Improved Linear Consistency Attack on Irregular 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 irregular 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, irregular clocked generators, linear consistency test

1 Introduction

The goal in stream ciphers is to expand a short key into a long keystream $\mathbf{z}$ that is difficult to distinguish from a truly random bit stream. It should not be possible to reconstruct the short key from $\mathbf{z}$. 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 irregular clocked linear feedback shift register ($LFSR$). This model produces bit streams with high linear complexity, which is a important criteria for pseudo random sequences.

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

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

Fig. 1. The general model for irregular 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 [?,?] 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) [?] and [?]. 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 $\mathbf{u}^* = (..,*,z_i,...,z_j,...*,...,z_k,...,*,...)$, 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 $LFSR_{\mathrm{u}}$ and $\mathbf{u}^*$. If the equation set is consistent the guess is outputted as the correct initialization bits for $LFSR_{\mathrm{s}}$. The Gaussian algorithm will use about $\frac{1}{3}l_{\mathrm{u}}^3$ calculations and the total complexity for the attack is $O(2^{l_{\mathrm{s}}} \cdot l_{\mathrm{u}}^3)$.

In [?,?] and the recent paper [?] 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 a exhaustive search on the remaining key space.

It is hard to estimate the running time for the attacks in [?,?,?,?]. The attack in [?] is estimated to have a upper bound complexity $O(L^3 \cdot 2^{L\lambda})$, where $\lambda = \log A/(1+\log A)$, $L = l_1 + l_2$ 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 $LFSR_{\mathrm{s}}$ and $LFSR_{\mathrm{u}}$ 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 $LFSR_{\mathrm{u}}$. 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 $LFSR_{\mathrm{s}}$. 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 ?? we describe a modified version of Wagners General birthday algorithm [?] 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 **??**, 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 **??**. 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 [**?**,**?**,**?**,**?**,**?**]. In Section **??** we present some simulations of the algorithm.

2 A General Model for Irregular Clocked Generators

2.1 Description

We will first give a general description of irregular 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 $c_t = D(s_v, s_{v+1}, ..., s_{v+l_\mathrm{s}-1}) \in \{a_1, a_2, ..., a_A\}, a_j \geq 0$ be a function where the input $(s_v, s_{v+1}, ..., s_{v+l_\mathrm{s}-1})$ 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 $\mathbf{u} = (u_0, u_1, ...)$ 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 $\mathbf{u} = (u_0, u_1, ..., u_{N-1})$ 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 $\mathbf{s}^\mathrm{I} = (s_0, s_1, ..., s_{l_\mathrm{s}-1})$ and $\mathbf{u}^\mathrm{I} = (u_0, u_1, ..., u_{l_\mathrm{s}-1})$ 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 a irregular clock control stream cipher as defined in Section **??**, 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 $c_t = D(s_t)$:

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

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

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

The LILI-128 Clock Control Generator. The clock control generator which is one of the building blocks in the LILI-128 cipher [**?**] 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 $(s_v, s_{v+1}, ..., s_{v+l_\mathrm{s}-1}) = (\underbrace{1, 0, ..., 0}_{y}, *, ..., *)$. Then the clocking function is defined as $D(s_v, s_{v+1}, ..., s_{v+l_\mathrm{s}-1}) = y$. 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 $c_t = D(s_{k(t)}, s_{k(t)+1}, ..., s_{k(t)+l_\mathrm{s}-1})$,where $k(t) \leftarrow k(t-1) + c_t$ for each iteration. $Q(\mathbf{c}, \mathbf{u})$ is the same as for the generators above. If we analyze the clock control sequence, $c_t \in \{1, 2, ..., ..., l_\mathrm{s} - 1\}$, 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 Irregular Clocked Generators

The idea behind the attack is to guess the clock control sequence $\mathbf{c}$, and reconstruct the original positions the key stream 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 (**??**). 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 [**?**] 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 **??**, 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 **??** we can reverse the keystream (**??**) back to (**??**) but not completely back to the original stream (**??**), since the deleted bits are not known.

