Practical Design of Nonlinear Combiner Model as a Software Stream Cipher

Sandeepan Chowdhury, Subhamoy Maitra and Bimal Roy
Applied Statistics Unit, Indian Statistical Institute,
203, B T Road, Calcutta 700 108, INDIA
email : {sand_t, subho, bimal}@isical.ac.in

Abstract

Here we present a general design criteria for the nonlinear combiner model which is well known for the last two decades. Our proposed specifications present a model that can resist all the existing attacks. This model combines the outputs of several independent Linear Feedback Shift Register (LFSR) sequences using a Boolean function to produce the running key sequence. Hardware implementation of this model is known to be quite efficient. However, the naive bitwise software implementation of the LFSRs and the Boolean function is not efficient. We have explored efficient block oriented software implementation technique to make it competitive with the recently proposed fast ciphers. To make everything concrete, we propose a simple scheme with specific design parameters having 128/192/256 bit key length. We claim that there is no existing attack better than the exhaustive key search on this specific scheme. Our software implementation can achieve a speed as high as 18–70 Gbps on varied platforms (Pentium III, 666 MHz to SUN ULTRA 60 SPARC 2, 360 MHz). No other existing stream cipher can achieve such a speed in software. Our analysis clearly presents the nonlinear combiner model as a strong contender to other new stream cipher schemes.

Keywords : Linear Feedback Shift Register, Block Oriented Software Implementation, Boolean Function, Resiliency, Nonlinearity, Algebraic Degree.

1 Introduction

Recently number of new stream ciphers have been proposed with the aim of developing a standard. Hardware implementation of stream cipher is quite fast. However, the naive software implementation of the stream ciphers are not quite good. In order to be efficient in software, the scheme either needs to exploit some simple encryption algorithm, like RC4 or needs to utilise a word oriented implementation strategy as SEAL [31], SSC2 [38], SNOW [10, 11], t-classes of SOBER [32], and SCREAM [19]. RC4, the most widely used stream cipher algorithm in software, has been recently found to have a lot of weaknesses in its key scheduling algorithms [13] and its use in certain areas are no more safe [25]. Note that many of the schemes are found to have certain weaknesses in their novel way of implementation, within

a very short time of being proposed. SSC2 is found to be weak against certain cryptanalytic attacks [18, 17]. SNOW (version 1.0) has also been found to have certain weaknesses [16, 6] and a new variant of SNOW (version 2.0) [11] has to be proposed. SCREAM-0, a version of SCREAM, also shows some weaknesses [19]. In this context we explore the possibility of the old but well studied nonlinear combiner model [35, 36, 9, 27] as an efficient software stream cipher system.

Most unfortunately, the standard nonlinear combiner model, well known from early eighties, is not being evaluated seriously. The cryptanalytic attacks in the last two decades have been the major motivation towards progressive development of individual components of this model. Reviewing these attacks, as well as existing design criteria, we present a general method for implementing a robust nonlinear combiner model, secure against all the existing attacks. Moreover, we introduce a block oriented implementation strategy for fast software implementation of the scheme.

The main building block of the nonlinear combiner model are the LFSRs and a Boolean function. The scheme is presented in Figure 1. In this model, the output sequences of several independent Linear Feedback Shift Registers (LFSR) are combined using a nonlinear Boolean function to produce the running key stream K. The n LFSRs and the Boolean function are presented as $S_1, \ldots, S_n$ and f (see Figure 1) respectively. This key stream K is bitwise XORed with the message bit stream M to produce the cipher C. The decryption machinery is identical to the encryption machinery. The initial conditions in the LFSRs constitute the key of the system.

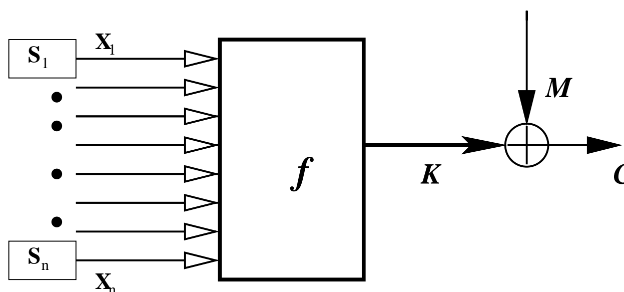

Figure 1: Nonlinear Combiner Model

This model is existing for a long time and has been subjected to different kinds of attacks [36, 26, 9, 20, 21, 12, 2, 1, 3, 22, 28, 29]. The slow software realisation of the bit oriented LFSRs reduce the efficiency of this model in software. May be due to this reason, this simple but reliable model has not been considered as a possible candidate and no effort has been made in the open literature to completely specify a scheme on the basis of this model. However, the very fact that the model has been cryptanalysed in great detail, is a strength. As the attacks on this model appear to be stabilised, it is now possible to fix the proper design criteria for the LFSRs [26, 2, 24] as well as that of the Boolean function [34, 30, 37]. So considering the present scenario, we re-examine this model in detail. At this point, let us highlight the salient features of this initiative.

1. We start the basic design using an important formula presented by Chepyzhov, Johansson and Smeets [3] which is actually a result from cryptanalysis. Further, it can be immediately checked that our design methodology provides robustness against the attack proposed by Canteaut and Trabbia [2], and Mihaljevic, Fossorier and Imai [29]. It should be acknowledged that the nice closed forms of these formulae [3, 2, 29] on cryptanalysis help in using themselves in terms of design too. Then we suitably incorporate recent theoretical developments in design of Boolean function and LFSRs to choose each component of the system.

2. We explain the strategy for block oriented implementation of LFSRs to get considerable speed up in software. Further we show that the algebraic normal form representation of Boolean functions can be efficiently used for block oriented strategy.

3. On the basis of our analysis, we present a concrete 128/192/256 bit scheme specifying the exact LFSR connection polynomials and the Boolean function. These components are intentionally chosen at a lower end to provide better leverage to the cryptanalysts. Even then, to the best of our knowledge there is no existing attack which performs better than the exhaustive key search.

The organization of the paper is as follows. First we present some preliminary notions. In Section 3 we develop the design approach and discuss the strength of this scheme in light of the existing attacks. Next we discuss the issue of efficient software implementation in Section 4. As an example of our design strategy, we propose a specific 128/192/256 scheme in Section 5.

2 Preliminaries

The basic components of the nonlinear combiner model are some LFSRs and a Boolean function.

An LFSR of length d generates a binary pseudorandom sequence following a recurrence relation over GF(2). It takes a small seed of length d and expands it to a much larger binary pseudorandom sequence with high periodicity. We consider a degree d primitive polynomial over GF(2) of form $x^d \oplus \bigoplus_{i=0}^{d-1} a_i x^i$. The corresponding binary recurrence relation is $s_{j+d} = \bigoplus_{i=0}^{d-1} a_i s_{j+i}$. Note that one uses a primitive polynomial to get maximum possible periodicity $2^d - 1$. This polynomial is called the "connection polynomial" of the LFSR. The weight of this connection polynomial is the number of places where the coefficient has the value 1, i.e., $1 + \#\{a_i = 1\}$. The taps of the LFSR come from the positions where $a_i = 1$ and there is no tap at $a_i = 0$.

By a τ-nomial multiple of the connection polynomial we mean a multiple of the form $x^{i_1} + x^{i_2} + \ldots + x^{i_{\tau-1}} + 1$ with degree less than $(2^d - 1)$.

An n-variable Boolean function f is defined as $f : GF(2^n) \rightarrow GF(2)$. Now we present some definitions relevant to Boolean functions.

Definition 2.1 *The addition operator over $GF(2)$ is denoted by $\oplus$. For binary strings S_1, S_2 of same length λ, we denote by $\#(S_1 = S_2)$ (respectively $\#(S_1 \neq S_2)$), the number of places where S_1 and S_2 are equal (respectively unequal). The Hamming distance between S_1, S_2 is*

denoted by $d(S_1, S_2)$, i.e., $d(S_1, S_2) = \#(S_1 \neq S_2)$. Also the Hamming weight or simply the weight of a binary string S is the number of ones in S. This is denoted by $wt(S)$. An n-variable function f is said to be balanced if its output column in the truth table contains an equal number of 0's and 1's (i.e., $wt(f) = 2^{n-1}$).

