \cdot m_0$	13 sec.	2^{32}
10	**89**	$\mathbf{2^{29}}$	99	$2.1 \cdot m_0$	6 sec	2^{32}
11	**89**	$\mathbf{2^{29}}$	99	$2.1 \cdot m_0$	12 sec	2^{33}
12	**89**	$\mathbf{2^{29}}$	99	$2.1 \cdot m_0$	24 sec	2^{34}

and $c_t = D(s_{t+12}, s_{t+20}) = 1 + s_{t+12} + 2s_{t+20}$. The data generator sub system is

$$g_\mathrm{u}(x) = x^{89} + x^{83} + x^{80} + x^{55} + x^{53} + x^{42} + x^{39} + x + 1,$$

and $v_k = f(u_k, u_{k+1}, u_{k+3}, u_{k+7}, u_{k+12}, u_{k+20}, u_{k+30}, u_{k+44}, u_{k+65}, u_{k+80})$, defined by a Boolean table of size 1024. Further on we get $P_\mathrm{d} = 0.6$, and $P_\mathrm{def} = 0.0256$ for $w = 4$, and $p = 0.501862$. The number of keybits in the secret key $(\mathbf{I}_\mathrm{s}, \mathbf{I}_\mathrm{u})$ is $39 + 89 = 128$.

6.2 Simulations

We have done the simulations on some versions of the LILI-128 cipher with LFSRs of different lengths to empirically verify the success formula in Section 5.1. Note that we use the full size LFSR$_\mathrm{u}$ from the LILI cipher in the three attacks in the bottom of the table. For $l_s = 11$ and $p = 0.501862$ we get $m_0 = 1.1 \cdot 10^6$.

We have implemented the attack in C code using the Intel icc compiler on a Pentium IV processor. Using the full 32-bit capability and all the implementation tricks explained in Appendix A our implementation uses only approximately 7 cycles per parity check test. Hence the algorithm works fast in practice and will take $7 \cdot 2^{l_\mathrm{s}} m$ processor cycles.

See Table 2 for the simulations. Each attack is run 100 times, and the table shows that the estimated success rate holds and that the algorithm is efficient.

6.3 A Complete Attack on LILI-128

Preprocessing. For the LILI cipher, we have found a multiple $h(x) = a(x)g_\mathrm{u}(x)$ which corresponds to the recursion $u_t + u_{t+139501803} + u_{t+210123252} + u_{t+1243366916} = 0$

and we have that

$$\mathrm{Prob}(v_t + v_{t+139501803} + v_{t+210123252} + v_{t+1243366916} = 0) = 0.501862. \qquad (18)$$

This precomputation took only 5 hours and 40 Gbyte hard disk space. We see that $l_{\mathrm{h}} = 1243366916$.

Finding $\mathbf{I}_{\mathrm{s}}$. We have $p = 0.501862$, and $m_0 = 39/C(0.501862) \approx 3.9 \cdot 10^6 \approx 2^{21.9}$. To be almost sure to succeed we use $m = 2.1 m_0$ equations. Hence, the runtime for attacking LILI-128 is

$$2^{39} \cdot 2^{23} = 2^{62}$$

parity checks. Using our implementation this corresponds to $2^{62} \cdot 7$ processor cycles. Using Proposition 2 with $P_{\mathrm{d}} = 0.6$ we need a keystream of length $M \approx 2^{29}$. The attack needs about 290 Mbyte of RAM. It can easily be parallelized and distributed among processors with virtually no overhead, since there is no need for communication between the processor, and no need for shared memory. If we have 1024 Pentium IV 2.53 GHz processors, each having access to about 290 MB of memory, the attack would take about 4.5 months using 68 Mbyte of keystream data.

Finding $\mathbf{I}_{\mathrm{u}}$ when $\mathbf{I}_{\mathrm{s}}$ is known. Our attack only finds the initialization bits $\mathbf{I}_{\mathrm{s}}$ for LFSR$_{\mathrm{s}}$. It is possible to combine the Quick Metric from [12] with the previous attack against LILI in [7] to find $\mathbf{I}_{\mathrm{u}}$ when $\mathbf{I}_{\mathrm{s}}$ is given. Since this is not the scope of this paper we will not go into details, and we refer to [7,12] for the exact description. The preprocessing stage will have complexity of order $2^{44.7}$ memory lookups, and runtime complexity of order $2^{42.5}$ parity checks. The complexity for the method above is much lower than the complexity for finding $\mathbf{I}_{\mathrm{s}}$ and will therefore have little effect on the overall runtime for a full attack.

7 Conclusion

We have proposed a new key recovery correlation attack on irregularly clocked keystream generators where the stream is filtered by a nonlinear Boolean function. Our attack uses a correlation property of Boolean functions, that gives higher correlation than previous methods. Thus, we need fewer equations to succeed. The property holds even if the function is correlation immune. Using this property together with the iteration techniques from [11] we get a low runtime and low memory complexity algorithm for attacking the model. The algorithm outputs the initialization bits $\mathbf{I}_{\mathrm{s}}$ for LFSR$_{\mathrm{s}}$. Knowing $\mathbf{I}_{\mathrm{s}}$ there exist previous algorithms which can determine $\mathbf{I}_{\mathrm{u}}$ efficiently.

Acknowledgment

We would like to thank Matthew Parker, John Erik Mathiassen and the anonymous referees for many helpful comments.

References

1. V. Chepyzhov, T. Johansson, and B. Smeets. A simple algorithm for fast correlation attacks on stream ciphers. In *Fast Software Encryption, FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 181–195. Springer-Verlag, 2001.
2. Philippe Chose, Antoine Joux, and Michel Mitton. Fast correlation attacks: An algorithmic point of view. In *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 209–221. Springer-Verlag, 2002.
3. Nicolas Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology-CRYPTO' 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 176–194, 2003.
4. Nicolas Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In *Advances in Cryptology - EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 345–359, 2003.
5. J.D Golić. Computation of low-weight parity-check polynomials. *Electronic Letters*, october 1996. 32(21):1981-1982.
6. Thomas Johansson and Fredrik Jönsson. Theoretical analysis of a correlation attack based on convolutional codes. In *Proceedings of 2000 IEEE International Symposium on Information Theory*, IEEE Trans. Comput., page 212, 2000.
7. Fredrik Jönsson and Thomas Johansson. A fast correlation attack on LILI-128. In *Inf. Process. Lett. 81(3)*, pages 127–132, 2002.
8. Sabine Leveiller, Gilles Zémor, Philippe Guillot, and Joseph Boutros. A new cryptanalytic attack for pn-generators filtered by a Boolean function. In *Selected Areas in Cryptography: 9th Annual International Workshop, SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, 2003.
9. M. Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology-EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer-Verlag, 1994.
10. W. Meier and O. Staffelbach. Fast correlation attacks on stream ciphers. In *Advances in Cryptology-EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 301–314. Springer-Verlag, 1988.
11. Håvard Molland. Improved linear consistency attack on irregularly clocked keystream generators. In *Fast Software Encryption, FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 109 – 126. Springer-Verlag, 2004.
12. Håvard Molland, John Erik Mathiassen, and Tor Helleseth. Improved fast correlation attack using low rate codes. In *Cryptography and Coding, IMA 2003*, volume 2898 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2003.
13. W.T. Penzhorn and G.J Kuhn. Computation of low-weight parity checks for correlation attacks on stream ciphers. In *Cryptography and Coding, IMA 1995*, volume 1025 of *Lecture Notes in Computer Science*, pages 74–83. Springer-Verlag, 1995.
14. Markku-Juhani Olavi Saarinen. A time-memory tradeoff attack against LILI-128. In *Fast Software Encryption, FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 231–236, 2002.
15. T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Trans. on Computers*, C-34:81–85, 1985.
16. L. Simpson, E. Dawson, J. Golic, and W. Millan. LILI keystream generator. In *SAC'2000*, volume 2012 of *Lecture Notes in Computer Science*, pages 231–236. Springer-Verlag, 2002. Available at http://www.isrc.qut.edu.au/lili.
17. David Wagner. A generalized birthday problem. In *Advances in cryptology-CRYPTO' 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303, 2002.

18. Xian-Mo Zhang and Yuliang Zheng. The nonhomomorphicity of Boolean functions. In *Selected Areas in Cryptography, SAC 98*, volume 1556 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 1998.

Appendix

A Implementation Details

To reach the runtime complexity $O(2^{l_s} \cdot m)$ and memory complexity down to $N + 128m$ bits, the implementation of the algorithm has some tricks. Since not all of these tricks are obvious we give more detailed descriptions of them below.

A.1 Runtime Details

Sliding window In Lemma 2 we get $\mathbf{v}^{i+1}$ by among other things deleting the c_0^i first bits of $\mathbf{v}^i$. This is done using the sliding window technique, which means that we move the viewing to the right instead of shifting the whole sequence to the left. This way the shifting can be done in just a couple of operations. To avoid heavy use of memory, we slide the window over an array of fixed length N, so that the entries that become free at the beginning of the array are reused. Thus, the left and right indexes of the sliding window after i iterations will be

$$(left, right) = (i \bmod N, i + N_i \bmod N),$$

where $N{>}N_i$, for all i, $0 \le i < 2^{l_s}$.

The same sliding window technique is also used on the equation set when equations are deleted and added to the equation set.

Updating the indexes In Lemma 2 every pointer z_{t+1} in $\mathbf{v}^*$ is replaced with z_t for every $0 \le t \le M$, which would take M operations. If we skip the replacements we note that after i iterations the entry z_t in $\mathbf{v}^*$ will become z_{t+i}. It is also important to note that when we write $\mathbf{v} = (\dots, z_0, \dots, z_t, \dots, z_M, \dots)$, the entries $z_0, \dots, z_t, \dots, z_M$ are pointers from $\mathbf{v}^*$ to $\mathbf{z}$. They are not the actual key bits. Thus, in the implementation we do not replace z_t with z_{t-1}. However, when we after i iterations in the search for equations find an equation $v_k^i + v_{k+j_1}^i + \cdots + v_{k+j_{w-1}}^i = 0$ that is defined, we replace the corresponding equation $z_{t_1} + z_{t_2} + \cdots + z_{t_w}$ with $z_{t_1-i} + z_{t_2-i} + \cdots + z_{t_w-i}$, to compensate.

Reducing the memory access time When we test an equation we must use pointers to pointers to the keystream. Then each equation test will have high memory access time. We can reduce this significantly by testing the equations on 32 states simultaneously. This is possible since the next state $\mathbf{I}_s^{i+1}$ is tested by shifting all the equations one entry to the left over $\mathbf{z}$. We can now take the bits

$z_{t_a}, z_{t_a+1}, \ldots, z_{t_a+31}$ for each of the term $1 < a \leq 4$ in the equations and put them into 32 bit registers. Now we can test the states and add one to the metrics of the states that satisfy the equation. This speeds up the runtime by a factor of approximately 20.

A.2 Memory Details

Reducing the use of memory Instead of storing all the pointers, we set 1 in $\mathbf{v}^i$ where the bits are defined and 0 otherwise. When we search in $\mathbf{v}^i$ to find entries where the equation λ_{v} is defined, we keep track of where in $\mathbf{z}$ the four terms in λ_{v} points to by counting the number of 1's we pass during the search. This is done for each of the 4 terms in the equation λ_{v}. This way we always know where in $\mathbf{z}$ the given equation of $\mathbf{v}^i$ points to. Using this trick the number of memory bits needed during an attack is reduced from $32N + 128m$ bits to

$$N + 128m$$

Implementing this trick will not affect the runtime of the attack.

Linear Properties in the Klimov-Shamir T-function

Håvard Molland and Tor Helleseth

The Selmer Centre[**]
Department of Informatics, University of Bergen, N-5020 Bergen, Norway.
H. Molland is currently a visiting Scholar at
Information Security Research Centre, Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001 Australia

Abstract. Linear equations have always been powerful tools in cryptanalysis. In this paper we present a general linear equation in the binary alphabet of minimum weight 3 that holds for all state lengths n and all shifts i of sequences generated by the T-function $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n$ proposed by Klimov and Shamir. It is surprising that these linear properties exist, and they indicate that the T-functions are not as 'wild' and non-algebraic as claimed by Klimov and Shamir. We also use the equation to propose a simple algebraic attack on cryptographic T-functions.

Key words: Stream cipher, sequences, T-function, linear property

1 Introduction

The objective of stream ciphers is to expand a short key into a long keystream that is difficult to distinguish from a truly random stream. The encryption is done by xoring the plaintext with the keystream, and it should not be possible to reconstruct the key from the keystream.

In many years linear feedback shift registers, *LFSR*s, have been one of the most important building blocks in keystream generators. The advantage with LFSRs is that they can easily be designed to produce maximum length streams, and they are fast and easy to implement in hardware. However, the LFSRs have a lot of linear properties, which make them easy to cryptanalyze and break. To make the LFSRs more secure they must be combined with other elements, such as S-boxes or Boolean functions. This complicates and slows down the ciphers in software.