The probability for a bit u^*_k being defined is $\mathrm{Prob}(u^*_k) = 1 - P_\mathrm{d}$. This happens when $k = k(t)$ holds for for some t, $0 \le t < M$. It follows that the sum $\delta = u^*_k + u^*_{k+j_1} + ... + u^*_{k+j_{w-1}}$ 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

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

3.2 Naive Attack

Using definition **??**, 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 $\mathbf{s}^\mathrm{I}$ be the initial state for $LFSR_\mathrm{s}$, and let $L^v(\mathbf{s}^\mathrm{I})$ be the inner state after v feedback shifts. Without loss of generality we assume $S = 1$ and that $LFSR_\mathrm{s}$ is clocked once for each output c_t. Thus, $v = t$ and $c_t = D(L^t(\mathbf{s}^\mathrm{I}))$ 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 $u_k + u_{k+j_1} + ... + u_{k+j_{w-1}} = 0$ that holds over $\mathbf{u}$. First we guess the initial state $LFSR_\mathrm{s}$ and generates the corresponding guess $\hat{\mathbf{c}}$ for $\mathbf{c}$ using the $D()$ function. Using definition **??** we can calculate $\mathbf{u}^* = Q^*(\hat{\mathbf{c}}, \mathbf{z})$. Then we try to find m (typically $m = l_\mathrm{s} + 10$, 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 $\mathbf{s}^\mathrm{I}$. 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 $(k_1, k_2, ..., k_m)$ 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 [**?**]. 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 a initial guess $\hat{\mathbf{s}}^{\mathrm{I}} = (0, 0, ..., 1)$, and let the i'th guess be the internal state of the $LFSR_{\mathrm{s}}$ after i feedback shifts.

Let $\mathbf{c}^i = (c_0^i, c_1^i, ..., c_{M-1}^i)$ be the i'th guess for the clock control sequence defined by $c_t^i = D(L^{i+t}(1, 0, ..., 0))$, $0 \le t < M$. 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 a 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 $(*, ..., *, z_0)$ in $\mathbf{u}^i$, append the $c_{M-1}^{i+1} = c_M^i$ entries $(*, ..., *, z_M)$ at the end, and replace z_t with z_{t-1} for $1 \le t \le M$.*

Proof. See Appendix **??**.

Lemma **??** gives us a fast method for generating all possible guesses for $\mathbf{u}$ given a keystream $\mathbf{z}$. See Table **??** 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*

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

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

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

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

Proof. See Appendix **??**.

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 $\hat{\mathbf{s}}^{\mathrm{I}} = D(L^{i+1}(1, 0, ..., 0)$. 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 $L^i(1, 0, ..., 0)$.

3.4 The Complete Attack

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

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

that holds over all $\mathbf{u}$ generated by $LFSR_{\mathrm{u}}$ for any $k \ge 0$. Let the first guess for the initialization state for $\mathbf{s}$ be $\hat{\mathbf{s}}^{\mathrm{I}} = (1, 0, 0, ..., 0)$, generate $\mathbf{c}^0$ by $c_t^0 =$

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

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 $\mathbf{c}^{i+1}$

$D(L^t(1,0,...0))$, $t < M$, and $\mathbf{u}^0 = Q^*(\mathbf{c}, \mathbf{z})$. Next we try to find m places $(k_1, k_2, ..., k_m)$ in $\mathbf{u}^0$ where the equation λ is defined. From this we get the equation set

$$\begin{array}{ll} u^0_{k_1} + u^0_{k_1+j_1} + ... + u^0_{k_1+j_{w-1}} & = 0 \\ u^0_{k_2} + u^0_{k_2+j_1} + ... + u^0_{k_2+j_{w-1}} & = 0 \\ \vdots & \vdots \\ u^0_{k_m} + u^0_{k_m+j_1} + ... + u^0_{k_m+j_{w-1}} & = 0 \end{array}.$$

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 :

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

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 $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, ..., u_{l_u-1})$ 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 $L^0(1,0,...0)$ and $L^M(1,0...,0)$, Set $i \leftarrow 0$

1. Calculate $c^i_0 = D(L^i(1,0,...,0))$, and $c^{i+1}_{M-1} = c^i_M = D(L^{M+i}(1,0,...,0))$.
2. Use lemma **??** 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 **??** 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 $\mathbf{s}^{\mathrm{I}} = L^{i+1}(1, 0, ..., 0)$ 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 **??**.