Definition 2.2 *An n-variable Boolean function $f(X_1, \ldots, X_n)$ can be considered to be a multivariate polynomial over $GF(2)$. This polynomial can be expressed as a sum of products representation of all distinct r-th order products ($0 \leq r \leq n$) of the variables. More precisely, $f(X_1, \ldots, X_n)$ can be written as $a_0 \oplus (\bigoplus_{i=1}^{i=n} a_i X_i) \oplus (\bigoplus_{1 \leq i \neq j \leq n} a_{ij} X_i X_j) \oplus \ldots \oplus a_{12\ldots n} X_1 X_2 \ldots X_n$ where the coefficients $a_0, a_i, a_{ij}, \ldots, a_{12\ldots n} \in \{0, 1\}$. This representation of f is called the algebraic normal form (ANF) of f. The number of variables in highest order product term with nonzero coefficient is called the algebraic degree, or simply degree of f.*

We will later show that using ANF helps faster implementation of Boolean functions when realised in block oriented manner.

Definition 2.3 *Functions of degree at most one are called affine functions. An affine function with constant term equal to zero is called a linear function. The set of all n-variable affine (respectively linear) functions is denoted by $A(n)$ (respectively $L(n)$). The nonlinearity of an n variable function f is $nl(f) = min_{g \in A(n)}(d(f, g))$, i.e., the distance from the set of all n-variable affine functions.*

Definition 2.4 *Let $\overline{X} = (X_1, \ldots, X_n)$ and $\overline{\omega} = (\omega_1, \ldots, \omega_n)$ both belong to $\{0, 1\}^n$ and $\overline{X} \cdot \overline{\omega} = X_1\omega_1 \oplus \ldots \oplus X_n\omega_n$. Let $f(\overline{X})$ be a Boolean function on n variables. Then the Walsh transform of $f(\overline{X})$ is a real valued function over $\{0, 1\}^n$ that can be defined as $W_f(\overline{\omega}) = \sum_{\overline{X} \in \{0,1\}^n} (-1)^{f(\overline{X}) \oplus \overline{X} \cdot \overline{\omega}}$.*

Definition 2.5 [15] *A function $f(X_1, \ldots, X_n)$ is m-th order correlation immune (CI) iff its Walsh transform W_f satisfies $W_f(\overline{\omega}) = 0$, for $1 \leq wt(\overline{\omega}) \leq m$. Also f is balanced iff $W_f(\overline{0}) = 0$. Balanced m-th order correlation immune functions are called m-resilient functions. Thus, a function $f(X_1, \ldots, X_n)$ is m-resilient iff its Walsh transform W_f satisfies $W_f(\overline{\omega}) = 0$, for $0 \leq wt(\overline{\omega}) \leq m$.*

By an (n, m, u, x) function we mean an n-variable, m-resilient function with degree u and nonlinearity x. It is known that for $m > \frac{n}{2} - 2$, the maximum possible nonlinearity can be $2^{n-1} - 2^{m+1}$ and such functions have three valued Walsh spectra [34]. The maximum possible algebraic degree of such functions is $n - m - 1$ [35]. Construction of such functions has been demonstrated in [37, 30].

3 The Design Approach

Existing correlation attacks on the nonlinear combiner model exploits the correlation between the cipherbit sequence and the bit sequence coming out of one or more LFSRs. To be specific, one considers the correlation between the sequence generated by one or more LFSRs and the cipher text C [35, 36] (in turn K). Here we consider an (n, m, d, x) Boolean function f. It is known that there always exists a linear function $\bigoplus_{i=1}^{n} \omega_i X_i$ (with $wt(\omega_1, \ldots, \omega_n) = m + 1$) such that the correlation coefficient between the stream coming out of f and the linear function $\bigoplus_{i=1}^{n} \omega_i X_i$ is not zero.

Definition 3.1 *The sequence σ generated by XORing the sequences generated by the individual LFSRs, say, $S_{i_1}, \ldots, S_{i_{m+1}}$ is same as that produced by the LFSR S with length $L = \sum_{i=1}^{m+1} d_i$ and connection polynomial $\psi(x) = \prod_{j=1}^{m+1} c_{i_j}(x)$ [9]. We refer to this single LFSR of length L and connection polynomial $\psi(x)$ as the equivalent LFSR.*

The nonlinear combiner model can be attacked if the initial condition of an equivalent LFSR can be obtained. The standard models of correlation attacks recover the initial condition of a target LFSR from a perturbed version of the original output sequence. These attacks fit into the nonlinear combiner model, if we consider the equivalent LFSR as the target LFSR, the sequence σ as the original sequence and the running key sequence K as the perturbed sequence. Note that it is generally not possible to get K and the perturbed sequence is actually C. Naturally the attacker will consider the smallest length equivalent LFSR. So from the designer's viewpoint, fix the design parameters of the nonlinear combiner model to make the smallest length equivalent LFSR robust against correlation attacks.

Under this framework, the "divide and conquer attack" [36] has a complexity $\mathcal{O}(2^L)$ and is clearly not feasible for high values of L (say > 128). The fast correlation attack proposed in [26] improved the complexity of this exhaustive search to $2^{\beta L}$ where $\beta < 1$. It has been mentioned in [26, Section 1] that the attack is not feasible if the target LFSR has more than 10 taps. So the weight of the equivalent LFSR S must be > 10. However one also needs to ensure that a high weight connection polynomial does not have a sparse multiple of low degree [26, Section 5]. This criteria has been dealt in more detail later.

The modified fast correlation attacks developed subsequently [20, 21, 3, 22, 2, 1, 28, 29] and they transformed this problem of cryptanalysis to some decoding problem. According to this model of cryptanalysis, the running key sequence K is the perturbed version of the output sequence σ of an LFSR (here the equivalent LFSR), after passing through a Binary Symmetric Channel (BSC) with error probability $p = P[\sigma_i \neq K_i] < \frac{1}{2}$ (see Figure 2). Now we relate this error probability p of the BSC with the parameters of the nonlinear combiner model. In that case,

$$p = P[\sigma_i \neq K_i] = \frac{1}{2} - \frac{\max_{\omega \in \{0,1\}^n}(|W_f(\overline{\omega})|)}{2^{n+1}}.$$

If we correlate the cipherbits C with σ, then the error probability p increases further [2] and hence the complexity of the attack increases. In order to establish the robustness of our scheme we consider the correlation between σ and K.

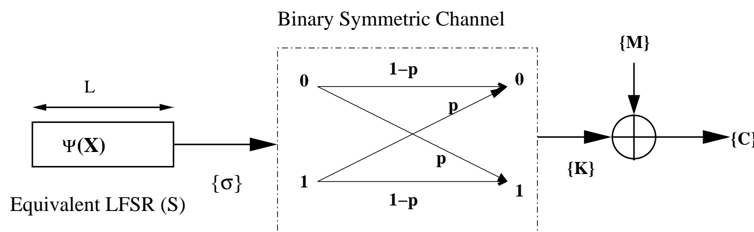

Figure 2: Coding theory based model of Correlation Attack

Consider the fast correlation attack described by Chepyzhov, Johansson and Smeets (CJS) [3]. Their idea is to reduce the decoding complexity of the $[N, L]$ binary code by associating it with another code $[n_2, k]$ of lower dimension ($k < L$). This $[n_2, k]$ code is obtained using sets of t ($t \geq 2$) parity check equations of the $[N, L]$ code [3, Algorithm A2]. This helps to recover k initial bits of the target LFSR (of length L). According to [3], the BSC has probability $p = \frac{1}{2} - \epsilon < 0.5$. The amount of cipherbits (N) required for cryptanalysis is given by $N = \frac{1}{4}(2k\,t!\,\ln 2)^{1/t}\epsilon^{-2}2^{\frac{L-k}{t}}$ [3] where k is the number of initial bits recovered correctly from the attack. For a given value of L and k, cipherbit requirement N increases as ϵ decreases. Comparing the expression of p in BSC and the Boolean function model, we get $\epsilon = \frac{\max_{\omega \in \{0,1\}^n}(|W_f(\overline{\omega})|)}{2^{n+1}}$.