Recently the T-function

$$\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n \tag{1}$$

[**] This work was supported by the Norwegian Research Council under Grant 146874/420.

was proposed by Klimov and Shamir in [2,1], as a building block for stream ciphers. The operation $\vee$ is the bitwise *or* operation, and $\mathbf{x}_i$ is a natural number for $0 \leq \mathbf{x}_i < 2^n$. The authors claim that the generator is non-algebraic due to the bitwise or operation. The sequence $\{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots\}$ generated by (1) has a single cycle of maximum length 2^n, the sequence has high linear complexity, and the generator is highly efficient in software. Thus, it may become an important building block in future ciphers. In [3] Klimov and Shamir construct T-functions of multiword states to get beyond the 32 or 64 bit limits in computers, making the generator more practical. It is still important to analyze the simple single word T-function, since it can be used as a building block in multiword T-functions.

Since the use of T-functions in cryptography are so recent, not much is known about their cryptographic properties and strength. Therefore, it is important to analyze every aspect of the generator. In this paper we present a linear equation that holds over all sequences generated by the single word T-function (1). The property indicates that the T-functions are not as non-algebraic as first claimed. The equation is given by

$$x_{i,j} + x_{i+2^{j-1},j} + x_{i,j-1} + a_2 x_{i,1} + a_1 x_{i,0} + a_0 = 0 \bmod 2$$

where $x_{i,j}$ is bit number j in the internal state $\mathbf{x}_i$ of the generator at time i, and the constants a_0, a_1, a_2 are determined by the constant $\mathbf{C}$ in the generator (1). We have implemented and carefully tested the results, and the tests confirm that the equation is correct. The details can be found in Theorem 1 in Section 4.

We have organized the paper as follows; in Section 2 we give a brief overview of T-functions, and we show some important previous known properties. In Section 3 we present four new properties, which are the building blocks for the proof of the linear property we present in Section 4. In Section 4.2 we show how the theorem works on the T-function with $\mathbf{C} = 5$ and we present a new attack on that generator.

2 Overview of T-functions

2.1 The Notation

Let $x_{i,j} \in F_2$, where F_2 is the finite field with two elements. In this paper we let

$$\mathbf{x}_i = x_{i,n-1}2^{n-1} + x_{i,n-2}2^{n-2} + \cdots + x_{i,j}2^j + \cdots + x_{i,1}2 + x_{i,0}$$

be the internal state of the T-function at time i. We can view the inner state as an integer $\mathbf{x}_i$, $0 \leq \mathbf{x}_i < 2^n$, or as a table of bits $\mathbf{x}_i = (x_{i,n-1}, x_{i,n-2}, \dots, x_{i,j}, \dots, x_{i,0})$, where $x_{i,j}$ is the bit number j in the inner state $\mathbf{x}_i$ at time i. Similarly we let $\mathbf{C} = C_{n-1}2^{n-1} + \cdots + C_2 2^2 + C_1 2 + C_0$, for $C_j \in F_2$.

The $\vee$ in $\mathbf{c} = \mathbf{a} \vee \mathbf{b} \bmod 2^n$ is the primitive *bitwise or* function between $\mathbf{a}$ and $\mathbf{b}$, where $0 \leq \mathbf{a}, \mathbf{b} < 2^n$. We let the $\oplus$ in $c = a \oplus b$ notate the primitive *xor*

Table 1. An example of the T-function $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n$ with $n = 4$, $\mathbf{C} = (0, 1, 0, 1) = 5$ and initialization state $\mathbf{x}_0 = (0, 0, 0, 0) = 0$

	3	2	1	0
$\mathbf{x}_0$	0	0	0	0
$\mathbf{x}_1$	0	1	0	$\underline{1}$
$\mathbf{x}_2$	0	0	1	|0
$\mathbf{x}_3$	0	1	$\underline{1}$	|1
$\mathbf{x}_4$	1	1	|0	0
$\mathbf{x}_5$	0	0	|0	1
$\mathbf{x}_6$	0	1	|1	0
$\mathbf{x}_7$	1	$\underline{0}$	|1	1
$\mathbf{x}_8$	1	|0	0	0
$\mathbf{x}_9$	1	|1	0	1
$\mathbf{x}_{10}$	1	|0	1	0
$\mathbf{x}_{11}$	1	|1	1	1
$\mathbf{x}_{12}$	0	|1	0	0
$\mathbf{x}_{13}$	1	|0	0	1
$\mathbf{x}_{14}$	1	|1	1	0
$\mathbf{x}_{15}$	$\underline{0}$	|1	1	1

operation between the bits a and b, or equivalent $c = a + b \bmod 2$. For simplicity we will sometimes let $\sum_{i=0}^{k-1} a_i \bmod 2$ be denoted by $\bigoplus_{i=0}^{k-1} a_i$, where $a_i \in F_2$. We arrange the sequence in an $l \times n$ matrix $\mathbf{x}$, where l is the length of the sequence, and n is the length of the inner state of the generator, as shown in Table 1. With this arrangement the i'th row in $\mathbf{x}$ is the inner state of the generator at time i.

2.2 T-function Basics

Let $\mathbf{x}_i = x_{i,n-1}2^{n-1} + \cdots + x_{i,2}2^2 + x_{i,1}2^1 + x_{i,0}$ be the n bit internal state of the T-function. In [2,1] the n bit T-function is defined as a function $f(\mathbf{x}_i)$ where the j'th bit of output of $\mathbf{y}_i = f(\mathbf{x}_i)$ is only dependent on the bits $(j, j-1, \ldots, 1, 0)$ in the n bit input $\mathbf{x}_i$, and independent of the bits $(n-1, n-2, \ldots, j+1)$, for $0 \leq j < n$. They define a parameter function as a function $\mathbf{y}_i = r(\mathbf{x}_i)$ where the j'th bit in $\mathbf{y}_i$ is only dependent on the bits $(j-1, j-2, \ldots, 1, 0)$ in $\mathbf{x}_i$. They show that the function

$$\mathbf{x}_i = f(\mathbf{x}_{i-1}) = r(\mathbf{x}_{i-1}) + \mathbf{x}_{i-1} \bmod 2^n \tag{2}$$

has a single cycle of maximum length 2^n if $r(\mathbf{x}_i)$ is an even parameter function. We refer to [1] for the detailed description of even parameters. Further on they show that when the bits 0 and 2 in $\mathbf{C}$ are 1, the function $\mathbf{x}_{i-1}^2 \vee \mathbf{C}$ is an even parameter and that the function $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n$ has a single cycle of maximum length.

Next we show some important properties of the T-functions.

1. Column j in $\mathbf{x}$ has period 2^{j+1}. That is $x_{i,j} = x_{i+2^{j+1},j}$.
2. The effective period of column j is 2^j, since $x_{i,j} = x_{i+2^j,j} \oplus 1$ (See Corollary 1 in this paper).
3. If $m < n$ the n-bit T-function 'contains' all the m-bit T-functions. That is, if we generate an n bit sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{2^n-1}$ and an m bit sequence $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{2^n-1}$ with $\mathbf{x}_0 = \mathbf{y}_0 \bmod 2^m$, then $\mathbf{x}_i = \mathbf{y}_i \bmod 2^m$, $0 \leq i < 2^n$.

2.3 Another Proof for Single Cycle of Maximum Length

This result is also proved in both [1] and [2]. Here we give a different proof based on [1,2], since we use these techniques in the new lemmas and theorems later in the paper.

We will show by induction that the n bit T-function generates a sequence with a single cycle of maximum length 2^n if and only if $C_0 = 1$ and $C_2 = 1$. We do this by proving that if the T-function with state length $n = m$ generates a sequence of maximum length 2^m, then the sequence generated by the T-function with state length $n' = m + 1$ generates a sequence of maximum length 2^{m+1}. It is easy to check that the T-function has a maximum length $2^3 = 8$ for the base case $n_0 = 3$.

As noted in [1] $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^m = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-2}^2 \vee \mathbf{C} + \mathbf{x}_{i-2} = \mathbf{x}_0 + \sum_{k=0}^{i-1} \mathbf{x}_k^2 \vee \mathbf{C}$. Assume $\mathbf{x}_0 = \mathbf{0}$ and let the induction hypothesis be that $\mathbf{x}_i \bmod 2^m$ has one maximum cycle of length 2^m. Note that bit m in the $m+1$ bit function $\mathbf{x}_i^2 \bmod 2^{m+1}$ only depends on the bits $0, 1, \ldots, m-1$ and is independent of the bit m in $\mathbf{x}_i$. It follows that $\mathbf{x}_i^2 \vee \mathbf{C} \bmod 2^{m+1} = (\mathbf{x}_i \bmod 2^m)^2 \vee \mathbf{C} \bmod 2^{m+1}$. Since the sequence ($\mathbf{x}_0 \bmod 2^m$, $\mathbf{x}_1 \bmod 2^m$, … , $\mathbf{x}_{2^m-1} \bmod 2^m$) has a single cycle of length 2^m, it is a permutation of $(0, 1, \ldots, 2^m - 1)$ [1,2]. Thus, we can write

$$\begin{aligned} \mathbf{x}_{2^m} &= \mathbf{x}_0 + \sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+1} \\ &= \sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+1} \\ &= \sum_{k=0}^{2^m-1} k^2 \vee \mathbf{C} \bmod 2^{m+1}. \end{aligned} \tag{3}$$

The $\vee\mathbf{C}$ only gives the contribution 2^j to the sum if and only if bit j in k^2 is 0 and bit j in $\mathbf{C}$ is 1. In the set $\{0^2, 1^2, \ldots, k^2, \ldots, (2^m - 1)^2\}$, using the probabilities given in the maximum length Theorem in [2], the number of entries with bit $j \leq m+1$ equal to zero is exactly

$$D_j = \begin{cases} 2^{m-1}(1+2^{-\lfloor \frac{j}{2} \rfloor}) & m+1 \geq j \geq 1 \\ 2^{m-1} & j=0 \end{cases} . \tag{4}$$

Let $\mathbf{C} = C_0 + 2C_1 + 2^2C_2 + \cdots + 2^{m+1}C_{m+1}$. We can now substitute the $\vee\mathbf{C}$ (the bitwise or $\mathbf{C}$) with an algebraic expression.

$$\begin{aligned} &\sum_{k=0}^{2^m-1} k^2 \vee \mathbf{C} \qquad (5)\\ &= \sum_{k=0}^{2^m-1} k^2 + \sum_{j=0}^{m+1} C_j D_j 2^j \\ &= \sum_{k=0}^{2^m-1} k^2 + C_0 2^{m-1} + C_1 2^{m+1} + \sum_{j=2}^{m+1} C_j 2^j 2^{m-1}(1+2^{-\lfloor \frac{j}{2} \rfloor}) \\ &= \sum_{k=0}^{2^m-1} k^2 + C_0 2^{m-1} + C_1 2^{m+1} + \sum_{j=2}^{m+1} C_j 2^{j+m-1} + \sum_{j=2}^{m+1} C_j 2^{m-1+\lceil \frac{j}{2} \rceil}. \qquad (6) \end{aligned}$$

It is well known that $\sum_{k=0}^{q} k^2 = \frac{q^3}{3} + \frac{q^2}{2} + \frac{q}{6}$, for all q. For $q = 2^m - 1$ it can be proved that

$$\sum_{k=0}^{2^m-1} k^2 = 2^{m-1} + 2^m \bmod 2^{m+2} \tag{7}$$

Using (4) we see that $2^2 D_2 = 2^{m+1} + 2^m$, and that $2^j D_j = 0 \bmod 2^{m+1}$, when $j > 2$. Recall that $\mathbf{x}_0 = \mathbf{0}$. Using the formulas above we get

$$\begin{aligned} \mathbf{x}_{2^m} &= \mathbf{x}_0 + \sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+1} \\ &= \sum_{k=0}^{2^m-1} k^2 + C_0 2^{m-1} + C_1 2^{m+1} + C_2 2^{m+1} + C_2 2^m \\ &+ \overbrace{\sum_{j=3}^{m} C_j 2^{j+m-1}}^{=0 \bmod 2^{m+1}} + \overbrace{\sum_{j=3}^{m} C_j 2^{m-1+\lceil \frac{j}{2} \rceil}}^{=0 \bmod 2^{m+1}} \bmod 2^{m+1} \\ &= 2^{m-1} + 2^m + C_0 2^{m-1} + C_2 2^m \bmod 2^{m+1} \\ &= (1+C_0)2^{m-1} + (1+C_2)2^m \bmod 2^{m+1}. \end{aligned}$$

Recall that we now are looking at a T-function sequence with state length $n' = m+1$, and that the results are based on that the $n = m$ bit T-function sequence has a single cycle of maximum length 2^n. It is easy to check that the only possible values of C_0 and C_2 that gives $\mathbf{x}_{2^{n'-1}} = 2^{n'} \bmod 2^{n'}$ are $C_0 = 1$ and $C_2 = 1$. If $\mathbf{x}_{2^{n'-1}} = 2^{n'-1} \neq \mathbf{x}_0 = 0 \bmod 2^{n'}$, then the sequence $\mathbf{x}_i$ has minimum length $2^{n'-1} + 1$. But the only possible lengths are powers of 2[1], and the cycle can not have period greater than $2^{n'}$. Thus, we have proved by induction that the $n' = m+1$ bit T-function sequence has one maximum length cycle of length $2^{n'}$, if and only if $C_0 = 1$ and $C_2 = 1$ for $\mathbf{x}_0 = 0$. The bits $C_1, C_3, \ldots, C_{n'-1'}$ can be arbitrary bits. Most often these bits are set to zero so that $\mathbf{C} = 5$. Since a single maximum length cycle involves all possible internal states, the generator will produce maximum length sequences for any $\mathbf{x}_0$.