All changes during the iterations are done on $\mathbf{u}^i$ and the equation set Ω. Thus, each guess $L^i(1, 0, ..., 0)$ 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 **??**. 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 $\mathbf{s}^{\mathrm{I}}_0 = (1, 0, 0)$. From this we generate the clock control sequence using the function $c^0_t = D(L^t(1, 0, 0))$, $0 \leq t \leq M - 1$, and we get

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

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 $LFSR_{\mathrm{s}}$ once, and it will have $\mathbf{s}^{\mathrm{I}}_1 = L^1(1, 0, 0) = (0, 0, 1)$ as inner state. We calculate $c^1_{M-1} = c^0_M = D(L^M(1, 0, 0))$. Then we use Lemma **??** to calculate $\mathbf{u}^1$ from $\mathbf{u}^0$. That is, we delete the $c^0_0 = 2$ entries $(*, z_0)$, append the ($c^1_{18} = 2$) 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 $\mathbf{u}$:

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

If an equation is defined for the entry t in $\mathbf{z}$ for the guess $\mathbf{s}_0^{\mathrm{I}}$, it will now be defined for the entry $t-1$ in $\mathbf{z}$ for the guess $\mathbf{s}_1^{\mathrm{I}}$ as guaranteed by Theorem **??**. 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 [?] 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 **??**. See Table **??** for some multiples found by the algorithm.

When we have found a polynomial $h(x) = 1+x^{j_1}+...+x^{j_{w-1}}$ with $j_{w-1} = l_{\mathrm{h}}$, the corresponding equation λ over $\mathbf{u}$ is $u_k + u_{k+j_1} + ... + u_{k+j_{w-1}} = 0$. We want to find m places in the stream $\mathbf{u}$ where λ is defined. From equation (**??**) 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 (**??**) 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 a equation set* Ω *of* $m > l_s$ *equations over* $\mathbf{z}$*, the length of the* $\mathbf{z}$ *stream must be*

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

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

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

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 **??** 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 **??**) 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

Degree $l_{\rm s}$ of $g_{\rm s}(x)$	Degree $l_{\rm u}$ of $g_{\rm u}(x)$	Degree $l_{\rm h}$ 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 $l_{\rm u} = 40$ and $l_{\rm u} = 60$. We have set the number of equations to $m = 35$. The polynomials $g(x)$ and $h(x)$ are from Table **??**

$\mathbf{u}^i$to find the initialization bits for $LFSR_{\rm u}$. This has complexity $O(l_{\rm u}^3)$ and will have little effect on the everall 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 **??**. The clock function $D()$ is the LILI-clock function as described in Section **??**. Note that we only attack the irregular 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.

We have used Proposition **??** and Equation **??** 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 paritycheck equations does not effect the complexity. Table **??** shows how the running time of the attack is unchanged when the degree of $g_{\rm u}(x)$ gets larger. The impact from a larger $l_{\rm u}$ 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 keyspace. 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 **??**. To compare with previous LCT attacks, the Gaussian factor $\frac{1}{3}l_{\rm u}^3$ would be around 72000 for $l_{\rm u} = 60$, and around 21333 for $l_{\rm s} = 40$. In our attack the constant factor is only 2 in average.