It is important to note here that the Boolean function needs to have highest possible nonlinearity in order to possess maximum resistance against the best affine approximation (BAA) attack [9]. Moreover, maximum nonlinearity provides the minimum value of ϵ, which in turn increases the complexity of the attack. We generally select a resilient function with three valued Walsh spectra. The (n, m, u, x) functions with best possible nonlinearity must have three valued Walsh spectra for $m > \frac{n}{2} - 2$ [34]. In this case, $\max_{\omega \in \{0,1\}^n} |W_f(\overline{\omega})| = 2^{m+2}$ and hence, $\epsilon = \frac{2^{m+2}}{2^{n+1}} = 2^{m-n+1}$.

We start our design from the formulae available from [3] and later we will show the resistance against the other existing attacks.

3.1 Design Criteria for Equivalent LFSR

We denote the key length by q. From designer's viewpoint the time complexity of any attack should be $\geq 2^q$. The precomputation memory and time complexity of [3, Algorithm A2] are $N^{\lfloor \frac{t-1}{2} \rfloor}$ and $N^{\lceil \frac{t-1}{2} \rceil}$ [10, Section 4]. The decoding complexity is $2^k\, k\, \frac{2\ln 2}{(2\epsilon)^{2t}}$.

Fact 3.1 *The criteria for safe design against CJS attack gives $L = 4\,q$.*

Reason 3.1 *From the designer's point of view we can safely approximate (underestimate) N as $2^{\frac{L-k}{t}}$. Thus, $N^{\lceil \frac{t-1}{2} \rceil}$ can be approximated as $2^{\frac{L-k}{t} \cdot \lceil \frac{t-1}{2} \rceil}$. If t odd, then*

$$2^{\frac{L-k}{t} \cdot \lceil \frac{t-1}{2} \rceil} = 2^{\frac{L-k}{t} \cdot \frac{t-1}{2}} = 2^{\frac{L-k}{t} \cdot \frac{t}{2}} 2^{\frac{L-k}{t} \cdot \frac{-1}{2}} = 2^{\frac{L-k}{2}} 2^{-\frac{L-k}{2t}}.$$

If t is even,

$$\lceil \frac{t-1}{2} \rceil = \frac{t}{2}, \text{ and } 2^{\frac{L-k}{t} \cdot \lceil \frac{t-1}{2} \rceil} = 2^{\frac{L-k}{2}}.$$

So the problem is when t is odd. The term $2^{-\frac{L-k}{2t}}$ will be minimized when t is the smallest possible odd integer greater than 2, i.e., $t = 3$. Then the minimum value of the term is $2^{-\frac{L-k}{6}}$. So, we can underestimate $N^{\lceil \frac{t-1}{2} \rceil}$ as $2^{\frac{L-k}{2}} 2^{-\frac{L-k}{6}} = 2^{\frac{L-k}{3}}$. Hence for safe design, $\frac{L-k}{3} \geq q$, i.e., $L = 3q + k$. Now the decoding complexity is $2^k \cdot k \frac{2\ln 2}{(2\epsilon)^{2t}}$. If we consider k is as large as q, then $2^{\frac{L-k}{3}} = 2^q$ and also the decoding complexity is greater than 2^q. Thus we take $L = 3q + k = 4q$ for a safe design.

However, for a more tight design, we can very well decrease L by some small amount considering the other terms, mainly the terms related to ϵ.

Example 3.1 *Let us consider* $q = 256, L = 1000, \epsilon = 0.125$. *Note that* $4q = 1024$, *but considering* ϵ, *we reduce* L *a little bit and take* $L = 1000$. *For this value of* L, *we present in Table 4, the cipherbit requirement, preprocessing and decoding complexity for carrying out the CJS attack for different values of* t *and* k. *Note that, for any given values of* k *and* t, *the corresponding maximum value of time complexity is always at least* 2^{256}. *Thus for* $L = 1000$, *the attack of [3] can succeed in no way with complexity less than the exhaustive key search, i.e.,* 2^{256}.

Now we concentrate on the attack proposed by Canteaut and Trabbia (CT) [2]. The attack has used parity check equations of weight 4 and 5. According to their estimation, the cipherbit requirement is given by $N = 2^{\alpha_\tau(p)+\frac{L}{\tau-1}}$, where $\alpha_\tau(p) = \frac{1}{\tau-1} log_2[(\tau-1)! \frac{k_\tau}{C_{\tau-2}(p)}]$, $C_{\tau-2}$ is the capacity of the binary symmetric channel with overall error probability $p_{\tau-2}$, i.e., $C_{\tau-2} = p_{\tau-2}\log_2(p_{\tau-2}) + (1-p_{\tau-2})\log_2(1-p_{\tau-2})$ and L is the length of the target LFSR. Here $k_\tau \approx 3$ for $\tau = 3$ and $k_\tau = 1$ for $\tau \geq 4$. The overall error probability of the BSC is given by $p_{\tau-2} = \frac{1}{2}(1-(1-2p)^{\tau-2}$, p is the error probability of the BSC. Number of parity check equations of weight τ is $m(\tau) = \frac{N^{\tau-1}}{(\tau-1)!2^L}$. Complexity of the preprocessing stage is $\frac{N^{\tau-2}}{(\tau-2)!}$. The decoding time complexity is given by $5(\tau-1)\cdot N \cdot m(\tau)$. The total memory requirement is $2N + (\tau-1)\cdot m(d)$ computer words.

Fact 3.2 *The condition* $L = 4q$ *resists CT attack.*

Reason 3.2 *The time complexity of the preprocessing and decoding stages of CT attack is given by* $\frac{N^{\tau-2}}{(\tau-2)!}$ *and* $5(\tau-1)\cdot N \cdot m(\tau)$ *respectively. The memory requirement for the attack is* $2N + (\tau-1)\,m(\tau)$. *Here* τ, *(weight of the parity check equation) is 3, 4 or 5. So, clearly the complexity of the attack is proportional to the cipherbit requirement* N. *Now* N *is given as* $2^{\alpha_\tau(p)+\frac{L}{\tau-1}}$. *We can underestimate* N *as* $N \approx 2^{\frac{L}{\tau-1}}$, $(\alpha_\tau(p) > 0)$. *Putting* $\tau = 5$, *the minimum cipherbit requirement is* $N \approx 2^{\frac{L}{4}}$. *Putting* $L = 4q$, *we get* $N \approx 2^q$. *Hence, for* $L = 4q$, *the complexity of of the CT attack is* $\approx 2^q$.

The correlation attack described by Mihaljevic, Fossorier and Imai (MFI) [29] uses list decoding technique. and we show that our scheme is robust against this attack.

Fact 3.3 *The condition* $L = 4q$ *resists MFI attack.*

Reason 3.3 *The complexity of processing phase of the attack is given by* $2^B[(D-B)|\Omega| + (M-L)\,w]$ *[29, Section 5.2]. Here* $|\Omega| = 2^{B-L}\binom{N-L}{2}$ *gives the average number of parity checks for any bit,* w *is weight of the connection polynomial,* B *is the number of initial LFSR bits recovered by exhaustive search,* D *is the number of codeword bits under consideration. As the calculation of* M *(required number of bits for checking the minimum decoding distance) involves error probability* p *of BSC, it is hard to express the complexity in terms of* L *only. However, we note that the term* $2^{B-L}\binom{N-L}{2} \approx 2^{B-L}N^2$, *since* $N >> L$. *Clearly,* $|\Omega|$ *needs to have a nontrivial value, i.e.,* ≥ 1. *So we have* $2^{B-L}N^2 \geq 1$. *Hence* $N^2 \geq 2^{L-B}$. *Putting* $L = 4q$, $N \geq 2^{2q-\frac{B}{2}}$. *Assuming,* $B = 2q$, *we get* $N \geq 2^q$. *In that case processing complexity is also* $\geq 2^q$. *For any value of* $B < 2q$, *the cipher bit requirement and in turn the processing complexity will be much higher than* 2^q. *Thus considering* $L = 4q$, *secures the nonlinear combiner model against attack in [29].*

At this point let us refer to a few more recent attacks. The decimation attack described by Filiol [12] utilises a simulated LFSR of shorter length obtained by decimating the cipherbit sequence. The complexity of the attack is dependent on the length of this simulated LFSR. In [12, Table 3], it has been indicated that the complexity of the decimation attack is more than the the attack by Chepyzhov, Johansson and Smeet [3]. So using $L = 4q$, complexity of decimation attack will be more than 2^q. It is of great interest to see whether the recently published distinguishing attack [19], Guess-and-Determine attack [16], and the algebraic attacks [7, 8] become successful on the nonlinear combiner model, designed with the criteria described here. However, we could not find any immediate application of these attacks with time complexity less than 2^q. In fact, in Section 5 we present a concrete scheme and application of these attacks or any other attack on this scheme is interesting research question.