Property 2 is a direct consequence of Theorem 2 in [1] and the proof above. However the property does not seem to be recognized and discussed by Klimov and Shamir in [1,2,3]. We use this property in several proofs in this paper, and we believe it is so important that we state it in a Corollary.

Corollary 1. *Let* $\mathbf{x}$ *be a sequence generated by* $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1}$ *mod* 2^n, *with* $C_0 = C_2 = 1$. *Then* $x_{i,j} = x_{i+2^j,j,} \oplus 1$ *for all* i, j *and* $\mathbf{x}_0$.

Proof. It follows directly from the proof for maximum length single cycle. Let $\mathbf{x}$ be the n bit T-function sequence with $\mathbf{x}_0 = 0$. Thus, as shown above, we know that $\mathbf{x}_{2^{n-1}} = \sum_{k=0}^{2^{n-1}-1} \mathbf{x}_k^2 \vee \mathbf{C} = 2^{n-1} \bmod 2^n$. For an arbitrary $\mathbf{x}_0$ we get that

$$\begin{aligned}
\mathbf{x}_{2^{n-1}} &= \mathbf{x}_0 + \sum_{k=0}^{2^{n-1}-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^n \\
&= \mathbf{x}_0 + 2^{n-1} \bmod 2^n \\
&= x_{0,0} + x_{0,1}2 + \cdots + x_{0,n-2}2^{n-2} + (x_{0,n-1}+1)2^{n-1} \bmod 2^n.
\end{aligned}$$

Since the sequence for n contains all j bit T-function sequences for $j < n$, it follows that the j'th bit of $\mathbf{x}_{2^{n-1}}$ is given by $x_{i,j} = x_{i+2^j,j} \oplus 1$.

We know that column j in $\mathbf{x}$ has period 2^{j+1}. Using the lemma we see that the second half of the full period column is just a complement of the first half. It follows that the effective period of the n bit T-function is 2^{n-1} and not 2^n.

3 Preliminary Properties

First we state some new properties that we need to prove the linear equation theorem in Section 4. We start by showing that the states $\mathbf{x}_i$ and $\mathbf{x}_{i+2^{n-2}}$ have an interesting connection with the quadratic expression $\sum_{k=0}^{2^{j-1}-1} x_{i+k,0}x_{i+k,j-1} \bmod 2$.

Lemma 1. *Let* $\mathbf{x}$ *be a sequence generated by* $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1}$ *mod* 2^n*, for* $n \geq 5$*. Let* $C_0 = C_2 = 1$*, and let the rest of the bits in* $\mathbf{C}$ *be arbitrary. Then*

$$\mathbf{x}_{i+2^{n-2}} = \mathbf{x}_i + g_{i,n-1}(\mathbf{x})2^{n-1} + 2^{n-2} \textit{ mod } 2^n, \tag{8}$$

where

$$g_{i,j}(\mathbf{x}) = C_1 + C_3 + C_4 + (C_j + 1) \sum_{k=0}^{2^{j-1}-1} x_{i+k,0}x_{i+k,j-1} \textit{ mod } 2,$$

for $j = n - 1$*.*

Proof. Let the state length be $n = m + 2$, for an arbitrary $m \geq 3$. Since $\mathbf{x}^2 \bmod 2^{m+2}$ is independent of bit $m+1$, we can write that $\mathbf{x}_i^2 = ((\mathbf{x}_i \bmod 2^m) + \mathbf{x}_{i,m}2^m)^2 \bmod 2^{m+2}$. Using the simple rule $(A+B)^2 = A^2 + 2AB + B^2$ we get

$$\begin{aligned}
&\sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+2} \\
&= \sum_{k=0}^{2^m-1} (\mathbf{x}_k \bmod 2^m + x_{k,m}2^m)^2 \vee \mathbf{C} \bmod 2^{m+2} \\
&= \sum_{k=0}^{2^m-1} ((\mathbf{x}_k \bmod 2^m)^2 + 2(\mathbf{x}_k \bmod 2^m)x_{k,m}2^m + \overbrace{x_{k,m}2^{2m}}^{=0 \bmod 2^{m+2}}) \vee \mathbf{C} \bmod 2^{m+2} \\
&= \sum_{k=0}^{2^m-1} ((\mathbf{x}_k \bmod 2^m)^2 + (\mathbf{x}_k \bmod 2^m)x_{k,m}2^{m+1}) \vee \mathbf{C} \bmod 2^{m+2}.
\end{aligned}$$

Since $2^{m+1}\sum_{k=0}^{2^m-1}(\mathbf{x}_k \bmod 2^m)x_{k,m} \bmod 2^{m+2}$ only gives contribution 2^{m+1} when $\sum_{k=0}^{2^m-1}(\mathbf{x}_k \bmod 2^m)x_{k,m}$ is odd and zero otherwise, only the least significant bit in $\mathbf{x}_k$ counts. Thus, we can set

$$2^{m+1}\sum_{k=0}^{2^m-1}(\mathbf{x}_k \bmod 2^m)x_{k,m} \bmod 2^{m+2} = 2^{m+1}\sum_{k=0}^{2^m-1} x_{k,0}x_{k,m} \bmod 2^{m+2}.$$

Now we get

$$\begin{aligned}
&\sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+2} \\
&= \sum_{k=0}^{2^m-1} \left((\mathbf{x}_k \bmod 2^m)^2 + x_{k,0}x_{k,m}2^{m+1} \right) \vee \mathbf{C} \bmod 2^{m+2} && (9) \\
&= \sum_{k=0}^{2^m-1} \left((\mathbf{x}_k^2 \bmod 2^m) \vee (\mathbf{C} \bmod 2^{m+1}) \right) && (10) \\
&\quad + \sum_{k=0}^{2^m-1} (x_{k,0}x_{k,m}2^{m+1} \vee (2^{m+1}C_{m+1})) \bmod 2^{m+2}.
\end{aligned}$$

The step from (9) to (10) requires some comments. Normally we cannot say that $\mathbf{a} \vee (\mathbf{c} + \mathbf{d}) = \mathbf{a} \vee \mathbf{c} + \mathbf{a} \vee \mathbf{d} \bmod 2^m$ holds for all $\mathbf{a}, \mathbf{b}, \mathbf{c}$. For example $1 \vee (1 + 1) = 1 \neq 0 = 1 \vee 1 + 1 \vee 1 \bmod 2$. However the $(m + 1)$'th bit in the sum $\sum_{k=0}^{2^m-1} (\mathbf{x}_k \bmod 2^m)^2 \bmod 2^{m+2}$ is always zero since $D_{m+1} = 0 \bmod 2$, and gives no contribution to the $(m + 1)$'th bit in (9). Furthermore the sum $\sum_{k=0}^{2^m-1} x_{k,0}x_{k,m}2^{m+1}$ gives no contribution to the bits $(0, 1, \ldots, m)$ in (9). Thus, in each bit in the two sums the case is similar to $\begin{smallmatrix} a_0 \\ a_1 \end{smallmatrix} \vee (\begin{smallmatrix} b_0 \\ 0 \end{smallmatrix} \oplus \begin{smallmatrix} 0 \\ d_1 \end{smallmatrix}) = \begin{smallmatrix} a_0 \vee b_0 \\ a_1 \vee d_1 \end{smallmatrix}$, which make the step possible.

Since $C_0 = C_2 = 1$ and according to the Equations (5) and (3), we can substitute $\sum_{k=0}^{2^m-1} (\mathbf{x}_k \bmod 2^m)^2 \vee \mathbf{C}$ with

$$\sum_{k=0}^{2^m-1} k^2 \vee \mathbf{C} = \sum_{k=0}^{2^m-1} k^2 + 2^{m-1} + 2^m + (1 + C_1 + C_3 + C_4)2^{m+1} \bmod 2^{m+2}. \quad (11)$$

Next, if $C_{m+1} = 1$ then $\sum_{k=0}^{2^m-1} (x_{k,0}x_{k,m}2^{m+1} \vee (C_{m+1}2^{m+1})) = \sum_{k=0}^{2^m-1} 2^{m+1} = 0 \bmod 2^{m+1}$. If $C_{m+1} = 0$, we get that $2^{m+1} \sum_{k=0}^{2^m-1} (x_{k,0}x_{k,m} \vee (C_{m+1}2^{m+1})) = \sum_{k=0}^{2^m-1} x_{k,0}x_{k,m}2^{m+1}$. Thus, we have that

$$\begin{aligned}
&\sum_{k=0}^{2^m-1} (x_{k,0}x_{k,m}2^{m+1} \vee (C_{m+1}2^{m+1})) && (12) \\
&= (C_{m+1} + 1)2^{m+1} \sum_{k=0}^{2^m-1} x_{k,0}x_{k,m} \bmod 2^{m+2}.
\end{aligned}$$

Using (11) and (12) in (10) we get

$$\begin{aligned}&\sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+2}\\ =&\sum_{k=0}^{2^m-1} k^2 + (C_{m+1}+1)2^{m+1}\sum_{k=0}^{2^m-1} x_{k,0}x_{k,m} + 2^{m-1}\\ &+2^m + (1+C_1+C_3+C_4)2^{m+1} \bmod 2^{m+2}.\end{aligned}$$

Using (7) we get

$$\begin{aligned}&= (C_{n+1}+1)2^{m+1}\sum_{k=0}^{2^m-1} x_{k,0}x_{k,m} + 2^m + 2^{m-1} + 2^{m-1}\\ &\quad +2^m + (1+C_1+C_3+C_4)2^{m+1} \bmod 2^{m+2}\\ &= 2^{m+1}(C_{m+1}+1)\sum_{k=0}^{2^m-1} x_{k,0}x_{k,m}\\ &\quad +2^m + (C_1+C_3+C_4)2^{m+1} \bmod 2^{m+2}.\end{aligned}$$

At last we have that

$$\begin{aligned}\mathbf{x}_{2^m} &= \mathbf{x}_0 + \sum_{k=0}^{2^m-1} \mathbf{x}_k^2 \vee \mathbf{C} \bmod 2^{m+2}\\ &= \mathbf{x}_0 + 2^m + (C_1 + C_3 + C_4 + (C_{m+1}+1)\sum_{k=0}^{2^m-1} x_{k,0}x_{k,m})2^{m+1} \bmod 2^{m+2}. \quad (13)\end{aligned}$$

Since (13) holds for all $\mathbf{x}_0$, it will hold for all shifts i of the sequence, and we get

$$\mathbf{x}_{i+2^m} = \mathbf{x}_i + 2^m + g_{i,m+1}(\mathbf{x})2^{m+1} \bmod 2^{m+2}$$

which gives

$$\mathbf{x}_{i+2^{n-2}} = \mathbf{x}_i + 2^{n-2} + g_{i,n-1}(\mathbf{x})2^{n-1} \bmod 2^n,$$

when $j = n-1$ and

$$g_{i,j}(\mathbf{x}) = C_1 + C_3 + C_4 + (C_j+1)\sum_{k=0}^{2^{j-1}-1} x_{i+k,0}x_{i+k,j-1}.$$

This completes the proof.

The previous lemma gives an interesting connection between the states $\mathbf{x}_i$ and $\mathbf{x}_{i+2^{n-2}}$. However, we are more interested in connections between the bits in the sequence.