Thus the same attacks presented in Table **??** 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 irregular 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 $O(2^{l_{\rm s}})$. Thus the run time complexity is independent of the degree $l_{\rm u}$ 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 $\mathbf{s}^{\rm I}$ a metric, we can perform an correlation attack with complexity $O(m \cdot 2^{l_{\rm s}})$ 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. A. Menezes, P. van Oorschot, S. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 211-212, 1997.
2. D. Gollmann and W.G Chambers, “Clock-controlled shift registers: a review”, *IEEE Journal on Selected Areas in Communications*, 7 (1989), 525-533
3. K. Zeng, C. Yang and Y. Rao, “On the linear consistency test (LCT) in cryptanalysis with applications”, *Advances in Cryptology-CRYPTO ’89* (LNCS 435), 164-174, 1990
4. D. Wagner, “A Generalized Birthday problem”, Advances in cryptology-*CRYPTO* ’02, (LNCS 2442), 288-303, 2002
5. T. Johansson, and F. Jönsson, “Fast Correlation Attacks on Stream Ciphers via Convolutional Codes”, *Advances in Cryptology*-EUROCRYPT’99, Lecture Notes in Computer Science, Vol. 1592, Springer-Verlag, 1999, pp. 347-362.
6. Patrik Ekdahl, Willie Meier, Thomas Johannson, “Predicting the Shrinking Generator with Fixed Connections”, EUROCRYPT 2003, Lecture Notes in Computer Science, Vol. , Springer-Verlag, 1999, 330-334
7. J.D. Golic, “Cryptanalysis of three mutually clock-controlled stop/go shift registers.” *IEEE Trans. Inf Theory,* 46(3),525-533, 2000
8. E. Zenner, M. Krause, and S. Lucks,“Improved cryptanalysis of the self-shrinking generator”, *Proc ACISP ’01,* (LNCS 2119), 21-35, 2001
9. Jovan Dj. Golic, 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

10. Jovan Dj. Golic and Slobodan V. Petrovic,"A Generalized Correlation Attack with a Probabilistic Constrained Edit Distance", (LNCS 658), 472, 1993
11. Erik Zenner, "On the Efficiency of the Clock Control Guessing Attack", *ICISC 2002*, (LNCS 2587)
12. L. Simpson, E. Dawson, J. Dj Golic, and W. Millan, "LILI keystream generator", *SAC'2000,* (LNCS 2012),248-261,2000

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 **??** 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 **??** 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 $\mathbf{u} = (..., z_0 ..., z_t, ..., z_M, ...)$, the entries $z_0, ..., z_t, ..., z_M$ 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}. But when we after i iterations in the search for equations find an equation $u_k^i + u_{k+j_1}^i + ... + u_{k+j_{w-1}}^i = 0$ that is defined, we replace the corresponding $z_{t_1} + z_{t_2} + ... + z_{t_w}$ with $z_{t_1 - i} + z_{t_2 - i} + ... + z_{t_w - i}$, 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 **??** to generate $\mathbf{u}^{i+1}$.

B Proofs of Lemma ?? and Theorem ??

B.1 Proof of Lemma ??

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 $u^{i+1}_{k-c^i_0} = z_{t-1}$. If we delete the first c^i_0 bits of $\mathbf{u}^i$ and get $\mathbf{u}'$ we will have that if $u'_k = z_t$ for given k, then $u^{i+1}_k = z_{t-1}$ for $k = \sum_{j=0}^{t-1} c^{i+1}_j - 1$. 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 $u''_k = u^{i+1}_k$, $0 \leq k < N_i - c^i_0$. To finally transform $\mathbf{u}''$ into $\mathbf{u}^{i+1}$ we just have to append the c^{i+1}_{M-1} entries $(*, ..., z_{M-1})$ at the end of $\mathbf{u}''$.

B.2 Proof of Theorem ??

Proof. Let

$$\mathbf{u}^i = (..., \underbrace{z_0}_{c^i_0-1}, ..., *, ..., \underbrace{z_t}_{k}, ..., \underbrace{z_{t+j_1}}_{k+k_1}, ..., *, ... \underbrace{, z_{t+j_w}}_{k+k_{w-1}}, ..., \underbrace{z_{M-1}}_{N_i-1})$$

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 $\beta_{\mathbf{u}_i,k} = u^i_k + u^i_{k+k_1} + ... + u^i_{k+k_{w-1}}$ is defined over $\mathbf{u}^i$. The corresponding sum over $\mathbf{z}$ will be $\gamma_{\mathbf{z},t} = z_t + z_{t+j_1} + ... + z_{t+j_{w-1}}$. Then the clock control sequence we get from $c^{i+1}_t = L^{i+1+t}(1, 0, ..., 0)$ will be

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

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 $\underbrace{(*, ..., z_0)}_{c^i_0}$ in $\mathbf{u}^i$, appending $\underbrace{(*, ..., *, z_M)}_{c^{i+1}_M}$ at the end, and then replacing z_t with z_{t-1} for $1 \leq t \leq M$, as explained in lemma **??**. From this we get the sequence