3.2 Fixing Boolean function and LFSR parameters

We have already noted that $\epsilon = \frac{2^{m+2}}{2^{n+1}} = 2^{m-n+1}$. For a given value of ϵ,

1. we first need to decide about n, m and
2. once n, m are fixed, the length of each LFSR is calculated.

Now the important question is deciding n and m. First of all, we are interested about resilient functions with maximum possible nonlinearity and 3-valued Walsh spectra which needs the condition $m > \frac{n}{2} - 2$ [34]. It is known that the algebraic degree of such function is $u = n - m - 1$ [37]. Further from the viewpoint of software implementation, it is better to have lesser values of n, m, i.e., less number of LFSRs which makes the software more efficient (see Subsection 4.1 for details). Thus the requirement is to get n, m satisfying the conditions (i) $m > \frac{n}{2} - 2$, and (ii) $2^{m-n+1} \leq \epsilon$.

High algebraic degree of the Boolean function ensures high linear complexity for the output keystream K obtained from the Boolean function f [9]. For achieving high linear complexity one needs n-variable m-resilient Boolean functions with the maximum possible algebraic degree $n - m - 1$ [35]. It is now known that the resilient function with maximum nonlinearity must have the maximum algebraic degree too [34].

From Definition 3.1, length of an equivalent LFSR $L = \sum_{j=1}^{m+1} d_{i_j}$. As the attacker will target the smallest length equivalent LFSRs, we need the average length of an LFSR (or degree of its connection polynomial) $d_{av} = \frac{L}{m+1}$.

Next we fix the degree of the LFSRs $d_1, d_2, \ldots, d_n$ and the criteria for connection polynomials. The degrees need to be pairwise coprime [23, Page 224] in order to maximise linear complexity of the running key sequence K. Also we need that $d_{i_1} + d_{i_2} + \ldots + d_{i_{m+1}} \geq L$ for any subset of $m + 1$ LFSRs. Now we consider the following fact.

Fact 3.4 *The connection polynomial $\psi(x)$ of equivalent LFSR must have weight > 10, but weight of the connection polynomial $c_i(x), 1 \leq i \leq n$, of the individual LFSRs S_i should be low (< 10).*

Reason 3.4 *To resist the attacks of Meier and Stafflebach [26], the connection polynomial $\psi(x)$ of equivalent LFSR must have a high (> 10) weight (already explained in Subsection 3).*

The number of logical operations required for generating the output from an LFSR is dependent on the number of XOR operations (i.e., the number of taps) in its recurrence relation. So, for the sake of fast implementation in software, we need connection polynomial of individual LFSRs of low weight (see Subsection 4.1 for details). These two apparently contradictory requirements are achievable. For example, consider three trinomials of degree d_1, d_2, d_3, say. If properly chosen, the product polynomial may have weight as high as 27. Thus we have to ensure that weight of $\prod_{j=1}^{m'} c_{i_j}(x)$ should be high, for any subset (of LFSRs) having size m' $(m+1 \leq m' \leq n)$.

Now consider about τ-nomial multiples of $\psi(x) = \prod_{j=1}^{m+1} c_{i_j}(x)$ of degree L. It has been explained in [2, 24] that the expected degree of the least degree τ-nomial multiple of $\psi(x)$ is of the order of $2^{\frac{L}{\tau-1}}$. The most important attack presented using τ-nomial multiples is the CT attack [2]. In that case, the value of τ has been taken as $3, 4, 5$. Note that, even if $\tau = 5$, the minimum degree τ-nomial multiple is of the order $2^{\frac{L}{4}} = 2^q$ and the CT attack seems to be infesible under this circumstances.

4 Software Implementation

In this section we discuss the block oriented software implementation of LFSRs and the Boolean function to enhance the processing speed.

4.1 Software Implementation of LFSRs

Hardware implementation of an LFSR generates a single bit per clock. In naive software implementation more than one (say v) logical operations (bitwise XOR, AND, and bit shifting) are required to generate a single output bit. The motivation behind the fast software implementation of an LFSR is to generate a block of b output bits in $< bv$ logical operations. For this purpose we discuss the block oriented implementation. Here the LFSR outputs one block at a time. We denote this output of b bit vectors as a "block", i.e., $\mathbf{w}_j = \{s_{jb+(b-1)}, \ldots, s_{jb+1}, s_{jb}\}$. Thus the output N bit sequence $s_j, 0 \leq j < N$, obtained by bitwise operating an LFSR, is now generated as the sequence of blocks $\mathbf{w}_j, 0 \leq j < N'$ where $N = N' b$. For convenience of storage and operations in processors, it is natural to take $b = 8, 16, 32, 64$ etc.

First we discuss a block oriented implementation of an LFSR by matrix operation. The standard matrix representation of an LFSR can be seen as $\mathbf{x}_b = \mathcal{A}^b \mathbf{x}_0$, where $\mathcal{A}$ is the $d \times d$ state transition matrix and $\mathbf{x}_0$ is the d-bit initial state vector of the LFSR [23]. Note that, to get one b length block we need a binary matrix multiplication. Later in Subsection 4.1.1 we present our scheme and it is clear that our scheme is much faster than the matrix multiplication strategy.

Moreover, it can be shown that the bitwise recurrence relation of an LFSR $s_{j+d} = \bigoplus_{i=0}^{d-1} a_i s_{j+i}$, can be extended to that over blocks $\mathbf{w}_{j+d} = \bigoplus_{i=0}^{d-1} a_i \mathbf{w}_{j+i}$. For implementing this block oriented relation, we consider an LFSR consists of d blocks corresponding to the original LFSR of d bits. We denote the LFSR blocks by $(S[d-1], \ldots, S[1], S[0])$ from left to right. The initial state is represented as $S[d-1] = \mathbf{w}_{d-1}, \ldots, S[1] = \mathbf{w}_1, S[0] = \mathbf{w}_0$. So one needs the first bd bits of the LFSR bit sequence s_j for initialisation of this d blocks.

Once initialised, we use the block oriented recurrence relation to generate the output sequence $\{\mathbf{w}_j\}_{j\geq 0}$. Here, only $\frac{t}{b}$ logical operations are required per output bit, where t is the total number of taps. Considering a primitive trinomial (i.e., $t = 2$) and block size $b = 64$, $\frac{t}{b} = \frac{1}{32}$. However, precomputation involving the generation of bd bits for initialisation of this d blocks renders the process slow and memory dependent ($\approx$ 3 KByte generation and storage for $d \approx 300$). For synchronised operation, every reinitialisation of initial d blocks is a time consuming affair. Further, the array operation needs indirect addressing at machine level and becomes inefficient. Note that, it is not practical to write a program by hard coding d LFSR blocks by distinct variables for each LFSR where $d \approx 300$. The average number of logical operation per bit ($\frac{1}{32}$) for a trinomial in this method is theoretically less than that of our method ($\frac{1}{8}$) in Subsection 4.1.1. However, due to software overheads, in actual implementation this method runs much slower than our method. We have written computer programs and checked that the processing speed of our strategy is almost 10 times faster than this.