Lemma 2. *Let $n \geq 5$ and $C_0 = C_2 = 1$. Let $\mathbf{x}$ be the stream generated by $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1}$ mod 2^n, for any initialization state $\mathbf{x}_0$ and any $C_1, C_3, C_4, \ldots, C_{n-1}$. Then*

$$x_{i+2^{j-1},j} + x_{i,j} = x_{i,j-1} + g_{i,j}(\mathbf{x}) \ mod\ 2,\ 4 \leq j \leq n-1,$$

where

$$g_{i,j}(\mathbf{x}) = C_1 + C_3 + C_4 + (C_j + 1) \sum_{k=0}^{2^{j-1}-1} x_{k+i,0} x_{k+i,j-1} \ mod\ 2$$

Proof. From Lemma 1 we have that

$$\mathbf{x}_{2^{n-2}} = \mathbf{x}_0 + g_{0,n-1}(\mathbf{x}) 2^{n-1} + 2^{n-2} \bmod 2^n,$$

where $g_{0,n-1}(\mathbf{x}) = (C_1 + C_3 + C_4 + (C_{n-1} + 1) \sum_{k=0}^{2^{n-2}-1} x_{k,0} x_{k,n-2}) \bmod 2$. We rewrite $\mathbf{x}_0$ to the equivalent

$$\mathbf{x}_0 = (\mathbf{x}_0 \bmod 2^{n-2}) + x_{0,n-2} 2^{n-2} + x_{0,n-1} 2^{n-1} \bmod 2^n,$$

and we get

$$\begin{aligned} \mathbf{x}_{2^{n-2}} &= (\mathbf{x}_0 \bmod 2^{n-2}) + x_{0,n-2} 2^{n-2} && (14) \\ &\quad + x_{0,n-1} 2^{n-1} + g_{0,n-1}(\mathbf{x}) 2^{n-1} \\ &\quad + 2^{n-2} \bmod 2^n \\ &= (\mathbf{x}_0 \bmod 2^{n-2}) + (x_{0,n-2} + 1) 2^{n-2} && (15) \\ &\quad + (g_{0,n-1}(\mathbf{x}) + x_{0,n-1}) 2^{n-1} \bmod 2^n. \end{aligned}$$

When $x_{0,n-2} = 1$ we get $(x_{0,n-2} + 1) 2^{n-2} = 2 \cdot 2^{n-2} = 2^{n-1}$, which adds 1 to the coefficient for 2^{n-1}, and the resulting coefficient $(x_{0,n-2} + 1)$ for 2^{n-2} will be zero. Therefore, we get

$$\begin{aligned} \mathbf{x}_{2^{n-2}} &= (\mathbf{x}_0 \bmod 2^{n-2}) + (x_{0,n-2} \oplus 1) 2^{n-2} \\ &\quad + (g_{0,n-1}(\mathbf{x}) + x_{0,n-2} + x_{0,n-1}) 2^{n-1} \bmod 2^n. \end{aligned}$$

This gives

$$x_{2^{n-2},n-1} = g_{0,n-1}(\mathbf{x}) + x_{0,n-2} + x_{0,n-1} \bmod 2. \tag{16}$$

Here we have stated the lemma for bit number $j = n-1$. Since the sequence $\mathbf{x}_i$ mod 2^n contains all the sub sequences $\mathbf{x}_i$ mod 2^m, $0 \leq m < n$, it will hold for all bits j, $4 \leq j < n$, and we get the lemma by substituting $n-1$ with j.

Let $B_{i,j}(\mathbf{x})$ be defined by $B_{i,j}(\mathbf{x}) = \bigoplus_{k=0}^{2^j-1} x_{i+k,0}x_{i+k,j}$. In the next lemma we show that the sequence given by $(B_{0,j}(\mathbf{x}), B_{1,j}(\mathbf{x}), \ldots, B_{i,j}(\mathbf{x}), \ldots)$, for all j, is a shift of the second column $(x_{0,1}, x_{1,1}, \ldots, x_{i,1}, \ldots)$ in $\mathbf{x}$. This observation indicates that we can replace $\bigoplus_{k=0}^{2^{j-1}-1} x_{k+i,0}x_{k+i,j-1}$ in $g_{i,j}(\mathbf{x})$ with $x_{i,1}$.

Lemma 3. *For any initialization state $\mathbf{x}_0$, state length $n \geq 5$ and $C_0 = C_2 = 1$, let $\mathbf{x}$ be the stream generated by $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1}$ mod 2^n. Let $B_{i,j}(\mathbf{x})$ be defined by $B_{i,j}(\mathbf{x}) = \bigoplus_{k=0}^{2^j-1} x_{i+k,0}x_{i+k,j}$. Then the bit stream*

$$(B_{0,j}(\mathbf{x}), B_{1,j}(\mathbf{x}), \ldots, B_{i,j}(\mathbf{x}), \ldots)$$

has a period of 4 and is a cyclic shift of the bit stream $(x_{0,1}, x_{1,1}, \ldots) = (\ldots, 0, 0, 1, 1, 0, 0, \ldots)$ which also has period 4.

Proof. Let $\mathbf{y}$ be the T-function sequence $(\mathbf{y}_0, \mathbf{y}_1, \ldots)$ with initialization state $\mathbf{y}_0 = \mathbf{0}$. Since the sequence has full period we have $y_{2^{n-2}+k,0} = y_{k,0}$, and from Corollary 1 we have $y_{2^j+i,j} = y_{i,j} \oplus 1$. This gives the following calculations,

$$\begin{aligned}
B_{i+1,j}(\mathbf{y}) &= \bigoplus_{k=1+i}^{2^j+i} y_{k,0}y_{k,j} \\
&= \bigoplus_{k=1+i}^{2^j+i-1} y_{k,0}y_{k,j} \oplus \overbrace{y_{2^j+i,0}}^{y_{i,0}} \cdot \overbrace{y_{2^j+i,j}}^{y_{i,j}\oplus 1} \\
&= \bigoplus_{k=1+i}^{2^j+i-1} y_{k,0}y_{k,j} \oplus y_{i,0}(y_{i,j} \oplus 1) \\
&= y_{i,0}y_{i,j} \oplus \bigoplus_{k=1+i}^{2^j+i-1} (y_{k,0}y_{k,j}) \oplus y_{i,0} \\
&= B_{i,j}(\mathbf{y}) \oplus y_{i,0}.
\end{aligned}$$

Since $y_{i,0} \oplus y_{i+1,0} = 1$ for all i, we get that

$$\begin{aligned}
B_{i+2,j}(\mathbf{y}) &= B_{i+1,j}(\mathbf{y}) \oplus y_{i+1,0} \\
&= B_{i,j}(\mathbf{y}) \oplus y_{i,0} \oplus y_{i+1,0} \\
&= B_{i,j}(\mathbf{y}) \oplus 1.
\end{aligned}$$

We see that we can generate the sequence

$$(\, B_{0,j}(\mathbf{y})\,,\, B_{1,j}(\mathbf{y})\,,\, B_{2,j}(\mathbf{y}), \ldots)$$

by $B_{i+2,j}(\mathbf{y}) = B_{i,j}(\mathbf{y}) \oplus 1$, and with initialization $(\,B_{0,j}(\mathbf{y})\,,\,B_{1,j}(\mathbf{y})\,) = (\,B_{0,j}(\mathbf{y})$, $B_{0,j}(\mathbf{y}))$ since $y_{0,0} = 0$. Thus, $B_{i,j}(\mathbf{y})$, $i \geq 0$ will generate a cyclic shift of the bitstream

$$(\dots, 0, 0, 1, 1, 0, 0, 1, 1, \dots)$$

of period 4 which is a cyclic shift of $(x_{0,1}, x_{1,1}, x_{2,1}, \dots)$ for all $\mathbf{x}_0$.

Next we show that for a given initialization state $\mathbf{x}_0$, the quadratic expression will be the same for all $j \geq 4$, that is $B_{i,j}(\mathbf{x}) = B_{i,j-1}(\mathbf{x})$, $4 \leq j < n$.

Lemma 4. *For any initialization state $\mathbf{x}_0$, state length $n \geq 5$ and $C_0 = C_2 = 1$, let $\mathbf{x}$ be the stream generated by $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n$. Then*

$$B_{i,j}(\mathbf{x}) = B_{i,j-1}(\mathbf{x}),\ \textit{for all } i,\ \textit{for } 4 \leq j < n.$$

Proof. Let $j = n-1$. Since $x_{k,0} = x_{k+2^{n-2},0}$, we rewrite the quadratic expression to

$$\begin{aligned} B_{0,n-1}(\mathbf{x}) &= \bigoplus_{k=0}^{2^{n-1}-1} x_{k,0}x_{k,n-1} \\ &= \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}x_{k,n-1} \oplus \bigoplus_{k=0}^{2^{n-2}-1} x_{k+2^{n-2},0}x_{k+2^{n-2},n-1} \\ &= \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}(x_{k,n-1} \oplus x_{k+2^{n-2},n-1}). \end{aligned}$$

Now using Lemma 2 for $j = n-1$ and $i = k$, we get $x_{k,n-1} \oplus x_{k+2^{n-2},n-1} = x_{k,n-2} \oplus g_{k,n-1}(\mathbf{x})$ and

$$\begin{aligned} B_{0,n-1}(\mathbf{x}) &= \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}(x_{k,n-1} \oplus x_{k+2^{n-2},n-1}) \\ &= \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}(x_{k,n-2} \oplus g_{k,n-1}(\mathbf{x})) \\ &= \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}x_{k,n-2} \oplus \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}g_{k,n-1}(\mathbf{x}) \end{aligned}$$

We show in Appendix B that $\bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}g_{k,n-1}(\mathbf{x}) = 0$, $n \geq 5$. Since this hold for all initialization states $\mathbf{x}_0$, we have that $B_{i,n-1}(\mathbf{x}) = B_{i,n-2}(\mathbf{x})$. Finally, since the sequence $\mathbf{x}_k \bmod 2^n$ contains all sequences mod 2^m, $m \leq n$, we get the lemma by substituting $n-1$ with j.

4 The Main Results

4.1 The Linear Equation

We proved in the previous section that the bits $x_{i,j}$, $x_{i+2^{j-1},j}$ and $x_{i,j-1}$ have an interesting connection with the quadratic expression $B_{i,j-1}(\mathbf{x})$. We proved in Lemma 3 that the sequence generated by $B_{i,j-1}(\mathbf{x})$, for all $i \geq 0$ is just a cyclic shift of the bit sequence $x_{i,1}$, for all $i \geq 0$. We show in the next theorem that the shift (depending on the value of $\mathbf{C}$) is given by adding $x_{i,0}$ and a constant term a_0. Thus, we can replace the quadratic expression $B_{i,j-1}(\mathbf{x})$ by a simple linear combination of $x_{i,0}$ and $x_{i,1}$. This gives main result in this paper.

Theorem 1. *Let $n \geq 5$, let $\mathbf{x}$ be the stream generated by $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1}$ mod 2^n, let $C_0 = C_2 = 1$ and let $C_1, C_3, C_4, \ldots, C_{n-1}$ be arbitrary bits. Then the linear equation*

$$x_{i,j} + x_{i+2^{j-1},j} + x_{i,j-1} + a_2 x_{i,1} + a_1 x_{i,0} + a_0 = 0 \ mod\ 2,$$

always holds for all i when $4 \leq j < n$, where $a_0 = C_3C_j + C_j + C_1 + C_4 + 1$, $a_1 = C_1(C_j + 1)$ and $a_2 = C_j + 1$.

Proof. Let $(\mathbf{y}_0, \mathbf{y}_1, \ldots)$ be the sequence generated by the T-function (1) with initialization state $\mathbf{y}_0 = 0$. We can easily check that column 1 of $\mathbf{y}$, that is $(y_{0,1}, y_{1,1}, \ldots)$, can be generated by $y_{i+2,1} = y_{i,1} \oplus 1$, with initialization $(y_{0,1}, y_{1,1}) = (0, C_1)$. Let $B_{i,j}$ be defined as in Lemma 3. Recall from the lemma that $B_{i,j}(\mathbf{y})$ can be generated by $B_{i+2,j}(\mathbf{y}) = B_{i,j}(\mathbf{y}) \oplus 1$, with initialization $(B_{i,0}(\mathbf{y}), B_{i,0}(\mathbf{y}))$. By adding the two bitstreams $B_{i,n-1}(\mathbf{y})$ and $y_{i,1}$ we get

$$B_{i,n-1}(\mathbf{y}) \oplus y_{i,1} = B_{0,n-1}(\mathbf{y}) \oplus C_1 y_{i,0} \tag{17}$$

which gives

$$B_{i,n-1}(\mathbf{y}) \oplus y_{i,1} \oplus C_1 y_{i,0} = B_{0,n-1}(\mathbf{y}), \text{ for all } i.$$

Since $\mathbf{x}$ is a cyclic shift of $\mathbf{y}$ we know there exists an i such that $\mathbf{y}_i = \mathbf{x}_0$, and we get

$$B_{0,n-1}(\mathbf{x}) \oplus x_{0,1} \oplus C_1 x_{0,0} = B_{0,n-1}(\mathbf{y}).$$

This gives

$$B_{0,n-1}(\mathbf{x}) = B_{0,n-1}(\mathbf{y}) \oplus x_{0,1} \oplus C_1 x_{0,0}, \tag{18}$$

and finally using (18) with Lemma 2 for $j = n - 1$ we get

$$x_{2^{n-2},n-1} \oplus x_{0,n-1} = g_{0,n-1}(\mathbf{x}) \oplus x_{0,n-2}, \tag{19}$$

where

$$\begin{aligned} g_{0,n-1}(\mathbf{x}) &= C_1 \oplus C_3 \oplus C_4 \oplus (C_{n-1} \oplus 1) \bigoplus_{k=0}^{2^{n-2}-1} x_{k,0}x_{k,n-2} \\ &= C_1 \oplus C_3 \oplus C_4 \oplus (C_{n-1} \oplus 1)B_{0,n-2}(\mathbf{x}) \\ &= C_1 \oplus C_3 \oplus C_4 \oplus (C_{n-1} \oplus 1)(B_{0,n-2}(\mathbf{y}) \oplus x_{0,1} \oplus C_1x_{0,0}). \end{aligned}$$