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

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

C Searching for Parity Check Equations

C.1 The Generator Matrix

Let $g(x) = 1 + g_{l-1}x + g_{l-2}x^2 + ... + g_1 x^{l-1} + x^l$, $g_i \in \mathcal{F}_2$, $g_l = g_0 = 1$ be the primitive feedback polynomial of degree l for a shift register that generates the sequence $\mathbf{u} = (u_0, u_1, ..., u_{N-1})$. The corresponding recurrence is $u_{t+l} = g_1 u_{t+l-1} + g_2 u_{t+l-2} + ... + g_l u_t$. Let α be defined by $g(\alpha) = 0$. From this we get the reduction rule $\alpha^l = g_1\alpha^{l-1} + g_2\alpha^{l-2} + ... + g_{l-1}\alpha + 1$. Then we can define the generator matrix for sequence u_t,$0 < t < N$ by the $l \times N$ matrix

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

For each $i > l$, using the reduction rule α^i can be written as $\alpha^i = h^i_{l-1}\alpha^{l-1} + ... + h^i_2\alpha^2 + h^i_1\alpha + h^i_0$. 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

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

Now the sequence $\mathbf{u}$ with length N and initialization state $\mathbf{u}^{\mathrm{I}} = (u_0, u_1, ..., u_{l-1})$, 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 $\mathbf{u}$ be a sequence generated by the generator polynomial $g(x)$ with degree l. It is well known that if we can find $w > 2$ columns in the generator matrix G, that sum to zero,

$$(\mathbf{g}_{j_0} + \mathbf{g}_{j_1} + \ldots + \mathbf{g}_{j_{w-1}})^{\mathbf{T}} = (0, 0, ..., 0), \tag{13}$$

for $l \leq j_0, j_1, ..., j_{w-1} < N$, we get an equation of the form

$$u_{j_0} + u_{j_1} + ... + u_{j_{w-1}} = 0. \tag{14}$$

The equation (**??**) can be formulated as $\alpha^{j_0} + \alpha^{j_1} + ... + \alpha^{j_{w-1}} = 0$. Thus, if (**??**) holds, the equation $\alpha^t(\alpha^{j_0} + \alpha^{j_1} + ... + \alpha^{j_{w-1}}) = \alpha^{j_0+t} + \alpha^{j_1+t} + ... + \alpha^{j_{w-1}+t} = 0$ 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

$$u_{t+j_0} + u_{t+j_1} + ... + u_{t+j_{w-1}} = 0, \tag{15}$$

for $0 \leq t < N - j_w$.

We can also use the indexes $j_1, j_2, ...j_w$ to formulate the polynomial $h(x) = x^{j_0} + x^{j_1} + ... + x^{j_{w-1}}$. If $j_0, j_1...$ 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. Then for every choice of the columns $\mathbf{g}_{j_0}, \mathbf{g}_{j_1}, ..., \mathbf{g}_{j_{w-2}}$ in G search for the w'th column $\mathbf{g}_{j_{w-1}}$that gives $\mathbf{g}_{j_{w-1}} = \mathbf{g}_{j_0} + \mathbf{g}_{j_1} + ... + \mathbf{g}_{j_{w-2}}$. This algorithm is not very effective and has the complexity

$O(n^{w-1}\text{log}n)$. By using hashing techniques we can get down to $O(n^{w-1})$. We can do better if we use the iterative method explained next. The algorithm is a modification of the Generalized birthday algorithm in [?].

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 B_1. The columns that are equal in the lowest $l - B_1$ bits, will now be 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 zero. According to Section **??** we will now have found an 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 [?] for finding equations of the type $c_0u_0+c_1u_1+...+c_{B-1}u_{B-1} = u_i+u_j$. >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 (**??**) 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 $O(\sum_{i=0}^{r-1} m_i \text{log} m_i)$ since we have to sort the matrices $G_1, G_2..., G_r$. Thus, it is not the complexity that limits the algorithm, but the memory. 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. Then 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 $G_i = T_{\text{mem}} * B_{i-1}$. Thus we can set $N = m_1 = ... = m_{r-1} = T_{\text{mem}}$. 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*

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

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

Proof. $>$From equation (**??**) we have these formulas for $m_1, ..., m_r$:

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

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

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

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

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

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

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

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

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