4.1.1 Our Proposal

We extend the idea of [38, 4]. According to this approach, the LFSR (length d) consists of y number of blocks, each of length b, i.e., $d = yb$. Here we denote the blocks $(S[y-1], \ldots, S[1], S[0])$ from left to right. Initially $S[y-1] = \mathbf{w}_{y-1}, \ldots, S[0] = \mathbf{w}_0$. The block oriented recurrence relation for the LFSR is [4]

$$\mathbf{w}_{j+y} = \bigoplus_{i'=0}^{t-1} \left((\mathbf{w}_{j+q_{i'}-1} << (b - r_{i'})) \oplus (\mathbf{w}_{j+q_{i'}} >> r_{i'}) \right), \text{ for } j \geq 0. \qquad \text{Equation (1)}$$

Here $bq_{i'} + r_{i'} = p_{i'}$, and $p_0, p_1, \ldots, p_{t-1}$ are the tap positions. It may be noed that taps are only between the positions 0 and $d-b+1$. Number of logical operations required for each output bit is $\frac{t_1+4t_2}{b}$. Here, t_1 is the number of boundary taps, i.e., at any of the positions $0, b, 2b, \ldots, d(b-1)$, and t_2 is the number of non-boundary taps. Total number of taps $t = t_1 + t_2$. Considering primitive trinomials, $t_1 = 1$, $t_2 = 1$. Thus for $b = 64$, the number of logical operation per bit is $\frac{5}{64}$. Also the memory requirment is only $y = \frac{d}{b}$ blocks instead of d blocks as mentioned earlier. Further the intialisation can be done directly without any precompuation.

So, for the present work we prefer this idea. Unfortunately, the method presented in [38, 4] works for LFSRs of size $d = yb$, i.e., the degree has to be multiple of block size. Here we extend this technique for LFSR of any length $d = yb + a, 0 < a < b$, where d may not be a multiple of the block size. For the sake of faster implementation, we consider that there is no tap between the positions $d-1$ and $(d-b-a+1)$. Here, any LFSR (call it M_d for convenience) of length $d = yb + a, 0 < a < b$ is assumed to consist of $y+1$ blocks, namely $(S[y], S[y-1], \ldots, S[0])$ from left to right. In order to make all the blocks of same length b, we pad the the leftmost $(b-a)$ bits of the block $S[y]$ by zeroes. So at any stage we denote its state by the bit vector $\mathbf{D}_j = \{0, \ldots, 0, s_{j+d-1}, \ldots s_{j+yb}\}, j \geq 0$. Initially, $S[y] = \mathbf{D}_0, S[y-1] = \mathbf{w}_{y-1}, \ldots, S[0] = \mathbf{w}_0$. The y blocks $S[y-1], \ldots, S[0]$ can be viewed as an LFSR ($M_{d'}$) of length $d' = yb$. Recurrence relation for LFSR ($M_{d'}$) is obtained from Equation (1). The output of the LFSR (M_d) is dependent on both $M_{d'}$ and $S[y]$. So, we need to define two simultaneous block oriented recurrence relations, for the LFSR M_d. These can be obtained as,

$$\mathbf{w}_{j+y} = (\mathbf{w'}_j << a) \oplus \mathbf{D}_{j-1} \text{ for } j \geq 0, \text{ for } M_{d'} \text{ and}$$

$$\mathbf{D}_j = \mathbf{w'}_j >> (b - a) \text{ for } j \geq 0 \text{ for single block } S[y].$$

Here, $\mathbf{w}'_j$ can be obtained from Equation (1) considering the LFSR $M_{d'}$.

Note that, $\frac{t_1+4t_2+3}{b}$ logical operations are required for each output bit. Thus for a trinomial and $b = 64$, $\frac{8}{64} = \frac{1}{8}$ logical operations are needed per output bit. The storage requirement is only $y + 1$ blocks and no precomputation is required for initaialisation of the blocks. Considering all the aspects we advocate this method for fast software implementation of an LFSR.

4.2 Software Implementation of Boolean Function

The truth table of the Boolean function can be implemented in software using a lookup table (of 2^n bits) which gives one bit output for n bit input. In order to combine the n output blocks generated by the n LFSRs, the ANF of Boolean function provides an interesting option. First we present the following construction method [37, 30].

Construction 4.1 *Let f be an (n, m, d, x) function $(m > \frac{n}{2} - 2)$,*

$$f = (1 \oplus X_n)f_1 \oplus X_n f_2,$$

where f_1, f_2 are both $(n-1)$-variable m-resilient functions.

$$\text{Let } F = X_{n+2} \oplus X_{n+1} \oplus f$$

$$\text{and } G = (1 \oplus X_{n+2} \oplus X_{n+1})f_1 \oplus (X_{n+2} \oplus X_{n+1})f_2 \oplus X_{n+2} \oplus X_n.$$

$$\text{Also } H = (1 \oplus X_{n+3})F \oplus X_{n+3}G.$$

The function H constructed from f above is an $(n+3, m+2, d+1, 2^{n+1}+4x)$ function. Moreover, F, G are both $(n+2)$-variable, $(m+2)$-resilient functions.

Now consider that the algebraic normal form of f_1, f_2 are available and they need l_1, l_2 number of logical operations (AND, OR) respectively. Following Construction 4.1, we require l_1+l_2+16 logical operations to derive the function H. Table 5.2 in Appendix A gives stepwise breakup of required logical operations. We consider the input variables $(X_1, \ldots, X_n)$ are n output blocks from the LFSRs. So the output of the Boolean function is one block of b bits after $l_1 + l_2 + 16$ operations. Thus b bits are produced in $l_1 + l_2 + 16$ operations, which gives that on average, one bit is generated in $\frac{l_1+l_2+16}{b}$ operation(s).

4.3 Operational Details

The key initialisation involves intialisation of the blocks of the n LFSRs. Later we discussed a key scheduling algorithm in Subsection 5.1 for a 128/192/256 bit system. The detailed scheme described in Section 5.

After initialisation of the blocks of each LFSR, the generation of the running keystream starts. Following the block oriented operation (Subsection 4.1), each LFSR is executed once to generate one output block. The n output blocks from n LFSRs are combined using the method described in Subsection 4.2 to generated one running key block. Considering

trinomials as connection polynomials, each LFSR needs just 8 logical operation (5 XORs and 3 bit shiftings) to output one block of b bits. Taking into account the the operations for Boolean function (Subsection 4.2), one block of running key sequence K is generated after $8n + l_1 + l_2 + 16$ operations. So we require $\frac{8n+l_1+l_2+16}{b}$ number of logical operation for generating each bit of the key sequence.

Note that for our specific scheme, $l_1 = l_2 = 0, n = 6, b = 64$ and hence we get 1 bit per logical operation. However the compiler can utilise the parallelisation capacity of the present generation of processors to execute more than one logical operations in parallel and thereby increasing overall speed to a great extent.

5 A concrete scheme

On the basis of the design criteria discussed so far, we propose a specific scheme of the nonlinear combiner model with key length $q = 256$. The scheme can also be used for 128/192 bit key size. We consider a block size of $b = 64$ bits. The available data type "unsigned long long" in gcc/cc compilers provides this 64 bit storage block. As discussed in Section 3, we start with $L = 4q = 1024$. However, considering $\epsilon = 0.125$ (i.e., $p = 0.375$), the actual value of L can be decreased to ≈ 1000. We take a $(6, 2, 3, 24)$ Boolean function. Here $n = 6$, $m = 2$ satisfies the conditions (i) $m > \frac{n}{2} - 2$, and (ii) $2^{m-n+1} \leq \epsilon$. We follow the Construction 4.1 with $f = (1 \oplus X_3)X_1 \oplus X_3X_2$, i.e., $f_1 = X_1, f_2 = X_2$. In this case no operation is required to calculate f_1, f_2 and hence $l_1 = l_2 = 0$ and 16 logical operations are needed to get an output of 1 block for input of 6 blocks coming from the LFSRs.

The average degree of the connection polynomials are $d_{av} = \frac{1000}{3} \approx 333$. We choose the LFSRs of length 332, 333, 337, 343, 353, 377 and take the corresponding primitive trinomials (see Table 6 in Appendix A). Note that the degrees are mutually coprime. Considering the three $(m + 1 = 2 + 1 = 3)$ least lengths out of these 6 LFSRs, $L = 1002$, which is greater 1000, our initial design assumption.

Take the initial state vector considering all the LFSRs, i.e., in total 2075 bits. Since we choose the LFSRs of degree mutually coprime, this state will return once again after $(2^{332}-1)(2^{333}-1)(2^{337}-1)(2^{343}-1)(2^{353}-1)(2^{377}-1) \approx 2^{2075}$ states. Moreover, testing the generated running key sequence (K) for length upto $60,000$, we always observed the linear complexity [9] around half of its length.