Assume $C_{n-1} = 0$ and recall that $\mathbf{y}_0 = 0$, which gives $y_{0,n-2} = 0, y_{0,n-1} = 0$, $y_{0,1} = 0$ and, $y_{0,0} = 0$. Then from (19) we get

$$y_{2^{n-2},n-1} = g_{0,n-1}(\mathbf{y}) = B_{0,n-2}(\mathbf{y}) \oplus y_{0,1} \oplus y_{0,0}C_1 \oplus C_1 \oplus C_3 \oplus C_4$$

and therefore

$$B_{0,n-2}(\mathbf{y}) = y_{2^{n-2},n-1} \oplus C_1 \oplus C_3 \oplus C_4. \tag{20}$$

Lemma 4 says that $B_{i,n}(\mathbf{x}) = B_{i,n-1}(\mathbf{x})$ for all $n > 3$. Thus, we will get $B_{0,n-1}(\mathbf{y})$ with $C_{n-1} = 0$ for all $n \geq 5$ by calculating $B_{0,3}(\mathbf{y})$. We show in Appendix A that $y_{8,4} = C_1 \oplus 1$, when $\mathbf{y}_0 = 0$. Since $C_{n-1} = C_4 = 0$, we get from (20) that

$$B_{0,3}(\mathbf{y}) = y_{8,4} \oplus C_1 \oplus C_3 = 1 \oplus C_3.$$

When $C_{n-1} = 1$, $B_{0,n-2}(\mathbf{y})$ has no impact on $g_{0,n-1}(\mathbf{x})$ and we get $g_{0,n-1}(\mathbf{x}) = C_1 \oplus C_3 \oplus C_4$. Thus, for $C_{n-1} \in \{0, 1\}$

$$\begin{aligned} g_{0,n-1}(\mathbf{x}) &= C_1 \oplus C_3 \oplus C_4 \oplus (C_{n-1} \oplus 1)(1 \oplus C_3 \oplus x_{0,1} \oplus C_1x_{0,0}) \\ &= C_1 \oplus C_4 \oplus C_{n-1} \oplus C_{n-1}C_3 \oplus 1 \\ &\quad \oplus (C_{n-1}C_1 \oplus C_1)x_{0,0} \oplus (C_{n-1} \oplus 1)x_{0,1}. \end{aligned}$$

Since the sequence $\mathbf{x}_i \bmod 2^n$ contains all sequences generated by the m-bit functions $\mathbf{x}_i \bmod 2^m$ for $m < n$, we substitute $n-1$ with j, $4 \leq j < n$ and we let $a_0 = C_1 \oplus C_4 \oplus C_j \oplus C_jC_3 \oplus 1$, $a_1 = C_jC_1 \oplus C_1$ and $a_2 = C_j \oplus 1$. Now $g_{0,j}(\mathbf{x})$ in Lemma 2 is given by

$$g_{0,j}(\mathbf{x}) = a_0 + a_1x_{0,0} + a_2x_{0,1} \bmod 2.$$

Because the sequence is cyclic and since the equation holds for all initialization states $\mathbf{x}_0$, the equation will hold for all shifts i of the sequence $\mathbf{x}$. By this we get

$$x_{i,j} + x_{i+2^{j-1},j} + x_{i,j-1} + a_2x_{i,1} + a_1x_{i,0} + a_0 = 0 \bmod 2,$$

for all i and $4 \leq j < n$. This completes the proof.

4.2 Applications of Theorem 1

The equation for $\mathbf{C} = 5$. For a secure generator we want $\mathbf{C}$ to have as low Hamming weight as possible. The reason is simple; when a bit $C_j = 1$ in $\mathbf{C}$ we also know that bit j in the expression $\mathbf{x}_i^2 \vee \mathbf{C}$ is 1. Thus, the higher Hamming weight $\mathbf{C}$ has, the more we know about the output of $\mathbf{x}_i^2 \vee \mathbf{C}$. In the worst case scenario $\mathbf{C} = 2^n - 1 = (1, 1, 1, 1, \dots, 1)$ and we get that $\mathbf{x}_i^2 \vee (2^n - 1) = 2^n - 1 \bmod 2^n$ for all $\mathbf{x}_i$. The resulting T-function (1) is actually equivalent to the very insecure function $\mathbf{x}_i = \mathbf{x}_{i-1} - 1 \bmod 2^n$. The most interesting value for cryptographic applications is $\mathbf{C} = 5$, since that is the lowest possible Hamming weight $\mathbf{C}$ can have and still give a maximum length cycle. Therefore, we will in this section analyze the T-function with $\mathbf{C} = 5$.

Corollary 2. *Let $n \geq 4$ and let $\mathbf{x}_i$ be the stream generated by $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee 5 + \mathbf{x}_{i-1}$ mod 2^n. Then the following linear equation*

$$x_{i,j} + x_{i+2^{j-1},j} + x_{i,j-1} + x_{i,1} + 1 = 0 \ mod\ 2, \tag{21}$$

holds for all $i \geq 0$ and $3 \leq j < n$.

Proof. This follows directly from Theorem 1 with $C = 5$ and $n \geq 5$. It is easy to test that the lemma also holds for $j = 3$ for all i.

The theoretical cipher based on T-functions. The T-function is meant as a basic building block for ciphers and is not meant to be a cipher on its own. However, to show the cryptographically strength of the T-functions, Klimov and Shamir proposed a theoretical cipher in [3] using only a single T-function.

Let $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ be a sequence generated by the T-function $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee 5 + \mathbf{x}_{i-1} \bmod 2^n$. Let the m most significant bits $(x_{i,n-1}, x_{i,n-2}, \dots, x_{i,n-m})$ from each $\mathbf{x}_i$ be output as keystream words. The bits $(x_{i,n-m-1}, x_{i,n-m-2}, \dots, x_{i,0})$ are secret, and $\mathbf{K} = (x_{0,n-m-1}, x_{0,n-m-2}, \dots, x_{0,0})$ is the secret key of size $l = n - m$.

If we know the whole $\mathbf{x}_i$, we can use an algorithm from [1] to generate backward from $\mathbf{x}_i$ to $\mathbf{x}_0$. In a known plaintext attack we assume that we know the bits $(x_{i,n-1}, x_{i,n-2}, \dots, x_{i,n-m})$, for $0 \leq i < L$, where L is the length of the known keystream. Thus, the knowledge of $(x_{i,l-1}, \dots, x_{i,0})$ for any $i < L$ will give $\mathbf{x}_i$ and the key $\mathbf{K}$. The objective for a cryptanalyst is therefore to determine the unknown bits $(x_{i,l-1}, x_{i,l-2}, \dots, x_{i,0})$ for some $i < L$.

An attack on the theoretical cipher using the linear equation. Assume we have a cipher described as above, with $\mathbf{C} = \mathbf{5}$, state length n and a secret key $\mathbf{K}$ of length l, and assume we know $L = 2^l + 1$ m words $(x_{i,n-1}, x_{i,n-2}, \dots, x_{i,n-m,})$ of the keystream. Recall that $l = n - m$. The keystream will then have a full period of 2^n, and we know the bits $x_{i,l}$ $x_{i,l+1}$ for $0 \leq i \leq L$. Now we can simply use Corollary 2 to calculate $x_{0,1}$ and $x_{1,1}$ by $x_{i,1} = x_{i,l+1} + x_{i+2^l,l+1} + x_{i,l} + 1$ mod

2, for $0 \leq i \leq 1$. Knowing $x_{0,1}$ and $x_{1,1}$ we easily get $x_{0,0}$ by $x_{0,0} = x_{0,1} \oplus x_{1,1}$, and we can generate $x_{i,0}$ and $x_{i,1}$ for all $i \geq 0$. Since we also know $x_{i,l}$ we can use Corollary 2 to calculate $x_{0,l-1}$ by $x_{0,l-1} = x_{0,l} + x_{2^{l-1},l} + x_{0,1} + 1 \bmod 2$. Now we know 3 of the secret key bits using calculations that actually can be done by hand. However the attack is theoretical since we need a vast amount of keystream data.

We can continue to use this technique to calculate $(x_{0,l-2}, x_{0,l-3}, \ldots, x_{0,2})$. Unfortunately the amount of work doubles for each new bit we try to calculate since we need two bits of column j in $\mathbf{x}$ to get one bit in column $j-1$. Therefore, the total work to find all the l keybits will be $O(2^l)$, which is the same as exhaustive search.

We can get around this by only calculating the bits $(x_{0,l-1}, x_{0,l-2}, \ldots, x_{0,l-\lfloor l/2 \rfloor})$ which will have runtime $O(2^{l/2})$. Since there is no guessing involved in this algorithm, we now know these bits. Next we can simply do an exhaustive search for the rest of the bits with runtime $O(2^{l/2})$. The total runtime for this attack will then be $O(2^{l/2}) + O(2^{l/2}) = O(\sqrt{2^l})$.

Attacks on future ciphers. The cipher above is not very practical since normal computers can not efficiently square more than 32 or in some cases 64 bits. Therefore, the T-function must be used together with other building blocks, or several T-functions may be combined. The linear equations we have proposed in this paper may be more powerful or use less keystream bits, if applied on ciphers where the sequence generated by the T-functions has been filtered or combined by other functions. Especially, if for efficiency, the lower bits are not discarded. Traces of linearity have a tendency to survive the filtering of nonlinear functions. Thus, the property may be used to construct distinguishing attacks, guess and determining attacks, or even correlation attacks on such ciphers. If there are traces of some of the less significant bits after the filtering, these attacks may be powerful and use relatively few bits from the keystream.

5 Summary

We have found a general linear equation over F_2 that always holds over all sequences generated by the T-function $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n$. It is important to analyze every aspect of the generator, since it may become an important building block in the design of stream ciphers. The linear property indicates that there are more structures in the sequences than claimed by Klimov and Shamir. We have shown how the equation can be used to attack and reconstruct the initialization state of the sequence, if parts of the sequence are known. Further on, the equation may be used as a mathematical tool to prove new weaknesses in the generator in the future.

References

1. Alexander Klimov and Adi Shamir. Cryptographic applications of T-functions. In *SAC*, 2003.
2. Alexander Klimov and Adi Shamir. A new class of invertible mappings. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 470 – 483. Springer-Verlag, 2003.
3. Alexander Klimov and Adi Shamir. New cryptographic primitives based on multiword T-functions. In *Fast Software Encryption, FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 1 – 15. Springer-Verlag, 2004.

A Proof of $y_{8,4} = C_1 \oplus 1$ when $C_4 = 0$ and $\mathbf{y}_0 = 0$

We have that $C_4 = 0$ and recall that C_2 and C_0 is always set to 1. Thus, in this setting $\mathbf{C} = (0, C_3, 1, C_1, 1) = 5 + C_3 2^3 + C_1 2$ can only have the 4 values 5,7,13 and 15. We will prove that $y_{8,4} = C_1 \oplus 1$, by simply generate $\mathbf{y}_8 = y_{8,4}2^4 + \cdots + y_{8,1}2 + y_{8,0}$ with $\mathbf{y}_i = \mathbf{y}_{i-1}^2 \vee 5 + \mathbf{y}_{i-1}$, $\mathbf{y}_0 = 0$, for $\mathbf{C} \in \{5, 7, 13, 15\}$. It is easy to see from the table

$\mathbf{C}$	C_3	C_1	$y_{8,4}$
5	0	0	1
7	0	1	0
13	1	0	1
15	1	1	0

that $y_{8,4} = C_1 \oplus 1$.

B Proof of $\sum_{k=0}^{2^{n-2}-1} x_{k,0} g_{k,n-1}(\mathbf{x}) = 0 \bmod 2$

We know that $g_{k,n-1}(\mathbf{x}) = C_1 + C_3 + C_4 + (C_j + 1)B_{i,n-2}(\mathbf{x})$. Since $x_{k,0}$, $0 \leq k < 2^{n-2}$ has even number of ones and zeros we have that $\sum_{k=0}^{2^{n-2}-1} x_{k,0}(C_1+C_3+C_4) = 0 \bmod 2$.

If $C_j = 1$, then $g_{k,n-1}(\mathbf{x}) = C_1 + C_3 + C_4 \in \{0, 1\}$ and we get that

$$\sum_{k=0}^{2^{n-2}-1} x_{k,0} g_{k,n-1}(\mathbf{x}) = 0$$

mod 2. If $C_j = 0$ then

$$\sum_{k=0}^{2^{n-2}-1} x_{k,0} g_{k,n-1}(\mathbf{x}) = \sum_{k=0}^{2^{n-2}-1} x_{k,0} B_{k,n-2}(\mathbf{x}) \bmod 2.$$

From Lemma 3 we have that $B_{k,n-2}(\mathbf{x})$ is a cyclic shift of $(1,1,0,0,1,1,0,0,\dots)$ of period 4 and we know that $x_{k,0}$ is a cyclic shift of $(0,1,0,1,\dots)$ of period 2. It follows that $x_{k,0}B_{k,n-2}(\mathbf{x})$ has period 4 and

$$\sum_{k=0}^{2^{n-2}-1} x_{k,0}B_{k,n-2}(\mathbf{x}) = 2^{n-4}\sum_{k=0}^{3} x_{k,0}B_{k,n-2}(\mathbf{x}) = 0 \bmod 2,$$

$n \geq 5$, since $2^x y \bmod 2 = 0$ for all y and $x \geq 1$.