Now we look at the possible minimum weights of the connection polynomial $(\psi(x))$ corresponding to the equivalent LFSRs (see Table 8 in Appendix A). This is obtained for different subsets (size m') of the 6 LFSRs. Here $3 \leq m' \leq 6$. The least weight of the products of m' connection polynomials is 25 (> 10). Clearly the criteria given in Fact 3.4 is satisfied and no fast correlation attack is possible in the line of [26]. From [24], the expected degree of the least degree τ-nomial multiple for $L = 1002$ is $2^{\frac{1002}{\tau-1}}$ which is approximately $2^{500}, 2^{334}, 2^{250}$ for $\tau = 3, 4, 5$. Finding exact multiples of such high degrees seems to be very hard. Moreover, Table 5.2 in Appendix A presents the actual complexities of the CT [2] attack for our scheme and it is always $\geq 2^{256}$. We have already discussed (see Table 4 in Appendix A) the robustness of this scheme against the CJS attack.

The concrete scheme with specific parameters are presented here to show the way of evolving a safe stream cipher scheme using the criteria fixed in Section 3. We have taken

trinomials (least possible weight) as connection polynomials of individual LFSRs and still get the desired level of security. Similar schemes with different number of LFSRs and Boolean functions or for different connection polynomials for the present scheme can be designed in the same way depending on the security requirement (i.e., key length q) of the user. This flexibility is an interesting feature of the nonlinear combiner model.

5.1 Key Scheduling

We need to initialise the 6 blocks for each of the 6 LFSRs and in total 36 blocks each of 64 bits. We consider the 256-bit key string, as 4 blocks of 64-bits $\{\mathcal{K}_3, \mathcal{K}_2, \mathcal{K}_1, \mathcal{K}_0\}$. We need to expand this 4 blocks to 36 blocks. We label the 6 LFSRs as $S_1, \ldots, S_6$. The six 64 bit blocks of LFSR S_i are denoted as $S_i[j]$, $0 \leq j \leq 5$. Now we initialize the 36 blocks as follows in the increasing ordering of i, j, putting the blocks of an LFSR together in parenthesis.

$$(\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3, \mathcal{K}_4, \mathcal{K}_2 \oplus \mathbf{1}, \mathcal{K}_3),\ (\mathcal{K}_4 \oplus \mathbf{1}, \mathcal{K}_1 \oplus \mathbf{1}, \mathcal{K}_3 \oplus \mathbf{1}, \mathcal{K}_4 \oplus \mathbf{1}, \mathcal{K}_1, \mathcal{K}_2 \oplus \mathbf{1}),$$
$$(\mathcal{K}_4 \oplus \mathbf{1}, \mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3 \oplus \mathbf{1}, \mathcal{K}_1 \oplus \mathbf{1}, \mathcal{K}_4 \oplus \mathbf{1}),\ (\mathcal{K}_3 \oplus \mathbf{1}, \mathcal{K}_2, \mathcal{K}_4, \mathcal{K}_3 \oplus \mathbf{1}, \mathcal{K}_2, \mathcal{K}_1 \oplus \mathbf{1}),$$
$$(\mathcal{K}_3, \mathcal{K}_2, \mathcal{K}_1 \oplus \mathbf{1}, \mathcal{K}_4 \oplus \mathbf{1}, \mathcal{K}_2 \oplus \mathbf{1}, \mathcal{K}_1),\ (\mathcal{K}_4, \mathcal{K}_3, \mathcal{K}_3, \mathcal{K}_2 \oplus \mathbf{1}, \mathcal{K}_4 \oplus \mathbf{1}, \mathcal{K}_1).$$

Note that the leftmost block of each LFSR is $S_i[5]$. In each case we need to place zero at the leftmost $(b-a)$ bits. Considering the value in $S_i[5]$ as V_i, the simple operation $V_i = V_i >> (b-a)$ does this task. If the key is a 128-bit one $\mathcal{K}_1, \mathcal{K}_2$, we first expand it to a 256-bit key by $\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_2 \oplus \mathbf{1}, \mathcal{K}_1 \oplus \mathbf{1}$ and proceed as before. Similarly 192-bit key $\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3$ is expanded to 256-bit key by taking $\mathcal{K}_4 = \mathcal{K}_2 \oplus \mathbf{1}$.

We can incorporate 128 bit initialization vector IV_1, IV_2, each of 64 bits by XORing them with two specifically chosen blocks of each LFSR except the leftmost block. This will help in rekeying. Similarly self synchronization can be obtained using similar strategy.

5.2 Performance in Software

We study the encryption speed of our specific scheme in software, using a simple "C" language implementation available at Appendix B. Here $b = 64$ bits. So $(8 \times 6 + 16 = 64)$ logical operations are required to generate one block of running key stream K. Thus we need $\frac{64}{8} = 8$ logical operations for generating one byte output. The storage requirement for this implementation is less than $1KB$ for storing the LFSRs. The encryption speed of the scheme in different platforms is presented in Table 5.2. For comparison purpose we also present the encryption speed of other stream cipher schemes in Table 2. It is clear that our scheme presents a much better speed.

We should clarify why our proposed scheme achieves such a high speed. When compiled using "gcc -O3" or "cc -O3", the executable code becomes optimized and hence it can exploit the inherent parallelization of the high end processors. We have also measured the number of cycles required for generation of each output byte. Note that the theoretical calculation gives that 8 logical operations are needed to get one byte output. However, we get the value 0.20 on Silicon Graphics platform. This has been done using "perfex", the command line interface for processor activity counter in IRIX (Release 6.5) operating system. The tests were carried out with 400 MHz MIPS R12000(IP32) Processor, 512 MB memory and 32 MB data cache. We compare the results with that for some other schemes in Table 5.2.

Processor/RAM	Operating System	Compiler	Speed (Gbps)
PIII 666 MHz/ 128 MB	Linux (RH 7.0)	gcc -O3	18.18
PIII 1 GHZ/128 MB	Linux (RH 6.2)	gcc -O3	29.00
PIV 1.8 GHZ/512 MB	Linux (RH 6.2)	gcc -O3	50.00
MIPS R12000(IP32) 400 Mhz/512 MB	IRIS Release 6.5	gcc -O3	19.63
SUNULTRA 60 SPARC 2 360 MHz/256 MB	SUN OS	cc -O3	71.0

Table 1: Keystream Generation Speed (Gbits/sec) on various platforms.

Scheme	Processor/ OS	Compiler	*Speed (Gbps)
LILI-II [5]	PIII 300MHz/Win NT SPARCv9/Linux	Borland C++ 5.0 gcc	0.006 0.004
SSC2 [38]	Sun Ultra 1.143 MHz/Sun Solaris PII 233 MHz/Linux Sun SPARK2 40 MHz/Sun	gcc -O3 gcc -O3 gcc-O3	0.143 0.118 0.022
SNOW-1.0 [10]	PIII 500 MHz/Linux PIII 500 MHz/Win NT PII 400 MHz/Linux	gcc -O3 VC++ gcc -O3	0.610 0.520 0.380
SNOW-2.0 [11]	PIV 1.8 GHz/Linux	gcc -O3	3.610
SOBER-*t*32 t16 [32]	Sun Ultra Sparc 200 MHz/ Sun Solaris	-	0.076 0.044

Table 2: Performance of Different Software stream Ciphers.

Acknowledgment : This work has been supported partially by the ReX program, a joint activity of USENIX Association and Stichting NLnet.

References

[1] A. Canteaut and E. Filiol. Ciphertext Only Reconstruction of Stream Ciphers based on Combination Generators. In *Proceedings of FSE 2000*, LNCS volume 1978, 2001.

[2] A. Canteaut and M. Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In *Advances in Cryptology - EUROCRYPT 2000*, LNCS Volume 1807, pages 573–588. Springer Verlag, 2000.

[3] V. V. Chepyzhov, T. Johansson and B. Smeets. A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers. In *Proceedings of FSE 2000*, LNCS volume 1978. Springer Verlag, 2001.

[4] S. Chowdhury and S. Maitra. Efficient Software implementation of LFSR. in *INDOCRYPT 2001*, LNCS volume 2247. Springer Verlag, 2000.

Scheme	Processor/OS	cycles / byte
SCREAM	PIII 550 Mhz/ Linux	4.9
SEAL	- do-	5.0
SNOW 1.0	PIV 1.8 GHZ/ Linux	8.5
SNOW 2.0	- do-	4.5
Our Scheme	full details given above	0.20

Table 3: Number of operations (cycles/byte) required for key generation in optimized code.

[5] A. Clark, E. Dawson, J. Fuller, J. D. Golic, H. -J. Lee, W. Millan, S. -J. Moon, L. Simpson. The LILI-II Keystream Generator In *Information Security and Privacy- ACISP 2002*, LNCS volume 2384. Springer Verlag, 2002.

[6] D. Coppersmith, S. Halevi and C. Jutla. Cryptanalysis of stream ciphers with linear masking. *Advances in Cryptology - CRYPTO 2002*, LNCS, Springer Verlag, 2002.

[7] N. T. Courtois and W. Meier. Higher order correlation attacks, XL algorithm, and cryptanalysis of Toyocrypt. Accepted in ICISC 2002, to be published in LNCS.

[8] N. T. Courtois, and W. Meier. Algebraic attack on Stream Ciphers with linear feedback. Private correspondence, 2002.

[9] C. Ding, G. Xiao, and W. Shan. *The Stability Theory of Stream Ciphers.* LNCS volume 561. Springer-Verlag, 1991.

[10] P. Ekdahl and T. Johansson. SNOW - a New Stream Cipher. In *Proceedings of the first open NESSIE Workshop*, Heverlee, Belgium, November 13-14, 2000.

[11] P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In *SAC 2002*, August 2002, in pre-proceedings.

[12] E. Filiol. Decimation attack of stream ciphers. In *Progress in Cryptology - INDOCRYPT 2000*, LNCS volume 1977, pages 31–42. Springer Verlag, 2000.

[13] S. Fluhrer, I. Mantin and A. Shamir. *Weaknesses in key scheduling Algorithm of RC4* . at the *Eighth Annual Workshop on Selected Areas in Cryptography*, August, 2001.

[14] S. W. Golomb. Shift Register Sequences. Aegean Park Press, 1982.

[15] X. Guo-Zhen and J. Massey. A spectral characterization of correlation immune combining functions. *IEEE Transactions on Information Theory*, 34(3):569–571, May 1988.

[16] P. Hawkes, G. G. Rose. Guess-and-Determine Attacks on SNOW. In *SAC 2002*, August 2002, in pre-proceedings.

[17] P. Hawkes, G. G. Rose. Exploiting Multiples of the Connection Polynomial in Word-Oriented Stream Ciphers. In *Advances in Cryptology - ASIARCYPT 2000*, LNCS volume 1976, pages 303–316. Springer Verlag, 2000.

[18] P. Hawkes, F. Quick, G. G. Rose. A Practical Cryptanalysis of SSC2. In *SAC 2001*, in LNCS volume 2259. Springer Verlag, 2001.

[19] S. Halevi, D. Coppersmith and C. Jutla. Scream: A Software-Efficient Stream Cipher. In *FSE 2002*, LNCS volume 2365. Science, Springer Verlag, 2002.

[20] T. Johansson and F. Jonsson. Improved Fast Correlation Attacks on Stream Ciphers via Convolutional Codes. In *Advances in Cryptology - EUROCRYPT 1999*, LNCS volume 1952, pages 347–362. Springer Verlag, 1999.

[21] T. Johansson and F. Jonsson. Fast Correlation Attacks based on Turbo Code Techniques. In *Advances in Cryptology - CRYPTO 1999*, LNCS volume 1666, pages 181–197. Springer Verlag, 1999.

[22] T. Johansson and F. Jonsson. Fast correlation attacks through reconstruction of linear polynomials. In *Advances in Cryptology - CRYPTO 2000*, LNCS volume 1880, pages 300–315. Springer Verlag, 2000.

[23] R. Lidl and H. Niederreiter. Introduction to finite fields and their applications. Cambridge University Press, 1994.

[24] S. Maitra, K. C. Gupta and A. Venkateswarlu. Multiples of Primitive Polynomials and Their Products over GF(2). In *SAC 2002*, August 2002, pages 218–234 in preproceedings.

[25] I. Mantin and A. Shamir. *A Practical Attack on RC4* . In Preproceedings of *Fast Software Encryption Workshop, 2001*

[26] W. Meier and O. Stafflebach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1:159–176, 1989.

[27] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[28] M. J. Mihaljevic, M. P. C. Fossorier and H. Imai. A low-complexity and high performance algorithm for the correlation attack. In *Proceedings of FSE 2000*, LNCS volume 1978, 2001.

[29] M. J. Mihaljevic, M. P. C. Fossorier and H. Imai. Fast correlation attack algorithm with list decoding and an application. In *Proceedings of FSE 2001*, LNCS.

[30] E. Pasalic, S. Maitra, T. Johansson and P. Sarkar. New constructions of resilient and correlation immune Boolean functions achieving upper bounds on nonlinearity. In *Workshop on Coding and Cryptography*, Paris, January 2001.

[31] P. Rogaway, D. Coppersmith. A Software-optimized encryption Algorithm. In *Journal of Cryptology*, volume 11(4), 1998.

[32] G. Rose and P. Hawkes. The t-Class of SOBER Stream Ciphers. In *Proceedings of the first open NESSIE Workshop*, Heverlee, Belgium, 2000.

[33] R. Rueppel and O. Staffelbach. Products of Linear Recurring Sequences with Maximum Complexity. IEEE Transactions on Information Theory. IT-33(1): 124-131, 1987

[34] P. Sarkar and S. Maitra. Nonlinearity bounds and constructions of resilient Boolean functions. In *Advances in Cryptology - CRYPTO 2000*, LNCS volume 1880, pages 515–532. Springer Verlag, 2000.

[35] T. Siegenthaler. Correlation-immunity of nonlinear combining functions for cryptographic applications. *IEEE Transactions on Information Theory*, IT-30(5):776–780, September 1984.

[36] T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on Computers*, C-34(1):81–85, January 1985.

[37] Y. V. Tarannikov. On resilient Boolean functions with maximum possible nonlinearity. In *Proceedings of INDOCRYPT 2000*, LNCS volume 1977, 19-30, 2000.

[38] M. Zhang, C. Carrol and A. Chan. The software oriented stream cipher SSC2. In *Proceedings of FSE 2000*, LNCS volume 1978, 2001.

Appendix A : Tables

k	t	Bit Requirement (N)	Preporcessing Complexity	Decoding complexity
		$\frac{1}{4}(2k\,t!\,\ln 2)^{1/t}\epsilon^{-2}2^{\frac{L-k}{t}}$	$N^{\lceil\frac{t-1}{2}\rceil}$	$2^k \cdot k\frac{2\ln 2}{(2\epsilon)^{2t}}$
32	2	2^{491}	2^{491}	2^{40}
32	3	2^{329}	2^{329}	2^{44}
64	2	2^{476}	2^{476}	2^{72}
64	3	2^{319}	2^{319}	2^{76}
96	2	2^{460}	2^{460}	2^{104}
96	3	2^{309}	2^{309}	2^{108}
128	2	2^{444}	2^{444}	2^{136}
128	3	2^{298}	2^{298}	2^{140}
160	2	2^{428}	2^{428}	2^{168}
160	3	2^{287}	2^{287}	2^{172}
192	2	2^{413}	2^{413}	2^{200}
192	3	2^{277}	2^{277}	2^{204}
224	2	2^{397}	2^{397}	2^{232}
224	3	2^{266}	2^{266}	2^{236}
256	2	2^{381}	2^{381}	2^{264}
256	2	2^{256}	2^{256}	2^{268}

Table 4: Time complexity for Chepyzhov-Johansson-Smeets Attack [3] ($L = 100, \epsilon = 0.125$)

Algebraic Normal Form	&	⊕	Total
$f = (1 \oplus X_n) f_1 \oplus X_n f_2$	2	2	4
$F = X_{n+2} \oplus X_{n+1} \oplus f$	0	2	2
$G = (1 \oplus X_{n+2} \oplus X_{n+1}) f_1 \oplus (X_{n+2} \oplus X_{n+1}) f_2 \oplus X_{n+2} \oplus X_n$ $= f_1 \oplus (X_{n+2} \oplus X_{n+1})(f_1 \oplus f_2) \oplus X_{n+2} \oplus X_n$	1	5	6
$H = (1 \oplus X_{n+3}) F \oplus X_{n+3} G$	2	2	4
Total logical operations			16