Algebraic Structures over the Binaries for the Klimov-Shamir T-function

Håvard Molland

The Selmer Centre**
Department of Informatics, University of Bergen, N-5020 Bergen, Norway.
H. Molland is currently a visiting Scholar at
Information Security Research Centre, Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001 Australia

Abstract. Recently the Klimov and Shamir proposed a T-function that can be used to generate word based sequences of maximum length. They claim the function is non-algebraic due to the non-arithmetic "or" operation. In this, paper we present a simple algorithm that constructs the multivariate functions $g_{i,j}(\mathbf{x}_0)$ for all bits j in the word at time i in the sequence generated by the Klimov-Shamir T-function. The functions give the complete algebraic structure of the sequences for given state length n. By analyzing these functions, we have found several important properties of the T-function sequence, which we conjecture hold for all n. The problems of proving the claimed properties are open and should be objectives for further research.

1 Introduction

A triangular function (T-function) $\mathbf{y} = f(\mathbf{x})$ is a function where the j'th least significant bit y_j in the output $\mathbf{y}$ is only dependent on the bits $x_j, x_{j-1}, \ldots, x_0$ in the input. The sequence generator

$$\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n \tag{1}$$

based on the T-function $f(\mathbf{x}) = \mathbf{x}^2 \vee \mathbf{C} + \mathbf{x} \bmod 2^n$ was recently proposed by Klimov and Shamir in [3,2]. The operation $\vee$ is the bitwise *"or"* operation, $\mathbf{x}_i$ is a natural number for $0 \leq \mathbf{x}_i < 2^n$, and n is the number of bits in the internal state of the generator. If $C \equiv 5, 7 \bmod 8$, the word based sequence $\{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots\}$ generated by (1) has a single cycle of maximum length 2^n, the sequence has high linear complexity, and the generator is efficient in software when properly implemented. The authors claim that the generator is non-algebraic due to the bitwise "or" operation. Thus, it may become an important building block in future stream ciphers, where the objective is to produce long and pseudo random sequences. In Table 1 on the following page we have generated the sequence for $n = 4$ and initialization state $\mathbf{x}_0 = 0$.

** This work was supported by the Norwegian Research Council under Grant 146874/420.

Table 1. An example of the sequence generated by the T-function $\mathbf{x}_i = \mathbf{x}_{i-1}^2 \vee \mathbf{C} + \mathbf{x}_{i-1} \bmod 2^n$ with $n = 4$, $\mathbf{C} = (0,1,0,1) = 5$ and initialization state $\mathbf{x}_0 = (0,0,0,0) = 0$

	3	2	1	0
$\mathbf{x}_0$	0	0	0	0
$\mathbf{x}_1$	0	1	0	$\underline{1}$
$\mathbf{x}_2$	0	0	1	$\vert 0$
$\mathbf{x}_3$	0	1	$\underline{1}$	$\vert 1$
$\mathbf{x}_4$	1	1	$\vert 0$	0
$\mathbf{x}_5$	0	0	$\vert 0$	1
$\mathbf{x}_6$	0	1	$\vert 1$	0
$\mathbf{x}_7$	1	$\underline{0}$	$\vert 1$	1
$\mathbf{x}_8$	1	$\vert 0$	0	0
$\mathbf{x}_9$	1	$\vert 1$	0	1
$\mathbf{x}_{10}$	1	$\vert 0$	1	0
$\mathbf{x}_{11}$	1	$\vert 1$	1	1
$\mathbf{x}_{12}$	0	$\vert 1$	0	0
$\mathbf{x}_{13}$	1	$\vert 0$	0	1
$\mathbf{x}_{14}$	1	$\vert 1$	1	0
$\mathbf{x}_{15}$	$\underline{0}$	$\vert 1$	1	1

In much of the previous analysis of the T-function generator, the internal state $\mathbf{x}_i$ has been treated as a number mod 2^n. The "or" operation $\vee$ can be perceived as a non-arithmetic operation over the natural numbers. However, it is a highly arithmetic operation over the binaries, since $a \vee b = a \cdot b + a + b \bmod 2$. In this paper we view $\mathbf{x}_i$ solely as a vector $(x_{i,n-1}, x_{i,n-2}, \ldots, x_{i,0})$ where $\mathbf{x}_i = x_{i,0} + 2x_{i,1} + 2^2 x_{i,2} + \cdots + 2^{n-1} x_{i,n-1}$ and $x_{i,j}$ is in $\{0,1\}$. This way we get that $\mathbf{a} \vee \mathbf{b} = (a_{n-1}b_{n-1} + a_{n-1} + b_{n-1}, \ldots, a_0 b_0 + a_0 + b_0)$.

The objective for this paper is to generate and analyze the multivariate functions $x_{i,j} = g_{i,j}(\mathbf{x}_0)$ from the initialization state $\mathbf{x}_0 = (x_{0,n-1}, x_{0,n-2}, \ldots, x_{0,0})$ to each bit $x_{i,j}$ in the sequence. An algorithm that generates $g_{i,j}(\mathbf{x}_0)$ for all i and j is a powerful tool for analyzing the structure of the sequences. For bit streams generated by linear feedback shift registers, the linear functions for the bits are given by the generator matrix G, such that the sequence $\mathbf{u}$ can be generated by $\mathbf{u} = \mathbf{u}^{\mathrm{I}} G$ where $\mathbf{u}^{\mathrm{I}}$ is the initialization state. To get the multivariate functions over the binary variables for the Klimov-Shamir T-function, we must calculate $\mathbf{x}^2 \vee \mathbf{C} + \mathbf{x} \bmod 2^n$ by finding the multivariate functions for each bit in the outputs of the square, "or" and plus operations when the input is represented by a binary vector.

By analyzing these functions, we have identified several interesting and useful properties. However, we have not yet proved them, which should be the objective for further research. We summarize the main results in the following conjectures:

1. The equation

$$x_{i+2^{j-2}} = x_{i,0}x_{i,1} + x_{i,1}x_{i,j-2} + x_{i,1}x_{i,j-1} + x_{i,j-2}x_{i,j-1} + x_{i,j-2} + x_{i,j-1} + x_{i,j}$$

of algebraic degree 2 holds for the bits $0 \le j < n$ for all sequences generated by the T-function (1) with constant $\mathbf{C} = 5$.

2. It follows directly from property 1 that the autocorrelation $\Pr(x_{i,j} = x_{i+2^{j-2},j}) = 0.375$ holds. We have tested this autocorrelation for all $j < 32$, which gives evidence for that property 1 holds for all j.
3. We also show other autocorrelations which indicates other structures in the sequence.
4. When can arranging the functions in a form similar to the algebraic normal form (ANF), and let row number i in a $2^{j+1} \times (2^j + 1)$ matrix be the coefficients for the functions for bits in column j at time i in the sequence (see the tables 5 and 6 in Appendix B). Using this arrangement we identify several cyclic and recursive properties for the terms in the functions.

2 Generating the Multivariate Functions

Let $\mathbf{x} = (x_{n-1}, x_{n-2}, \ldots, x_0)$, $\mathbf{y} = (y_{n-1}, y_{n-2}, \ldots, y_0)$ and $\mathbf{y} = f(\mathbf{x}) = \mathbf{x}^2 \vee \mathbf{C} + \mathbf{x} \bmod 2^n$. Let $y_j = g_j(\mathbf{x})$ be the Boolean function from $\mathbf{x}$ to bit number j in the output $\mathbf{y}$ such that $\mathbf{y} = (\, g_{n-1}(\mathbf{x}), g_{n-2}(\mathbf{x}), \ldots, g_0(\mathbf{x})\,)$. We will now present a simple algorithm that outputs the function $g_j(\mathbf{x})$ for all $j < n$.

First we calculate $\mathbf{d} = \mathbf{x}^2 \bmod 2^n$ where $\mathbf{d} = (d_{n-1}, d_{n-2}, \ldots, d_0)$. This is shown in Figure 1 where the rows are summed mod 2^n. Recall that this involves

$$\begin{array}{lllllll}
 & x_0x_{n-1} & \cdots & \cdots & \cdots & x_0x_1 & x_0 \\
 & x_1x_{n-2} & \cdots & \cdots & x_1 & x_1x_0 & \\
 & x_2x_{n-3} & \cdots & x_2x_1 & x_2x_0 & & \\
 & \vdots & & & & & \\
 & x_{n-1}x_0 & & & & & \\
\hline
+ & & & & & & \\
= & d_{n-1} & \cdots & \cdots & \cdots & d_1 & d_0 \\
\hline\hline
\end{array}$$

Fig. 1. Squaring mod 2^n

carry bits which also must be calculated. The algorithm for this is of course trivial when we know the hard values for $x_{n-1}, \ldots, x_1, x_0$. However, we want the exact algebraic expression for all the sequence bits. Therefore, we must calculate the multivariate expression for each bit in the output $\mathbf{s}$ of the operation $\mathbf{s} = \mathbf{a} + \mathbf{b} \bmod 2^n$ over the binaries as shown in Figure 2.

It is easy to see that the carry bit c_j equals 1 if the integer sum $c_{j-1} + a_{j-1} + b_{j-1}$ equals 2 or 3. Thus, the carry bits are given by $c_j = (c_{j-1} \& a_{j-1}) \vee (c_{j-1} \& b_{j-1}) \vee (a_{j-1} \& b_{j-1})$ where the binary "or" operation $\vee$ can be expressed algebraically as

$$\begin{array}{r} \\ a_{n-1} \dots a_2\, a_1\, a_0 \\ +\, b_{n-1} \dots b_2\, b_1\, b_0 \\ \hline = s_{n-1} \cdots s_2\, s_1\, s_0 \\ \hline\hline \end{array} \quad \Rightarrow \quad \begin{array}{r} c_{n-1} \dots c_2\, c_1\; 0 \\ a_{n-1} \dots a_2\, a_1\, a_0 \\ \oplus\, b_{n-1} \dots b_2\, b_1\, b_0 \\ \hline = s_{n-1} \cdots s_2\, s_1\, s_0 \\ \hline\hline \end{array}$$

Fig. 2. Adding mod 2^n

$a \vee b = a \cdot b \oplus a \oplus b$, and the binary "and" operation & is given by multiplication, that is $a\&b = a \cdot b$. Using this expression we get that

$$\begin{aligned} c_j &= (c_{j-1}a_{j-1}) \vee (c_{j-1}b_{j-1}) \vee (a_{j-1}b_{j-1}) \\ &= (a_{j-1}b_{j-1}c_{j-1} \oplus a_{j-1}c_{j-1} \oplus b_{j-1}c_{j-1}) \vee a_{j-1}b_{j-1} \\ &= a_{j-1}b_{j-1}c_{j-1} \oplus a_{j-1}b_{j-1}c_{j-1} \oplus a_{j-1}c_{j-1} \oplus b_{j-1}c_{j-1} \oplus a_{j-1}b_{j-1} \\ &= c_{j-1}a_{j-1} \oplus c_{j-1}b_{j-1} \oplus a_{j-1}b_{j-1} \end{aligned}$$

We see that each bit j in the result $\mathbf{s}$ now can be expressed by $s_j = a_j \oplus b_j \oplus c_j$ where $c_j = c_{j-1}a_{j-1} \oplus c_{j-1}b_{j-1} \oplus a_{j-1}b_{j-1}$ and $c_0 = 0$.

Let $\mathbf{R}_i$ be the temporary results during the calculation, and let $\mathbf{r}_i$ be the row number i in Fig 1. Starting with $\mathbf{R_0} = \mathbf{r}_0$, we calculate $\mathbf{x}^2$ by $\mathbf{R}_i \leftarrow \mathbf{R}_{i-1} + \mathbf{r}_i$, for all $1 \leq i < n$, where "+" is done using the method above. If we in each step keep the algebraic expressions instead of putting in any hard values, the entry j in the result vector $\mathbf{R} = \mathbf{R}_{n-1}$ will be a function from $\mathbf{x}$ to bit j in $\mathbf{x}^2$ mod 2^n.

Table 2. Using this algorithm for $n = 5$ and $\mathbf{C} = (0, 1, 0, 1) = 5$ we get the following result for the first 3 elements $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \dots)$ in the sequence. For simplicity we let the initialization state $(x_{0,4}, x_{0,3}, \dots, x_{0,0})$ be denoted as $(x_4, x_3, \dots, x_0)$

$\mathbf{x}_0$	$\mathbf{x}_1$	$\mathbf{x}_2$	$\cdots$	$\mathbf{x}_{16}$
x_0	$x_0 + 1$	x_0	$\cdots$	x_0
x_1	$x_0 + x_1$	$x_1 + 1$		
x_2	$x_0x_1 + x_2 + 1$	$x_1 + x_2$	$\cdots$	x_2
x_3	$x_0x_1x_2 + x_0x_2 + x_2 + x_3$	$x_1x_2 + x_0 + x_1 + x_2 + x_3$	$\cdots$	x_3
x_4	$x_0x_1x_2x_3 + x_0x_1x_2 + x_0x_2x_3 + x_0x_1$ $+x_0x_2 + x_1x_2 + x_0x_3 + x_2x_3 + x_2 + x_4$	$x_0x_1x_2 + x_1x_2x_3 + x_1x_2 + x_0x_3$ $+x_1x_3 + x_2x_3 + x_1 + x_3 + x_4$	$\cdots$	x_4

Next we must "or" the temporary result for $\mathbf{R} = \mathbf{x}^2$ with the constant $\mathbf{C}$. This is simply done by letting $R_j \leftarrow C_j \vee R_j = C_j \cdot R_j \oplus C_j \oplus R_j$ for j, $0 \leq j < n$, where R_j and C_j are the bit number j in $\mathbf{R}$ and $\mathbf{C}$. We often use hard values for $\mathbf{C}$, for example $\mathbf{C} = 5$, which is done by simply setting R_j to the hard value $R_j = 1$ if and only if $C_j = 1$, and unchanged when $C_j = 0$.

Next we add $\mathbf{x}$ to the result by $\mathbf{R} \leftarrow \mathbf{R} + \mathbf{x}$ where "+" is done by using the method above. Setting $\mathbf{x}_0 = \mathbf{x}$ and $\mathbf{x}_1 = \mathbf{R}$ we have an algebraic expression for the two first elements in the sequence $(\mathbf{x}_0, \mathbf{x}_1, \dots)$. To calculate the whole sequence, we repeat the algorithm using the algebraic expressions for $\mathbf{x}_{i-1}$ as input to calculate $\mathbf{x}_i$.

Finally we let $x_{i,j} = g_{i,j}(\mathbf{x}_0)$ be defined as the multivariate function from $\mathbf{x}_0$ to $x_{i,j}$. In Table 2 we show the functions for $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$ and $\mathbf{x}_{16}$ for $n = 4$, where $\mathbf{x}_0 = (x_{n-1}, x_{n-2}, \dots, x_0)$ for simplicity. We can for example see see that $x_{2,3} = g_{2,3}(\mathbf{x}_0) = x_1x_2 + x_0 + x_1 + x_2 + x_3$. The algorithm is shown in Algorithm 1 in Appendix A, where it is programmed in Magma[1].

3 Observed Properties

By analyzing the functions $g_{i,j}(\mathbf{x}_0)$ from the initialization bits to each bit in the sequence, we have found several interesting properties.

3.1 Algebraic Structures and Autocorrelations

The linear equation $x_{i+2^{j-1},j} = x_{i,j} + x_{i,j-1} + a_2x_{i,1} + a_1x_{i,0} + a_0 \bmod 2$ where a_0, a_1, a_2 are determined by the constant $\mathbf{C}$, was proved in [4] to hold for all sequences generated by the T-function (1) and all $\mathbf{C}$. Using Algorithm 1, this equation is easily confirmed, and we can find other similar equations. In Table 3

Table 3. The equation with algebraic degree 2 seems to hold for all i and j

j	$x_{2^{j-2},j}$
5	$x_{8,5} = x_0x_1 + x_1x_3 + x_1x_4 + x_3x_4 + x_3 + x_4 + x_5$
6	$x_{16,6} = x_0x_1 + x_1x_4 + x_1x_5 + x_4x_5 + x_4 + x_5 + x_6$
7	$x_{32,7} = x_0x_1 + x_1x_5 + x_1x_6 + x_5x_6 + x_5 + x_6 + x_7$
8	$x_{64,8} = x_0x_1 + x_1x_6 + x_1x_7 + x_6x_7 + x_6 + x_7 + x_8$
9	$x_{128,9} = x_0x_1 + x_1x_7 + x_1x_8 + x_7x_8 + x_7 + x_8 + x_9$

we have generated the functions for $x_{8,5}$, $x_{16,6}$, $x_{32,7}$, $x_{64,8}$ and $x_{128,9}$ for $\mathbf{C} = 5$. The table shows that $x_{i+2^{j-2},j}$ seems to be determined by a quadratic equation with a distinct pattern. We have not yet proved the property, so we define it in the following conjecture.

Conjecture 1. Let $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ be a sequence generated by the T-function (1), for $n \geq 6$ and $\mathbf{C} = 5$. Then

$$x_{i+2^{j-2},j} + x_{i,j} = x_{i,0}x_{i,1} + x_{i,1}x_{i,j-2} + x_{i,1}x_{i,j-1} + x_{i,j-2}x_{i,j-1} + x_{i,j-2} + x_{i,j-1} \quad (2)$$

holds for all i and $5 \leq j<n$.

By generating a similar table for $x_{i+2^{j-3},j}$ we get the following conjecture.

Conjecture 2. Let $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ be a sequence generated by the T-function (1), for $n \geq 7$ and $\mathbf{C} = 5$. Then

$$\begin{aligned} x_{i+2^{j-3},j} + x_{i,j} = {} & x_{i,0}x_{i,1}x_{i,j-3}x_{i,j-2} + x_{i,0}x_{i,1}x_{i,j-1} + x_{i,1}x_{i,j-3}x_{i,j-1} \\ & + x_{i,1}x_{i,j-2}x_{i,j-1} + x_{i,j-3}x_{i,j-2}x_{i,j-1} + x_{i,0}x_{i,2} \\ & + x_{i,j-3}x_{i,j-1} + x_{i,j-2}x_{i,j-1} + x_{i,2} + 1 \end{aligned}$$

holds for $7 \leq j < n$.

Assuming that conjecture 1 is true, we can prove that there is an autocorrelation between certain bits in the sequence. We have tested Corollary 1 for all j up to 31 which gives further evidence for that Conjecture 1 is correct.

Corollary 1. *If Conjecture 1 holds, then the sequence* $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ *for* $\mathbf{C} = 5$ *has an autocorrelation given by* $P(x_{i,j} = x_{i+2^{j-2},j}) = 0.375$

Proof. From Conjecture 1, we assume that $x_{i+2^{j-2},j} + x_{i,j} = x_{i,0}x_{i,1} + x_{i,1}x_{i,j-2} + x_{i,1}x_{i,j-1} + x_{i,j-2}x_{i,j-1} + x_{i,j-2} + x_{i,j-1}$ holds. Assume $x_{i,1} = 0$ and recall that $a \vee b = ab + a + b$, which is zero only when $a = b = 0$. We see that $x_{i+2^{j-2},j} + x_{i,j} = x_{i,j-2}x_{i,j-1} + x_{i,j-2} + x_{i,j-1} = x_{i,j-2} \vee x_{i,j-1}$, and since $\Pr(x_{i,j-2} = 0) = \Pr(x_{i,j-1} = 0) = \frac{1}{2}$ we get $\Pr(x_{i+2^{j-2},j} + x_{i,j} = 0 | x_{i,1} = 0) = \Pr(x_{i,j-2} = 0) \cdot \Pr(x_{i,j-1} = 0) = 0.25$.

Next we assume $x_{i,1} = 1$ which gives $x_{i+2^{j-2},j} + x_{i,j} = x_{i,0} + x_{i,j-2}x_{i,j-1}$. Since $x_{i,0}$, $x_{i,j-2}$ and $x_{i,j-1}$ have evenly distributions of zero and ones it is easy to see that $\Pr(x_{i+2^{j-2},j} + x_{i,j} = 0 | x_{i,1} = 1) = 0.5$. At last we have that $P(x_{i,j} = x_{i+2^{j-2},j}) = \frac{1}{2}\Pr(x_{i+2^{j-2},j} + x_{i,j} = 0 | x_{i,1} = 1) + \frac{1}{2}\Pr(x_{i+2^{j-2},j} + x_{i,j} = 0 | x_{i,1} = 0) = 0.375$. ♢

Since the functions for $x_{i,j}$ and $x_{i+2^{j-k},j}$ become very complex when $j > 10$ and $k > 2$, it is difficult to see any structure when j and k become large. However, we can use the autocorrelation to show evidence for that there are probably other algebraic functions similar to Equation (2) (probably with higher algebraic degrees) that hold for $x_{i+2^{j-k},j}$ when $k > 2$.

In table 4 we have tested the autocorrelation for different distances 2^{j-k}, $k > 2$. Since the autocorrelation $\Pr(x_{i,j} + x_{i+2^{j-k},j} = 0)$ converges to a specific value as j increments for a fixed k, it indicates that $x_{i,j} + x_{i+2^{j-k},j}$ is based on the same function for all j as long as j is greater than some threshold. Thus, for a given k we can generate the function for the threshold $j = t$ where the autocorrelation starts to converge, and then use the autocorrelation to show that the functions probably are equal for $j > t$.

Table 4. The probability $\Pr(x_{i,j} = x_{i+2^{j-k},j})$ for $j < 32$ and $1 < k < 9$

	k	2	3	4	5	6	7	8
j								
9		0.375	0.5	0.578	0.523	0.477	0.512	0.511
10		0.375	0.5	0.578	0.523	0.557	0.539	0.482
11		0.375	0.5	0.578	0.523	0.449	0.492	0.482
12		0.375	0.5	0.578	0.523	0.447	0.470	0.495
13		0.375	0.5	0.578	0.523	0.447	0.470	0.511
14		0.375	0.5	0.578	0.523	0.447	0.469	0.489
15		0.375	0.5	0.578	0.523	0.447	0.469	0.489
16		0.375	0.5	0.578	0.523	0.447	0.469	0.490
17		0.375	0.5	0.578	0.523	0.447	0.469	0.490
⋮		⋮	⋮	⋮	⋮	⋮	⋮	⋮
31		0.375	0.5	0.578	0.524	0.447	0.469	0.490

3.2 The Term Tables for $g_{i,j}(\mathbf{x})$

Let $g_j(\mathbf{x})$ as defined in Section 2 be arranged in a special algebraic form given by

$$g_j(\mathbf{x}) = x_j + \sum_{\mathbf{k}=0}^{2^j-1} a_{\mathbf{k}} x_0^{k_0} x_1^{k_1} \cdots x_{j-1}^{k_{j-1}},$$

where $(k_{j-1}, k_{j-2}, \ldots, k_0)$ is the binary representation for $\mathbf{k}$, that is $\mathbf{k} = k_0 + k_1 2 + k_2 2^2 + \cdots + k_{j-1} 2^{j-1}$. The expanded version of $g_j(\mathbf{x})$ will then be of the form

$$\begin{aligned} g_j(\mathbf{x}) = {} & a_0 + a_1x_0 + a_2x_1 + a_3x_0x_1 + a_4x_2 + a_5x_0x_2 + a_6x_1x_2 + a_7x_0x_1x_2 \\ & + a_8x_3 + \ldots + a_{15}x_0x_1x_2x_3 + a_{16}x_4 + \cdots + a_{2^j-1}x_0x_1\cdots x_{j-1} + x_j. \end{aligned} \tag{3}$$

Let $(\mathbf{x}_0, \mathbf{x}_1, \ldots)$ be the sequence generated by the T-function (1). Let $x_{i,j} = g_{i,j}(\mathbf{x}_0)$ be the function from the initialization state $\mathbf{x}_0$ to bit j in $\mathbf{x}_i$ arranged in the special algebraic form showed above. Let the table T_j be given by the coefficients for the functions $g_{i,j}(\mathbf{x}_0)$, for fixed j and $0 \le i < 2^{j+1}$, see the tables 5 and 6 in Appendix B. We can read out of the table T_3 that $x_{6,3} = x_0 + x_1 + x_1x_2 + x_3$, where for simplicity $(x_0, x_1, x_2, x_3) = (x_{0,0}, x_{0,1}, x_{0,2}, x_{0,3})$ is the initialization state for the sequence.

These tables give us an interesting insight in how the dependency between the internal state and the initialization state evolve during the generation of the sequence. The coefficient tables for the cryptographic secure functions $g_{i,j}(\mathbf{x})$ for different i and j should look random and independent. However, the tables T_2, T_3, T_4 and T_5 show that this is not the case for the T-function (1).

The first observation is the cyclic property, which we can summarize in the following conjecture.

Conjecture 3. Let $(\mathbf{x}_0, \mathbf{x}_1, \dots)$ be the sequence generated by the T-function (1). Let $a_{i,k}$ be the coefficient for term number k in $g_{i,j}(\mathbf{x}_0)$. Then the $a_{0,k}, a_{1,k}, \dots, a_{i,k}, \dots, a_{2^j-1,k}$ has period $2^{1+\lfloor \log_2(2^j-k) \rfloor}$.

This is easily seen from the tables T_2, T_3,T_4 and T_5 in Appendix B, where the end of the first period of each column is marked with an underline.

Conjecture 4. The 2^j-1 rightmost columns in T_j are equal to the 2^j-1 rightmost columns in T_{j+1}.

These conjectures indicate that there are strong patterns in the evolvement of T_j when j grows.