Table 5: Construction of Boolean function

Length	Connection Polynomial
332	$x^{332} \oplus x^{123} \oplus 1$
333	$x^{333} \oplus x^{331} \oplus 1$
337	$x^{337} \oplus x^{135} \oplus 1$
343	$x^{343} \oplus x^{205} \oplus 1$
353	$x^{353} \oplus x^{69} \oplus 1$
377	$x^{377} \oplus x^{75} \oplus 1$

Table 6: Details of the LFSRs

Appendix B : The code

We present the complete code for key generation without key initialization.

```
#include <stdio.h>
#include <time.h>
typedef unsigned long long ull;

main(){
/*** 64 bit blocks for the 6 LFSRs ***/
ull
W0_bl0, W0_bl1, W0_bl2, W0_bl3, W0_bl4, D0,
W1_bl0, W1_bl1, W1_bl2, W1_bl3, W1_bl4, D1,
W2_bl0, W2_bl1, W2_bl2, W2_bl3, W2_bl4, D2,
W3_bl0, W3_bl1, W3_bl2, W3_bl3, W3_bl4, D3,
W4_bl0, W4_bl1, W4_bl2, W4_bl3, W4_bl4, D4,
```

τ	Cipherbits	Memory Req.	Proprocessing Comp.	Decoding Comp.
	$N = 2^{\alpha_\tau(p) + \frac{L}{\tau-1}}$	$2N + (\tau - 1) \cdot m(d)$	$\frac{N^{\tau-2}}{(\tau-2)!}$	$5(\tau - 1) \cdot N \cdot m(\tau)$
3	2^{503}	2^{504}	2^{503}	2^{512}
4	2^{337}	2^{338}	2^{673}	2^{349}
5	2^{254}	2^{255}	2^{760}	2^{271}

Table 7: Time/ Memory complexity for Canteaut and Trabbia Attack [2]

No. of LFSRs taken (m')	Minimum weight	Corresponding degree
3	25	1062
4	75	1395
5	209	1698
6	495	2075

Table 8: Minimum weight of the connection polynomial of composite LFSRs.

```
W5_bl0, W5_bl1, W5_bl2, W5_bl3, W5_bl4, D5;

/*** w[j+d] corresponding to each LFSR ***********/
ull  newbit0, newbit1, newbit2, newbit3, newbit4, newbit5;

/*** ouput block corresponding to each LFSR ***********/
ull  out0, out1 ,out2 ,out3, out4, out5;

/***w'[j] corresponding to each LFSR ***********/
ull  temp0, temp1, temp2, temp3, temp4, temp5 ;

/****** Boolean funcction variable ************/
ull  f, G, F, output ;
ull ALLONE= 18446744073709551615;     /* AllONE = 2^64-1 */

long   blNo = 1024*1024*32;        /* No. of output blocks req */
long   loop = 0;

clock_t t1, t2;     /* for measuring time */
double tm;

/********Generation of output block *************************/

        t1 = clock();   /* time measurement starts */

while (loop < blNo ){

 /** LFSR1**x^332 + x^123 + x^0 (a = 12, y = 5, r0 =  0, r1 = 59)********/

temp0 = W0_bl0 ^ (W0_bl1 >> 59) ^ (W0_bl2 << 5);  /* W'_k */
newbit0 = (temp0 << 12) ^ D0;        /* W_{k+d} */
D0 =  temp0 >> 52;                  /* b-a = 52 */

out0 = W0_bl0;                       /* output block */
```

```
/**** shifting registers by one block ******/

W0_bl0 = W0_bl1;
W0_bl1 = W0_bl2;
W0_bl2 = W0_bl3;
W0_bl3 = W0_bl4;
W0_bl4 = newbit0;

/** LFSR2**x^333 + x^2 + x^0 (a = 13, y = 5, r0 = 0, r1 = 2)********/

temp1 = W1_bl0 ^ (W1_bl0 >> 2) ^ (W1_bl1 << 62);
newbit1 = (temp1 << 13) ^ D0;
D1 =  temp1 >> 51;                /* b-a = 51 */

out1 = W1_bl0;                     /* output block */

/**** shifting registers by one block ******/

W1_bl0 = W1_bl1;
W1_bl1 = W1_bl2;
W1_bl2 = W1_bl3;
W1_bl3 = W1_bl4;
W1_bl4 = newbit1;

/***LFSR3*x^337 + x^135 + x^0 (a = 17, y = 5, r0 = 0, r1 = 7 )******/

temp2 = W2_bl0 ^ (W2_bl2 >> 7 ) ^ (W2_bl3 << 57 );
newbit2 = (temp2 << 17 ) ^ D2;
D2 =  temp2 >> 47;           /* b-a = 47 */

out2 = W2_bl0;                /* output block */

/**** shifting registers by one block ******/

W2_bl0 = W2_bl1;
W2_bl1 = W2_bl2;
W2_bl2 = W2_bl3;
W2_bl3 = W2_bl4;
W2_bl4 = newbit2;

/**LFSR4 *x^343 + x^205 + x^0 (a = 23, y = 5, r0 = 0, r1 = 13 )******/

temp3 = W3_bl0 ^ (W3_bl3 >> 13 ) ^ (W3_bl4 << 51 );
newbit3 = (temp3 << 23) ^ D3;
D3 =  temp3 >> 41;                /* b-a = 41 */
```

```
out3 = W3_bl0;                     /* output block */

/**** shifting registers by one block ******/

W3_bl0 = W3_bl1;
W3_bl1 = W3_bl2;
W3_bl2 = W3_bl3;
W3_bl3 = W3_bl4;
W3_bl4 = newbit3;

/**LFSR5*x^353 + x^69 + x^0 (a = 33, y = 5, r0 = 0, r1 = 5 )******/

temp4 = W4_bl0 ^ (W4_bl1 >> 5 ) ^ (W4_bl2 << 59 );
newbit4 = (temp4 << 33) ^ D4;
D4 =  temp4 >> 31;                 /* b-a = 31 */

out4 = W4_bl0;                     /* output block */

/**** shifting registers by one block ******/

W4_bl0 = W4_bl1;
W4_bl1 = W4_bl2;
W4_bl2 = W4_bl3;
W4_bl3 = W4_bl4;
W4_bl4 = newbit4;

/**LFSR6 *x^377 + x^75 + x^0 (a = 57, y = 5, r0 = 0, r1 = 11 )******/

temp5 = W5_bl0 ^ (W5_bl1 >> 11 ) ^ (W5_bl2 << 53 );
newbit5 = (temp5 << 57 ) ^ D4;
D5 =  temp5 >> 7;                     /* b-a = 7 */

out5 = W5_bl0;                         /* output block */

/**** shifting registers by one block ******/

W5_bl0 = W5_bl1;
W5_bl1 = W5_bl2;
W5_bl2 = W5_bl3;
W5_bl3 = W5_bl4;
W5_bl4 = newbit5;

/***********Boolean function**************/
```

```
 /*** f= (1 + X3)f1 + (X2.f2), f1 = X0, f2 = X1 *****/

f = ( ALLONE ^ out2 ) & out0 ^  ( out2 & out1 );

  /***  F = X5 + X4 + f ********/

F = out4 ^ out3 ^ f;

/*** G = f1 + (X5 + X4 )(f1 + f2) + X5 + X2 ***/

G =  out0 ^ ( out4 ^ out3 ) & ( out1 ^ out0 ) ^ out4 ^ out2;
/*** output, H = (1 + X6 ) F + X6 G *************/

output =  ( ALLONE  ^ out5 ) & F ^ ( out5 & G );

/**********************************************************/

loop ++ ;

    } /* end of while loop */

    t2 = clock(); /* end of time measurement*/
    tm = ((double)(t2 - t1))/((double) CLOCKS_PER_SEC);
    printf ("Time taken %10.3lf sec.\n", tm);

   /************ speed in Gigabits persec *****************/
    printf ("Speed = %lf Gbps\n", (32.0 * 64.0)/(tm * 1024.0));

}
```