4 Conclusions and Future Research

We have studied the algebraic structure over the binary variables in the sequences generated by the Klimov-Shamir T-function (1). We have shown that the T-function (1) is highly arithmetic when we view the internal state as a vector over the binaries. We have identified several important properties that can be used to get better insight in how the sequences evolve. The autocorrelation properties can be used in distinguishing attacks or may even be used in key recovery attacks. In addition, the autocorrelation shows that there probably are strong algebraic structures in the sequences that hold for all bits j. A better understanding of the term tables T_j may help us in finding other linear properties similar to the one in [4].

Future research should be done on proving the conjectures we have stated in this paper and on have how to make use of them. The patterns and structures in the term tables T_j should also be studied further along with general algebraic properties of the T-function.

References

1. Computational Algebra Group, School of Mathematics and Statistics, University of Sydney, "http://magma.maths.usyd.edu.au/magma/". *The Magma Computational Algebra System.*
2. Alexander Klimov and Adi Shamir. Cryptographic applications of T-functions. In *SAC*, 2003.
3. Alexander Klimov and Adi Shamir. A new class of invertible mappings. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 470 – 483. Springer-Verlag, 2003.
4. Håvard Molland and Tor Helleseth. Linear properties in the Klimov-Shamir T-function. Submitted to IEEE Transactions on Information Theory. Accepted for presentation at ISIT-2005, 2005.

A The Algorithm

We have implemented the algorithm in the mathematical programming language Magma [1]. Note that in Magma, the entries in vectors have indexes from 1 and not 0, which means that x_0 is represented by $x[1]$. The inputs to the procedure Tfunction(x,n,C) in Algorithm 1, are the size n of the internal states, the vector $\mathbf{x}_{i-1} = (g_{i-1,n-1}(\mathbf{x}), g_{i-1,n-2}(\mathbf{x}), \ldots, g_{i-1,0}(\mathbf{x}))$, and $\mathbf{C} = (C_{n-1}, C_{n-2}, \ldots, C_3, 1, C_1, 1)$. The procedure returns $\mathbf{x}_i = (g_{i,n-1}(\mathbf{x}), g_{i,n-2}(\mathbf{x}), \ldots, g_{i,0}(\mathbf{x})$. In Algorithm 2 we show how to declare the variables and how to call the procedure properly in such a way that the algorithm works as intended.

Algorithm 1 The algorithm for generating the multivariate function for each bit in $\mathbf{x}_{t+1}$ given the n functions for the bits in $\mathbf{x}_t$

```
Tfunction:=function(x,n,C)
    /*
       Defines the function Tfunction(x,n,C)
       x: table of the $n$ input functions
       C: binary vector for the constant  n: length of vectors
       output: 'Result' is the vector of the $n$ multivariate
               functions for $\mathbf{x}^2 \vee \mathbf{C} + \mathbf{x}$ mod $2^n$
    */
    //calculate  $\mathbf{Result} \leftarrow \mathbf{x}^2$ mod $2^n$
    result:=[x[j]*x[1]:j in [1..n]]; //The first row
    for j:=1 to n-1 do
       addrow:=[0:t in [0..(i-1)]];
       addrow cat:=[x[t]*x[j+1]:t in [1..n-j]];
       result:=BinaryAlgebraAdd(result,addrow,n);
    end for;

    //Calculate $\mathbf{Result} \leftarrow \mathbf{Result} \vee \mathbf{C}$ mod $2^n$
    for j:=1 to n do
       result[j]:=result[j]*C[j]+result[j]+C[j];
    end for;

    //Calculate $\mathbf{Result} \leftarrow \mathbf{Result} + \mathbf{x}$ mod $2^n$
    result:=BinaryAlgebraAdd(result,x,n);
    //Return $\mathbf{Result} = \mathbf{x}^2 \vee \mathbf{C} + \mathbf{x}$ mod $2^n$
    return result;
end function;

BinaryAlgebraAdd:=function(a,b,n)
    /*
       Defines the function BinaryAlgebraAdd(a,b,n)
       a,b : vectors for the terms, n: length of vectors
       output: 'Sum' is the table of $n$ multivariate functions,
               one for each bit in the sum of $\mathbf{a} + \mathbf{b}$ mod $2^n$
    */
    carry:=0;
    sum:=a;
    for j:=1 to n do
       sum[j]:=carry+a[j]+b[j];
       carry:=carry*b[j]+carry*a[j]+b[j]*a[j];
    end for;
    return sum;
end function;
```

Algorithm 2 The code for using Algorithm 1.

```
n:=5;
length:=32;

//Necessary for defining F_2 in Magma
P:=PolynomialRing(GF(2),2*n,"grevlex");
u:=quo<P|[$.i^2+$.i:i in [1..2*n]]>;
V:=["x" cat IntegerToString(i) cat "" :i in [1..n]];
V:=V cat ["c" cat IntegerToString(i) cat "" :i in [1..n]];
AssignNames(~u,V);

//Declare x and C
x:=[u.i:i in [1..n]];        //Declare x[1]...x[n] as variables i F_2
C:=[u.(n+i):i in [1..n]];    //Declare C[1]...C[n] as variables i F_2
T=[x];                       //Declare T where T[1]=x
C[3]:=1;C[1]:=1;             //set the bits 0,2 in C to 1

/*
  For hard values, set C:=[*,*,...*,1,*,1] of length n
  where * are chosen variables in {0,1};
*/

//Generate and store the sequence in T
for i:=1 to length do
     print i, x;
     x:=Tfunction(x,n,C);
     T[i+1]:=x;
end for;
```

B The Term Tables

Table 5. The terms in the multivariate equation for $0 \leq i < 2^j$ for the bit sequence in column $j = 2, 3, 4$ for $\mathbf{C} = 5$

T$_2$, for $g_{i,2}(\mathbf{x})$

	0	1	2	3	4
i	1	x_0	x_1	x_0x_1	x_2
0	**0**	**0**	**0**	**0**	**1**
1	**1**	**0**	**0**	**1**	1
2	**0**	**0**	**1**	0	1
3	**1**	**1**	**1**	1	1
4	**1**	0	0	0	1
5	**0**	0	0	1	1
6	**1**	0	1	0	1
7	**0**	1	1	1	1

T$_3$, for $g_{i,3}(\mathbf{x})$

	0	1	2	3	4	5	6	7	8
i	1	x_0	x_1	x_0x_1	x_2	x_0x_2	x_1x_2	$x_0x_1x_2$	x_3
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
1	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	1
2	**0**	**1**	**1**	**0**	**1**	**0**	**1**	0	1
3	**0**	**1**	**0**	**1**	**0**	**0**	**1**	1	1
4	**1**	**0**	**1**	**0**	**1**	0	0	0	1
5	**0**	**1**	**1**	**1**	**0**	1	0	1	1
6	**0**	**1**	**1**	**0**	**0**	0	1	0	1
7	**1**	**1**	**0**	**0**	**1**	0	1	1	1
8	**1**	0	0	0	0	0	0	0	1
9	**1**	0	0	0	1	1	0	1	1
10	**1**	1	1	0	1	0	1	0	1
11	**1**	1	0	1	0	0	1	1	1
12	**0**	0	1	0	1	0	0	0	1
13	**1**	1	1	1	0	1	0	1	1
14	**1**	1	1	0	0	0	1	0	1
15	**0**	1	0	0	1	0	1	1	1

T$_4$, for $g_{i,4}(\mathbf{x})$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	1	x_0	x_1	x_0x_1	x_2	x_0x_2	x_1x_2	$x_0x_1x_2$	x_3	x_0x_3	x_1x_3	$x_0x_1x_3$	x_2x_3	$x_0x_2x_3$	$x_1x_2x_3$	$x_0x_1x_2x_3$	x_4
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
1	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	1
2	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	0	1
3	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	1	1
4	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	0	0	0	1
5	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	1	0	1	1
6	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	0	1	0	1
7	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	0	1	1	1
8	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	0	0	0	0	0	0	0	1
9	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	1	0	0	1	1	0	1	1
10	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	1	1	0	1	0	1	0	1
11	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	0	0	1	0	0	1	1	1
12	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	0	1	0	1	0	0	0	1
13	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	0	1	1	0	1	0	1	1
14	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	1	1	0	0	0	1	0	1
15	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	0	0	0	1	0	1	1	1
16	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
17	**1**	0	0	1	1	1	1	1	0	1	0	0	1	1	0	1	1
18	**1**	0	1	0	0	0	1	1	1	1	1	0	1	0	1	0	1
19	**1**	0	1	0	1	0	0	1	1	0	0	1	0	0	1	1	1
20	**0**	1	0	1	1	0	1	0	1	0	1	0	1	0	0	0	1
21	**0**	0	0	1	1	1	1	0	0	0	1	1	0	1	0	1	1
22	**0**	0	0	1	1	1	0	1	1	1	1	0	0	0	1	0	1
23	**0**	1	0	0	1	0	1	0	0	0	0	0	1	0	1	1	1
24	**0**	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
25	**0**	1	1	1	0	0	1	0	1	1	0	0	1	1	0	1	1
26	**1**	1	1	0	1	0	0	1	0	1	1	0	1	0	1	0	1
27	**1**	0	0	1	1	0	1	0	0	0	0	1	0	0	1	1	1
28	**0**	1	0	1	0	0	1	0	0	0	1	0	1	0	0	0	1
29	**1**	0	0	0	1	0	1	1	1	0	1	1	0	1	0	1	1
30	**0**	1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	1
31	**1**	1	1	0	0	0	0	1	1	0	0	0	1	0	1	1	1

Table 6. The terms in the multivariate equation for $0 \leq i < 32$ for the bit sequence in column $j = 5$

T$_5$, for $g_{i,5}(\mathbf{x})$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
i	1	x_0	x_1	x_0x_1	x_2	x_0x_2	x_1x_2	$x_0x_1x_2$	x_3	x_0x_3	x_1x_3	$x_0x_1x_3$	x_2x_3	$x_0x_2x_3$	$x_1x_2x_3$	$x_0x_1x_2x_3$	x_4	x_0x_4	x_1x_4	$x_0x_1x_4$	x_2x_4	$x_0x_2x_4$	$x_1x_2x_4$	$x_0x_1x_2x_4$	x_3x_4	$x_0x_3x_4$	$x_1x_3x_4$	$x_0x_1x_3x_4$	$x_2x_3x_4$	$x_0x_2x_3x_4$	$x_1x_2x_3x_4$	$x_0x_1x_2x_3x_4$	x_5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	0	0	1	1	0	1	1
2	1	0	0	1	1	1	1	1	0	0	1	1	1	0	1	0	1	0	1	0	0	0	1	1	1	1	1	0	1	0	1	0	1
3	1	0	1	0	1	0	0	0	1	1	0	1	1	0	1	0	1	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1	1
4	0	1	1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	0	1	0	1	0	1	0	1	0	0	0	1
5	1	1	1	0	1	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	1	1	1	0	0	0	1	1	0	1	0	1	1
6	0	1	1	0	1	1	0	1	1	1	0	1	1	1	0	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0	1	0	1
7	1	1	0	0	0	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	1	1
8	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
9	0	0	1	0	1	0	0	0	1	0	0	1	1	1	1	1	1	0	1	1	0	0	1	0	1	1	0	0	1	1	0	1	1
10	1	1	1	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	0	1	0	0	1	0	1	1	0	1	0	1	0	1
11	0	0	1	1	1	0	0	0	0	0	1	1	0	0	1	1	1	1	0	1	1	0	1	0	0	0	0	1	0	0	1	1	1
12	1	1	0	1	1	1	1	0	0	0	1	0	1	1	0	1	1	1	0	1	0	0	1	0	0	0	1	0	1	0	0	0	1
13	0	1	0	1	0	1	1	0	0	0	1	0	1	0	1	1	0	1	0	0	1	0	1	1	1	0	1	1	0	1	0	1	1
14	0	1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	1
15	1	1	1	1	1	0	0	1	1	0	0	0	1	0	1	0	1	0	1	0	0	0	0	1	1	0	0	0	1	0	1	1	1
16	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
17	1	1	1	1	1	0	0	0	0	1	1	1	0	0	1	0	1	1	0	1	1	1	1	1	0	1	0	0	1	1	0	1	1
18	1	0	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	0	1	0	1	0	1
19	1	1	1	0	0	0	0	1	0	1	0	0	1	0	0	1	0	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1	1
20	0	0	0	1	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	0	0	1
21	1	0	0	1	0	0	1	0	0	1	1	0	0	1	0	1	0	1	0	1	1	1	1	0	0	0	1	1	0	1	0	1	1
22	1	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1	1	1	1	0	0	0	1	0	1
23	0	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	1	1
24	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
25	0	0	1	1	1	0	1	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	1	1
26	1	0	1	0	0	1	1	0	1	0	1	0	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0	1	0	1
27	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	0	1	1	1
28	1	0	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	0	0	1	0	1	0	0	0	1
29	1	0	1	1	1	1	0	1	1	0	0	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0	1	1	0	1	0	1	1
30	1	0	1	0	0	0	0	0	1	1	1	1	0	0	1	0	1	1	0	1	1	1	1	1	0	1	1	0	0	0	1	0	1
31	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	0	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
63	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	0	1	1	1